



Language Reference



VERSION 7.5

***Visual* dBASE[®] for Windows 95, 98 and NT**

dBASE, Inc. • Vestal, NY • Santa Cruz, CA
<http://www.dbase2000.com> • <news://news.dbase2000.com>

dBASE Inc. or Borland International may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT © 1996 Borland International, 1999 dBASE Inc. All rights reserved. All dBASE product names are trademarks or registered trademarks of dBASE Inc. All Borland product names are trademarks or registered trademarks of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Printed in the U.S.A.

Contents

Chapter 1

Introduction 1-1

| | |
|-------------------------------------|-----|
| How this book is organized. | 1-1 |
| Typographical conventions. | 1-2 |
| Using the online version | 1-3 |

Chapter 2

Language definition 2-1

| | |
|--|------|
| Basic attributes | 2-1 |
| Data types. | 2-2 |
| Simple data types. | 2-2 |
| String data | 2-2 |
| Numeric data | 2-3 |
| Logical data | 2-3 |
| Date data. | 2-4 |
| Null values | 2-4 |
| Database-specific data types | 2-4 |
| Memo data. | 2-4 |
| Binary and OLE data | 2-5 |
| Programming data types | 2-5 |
| Operators and symbols | 2-5 |
| Names. | 2-6 |
| Expressions | 2-6 |
| Basic expressions | 2-6 |
| Variables. | 2-7 |
| Assigning variables. | 2-7 |
| Using variables and field names in expressions | 2-7 |
| Type conversion | 2-8 |
| Automatic type conversion | 2-8 |
| Explicit type conversion | 2-8 |
| Arrays | 2-8 |
| Literal arrays | 2-9 |
| Complex expressions | 2-9 |
| Statements | 2-10 |
| Basic statements | 2-10 |
| Control statements | 2-11 |
| Functions and codeblocks. | 2-11 |
| Function pointers. | 2-12 |
| Codeblocks | 2-12 |
| Codeblocks vs. functions. | 2-13 |
| Objects and classes. | 2-14 |
| Dynamic subclassing. | 2-14 |
| Methods | 2-15 |
| A simple class. | 2-15 |
| Programs | 2-16 |

| | |
|-----------------------------------|------|
| Program files. | 2-16 |
| Program execution | 2-16 |
| Functions and classes | 2-17 |
| Comments | 2-17 |
| Preprocessor directives | 2-17 |
| A simple program | 2-18 |

Chapter 3

Syntax conventions 3-1

| | |
|------------------------------------|-----|
| Syntax notation | 3-1 |
| Syntax example | 3-2 |
| Capitalization guidelines. | 3-3 |
| SET command defaults. | 3-4 |

Chapter 4

Operators and symbols 4-1

| | |
|---|------|
| Operator precedence | 4-2 |
| Assignment operators | 4-3 |
| + ("plus") operator | 4-4 |
| - ("minus") operator | 4-5 |
| Numeric operators | 4-6 |
| Logical operators | 4-7 |
| Comparison operators | 4-7 |
| Object operators. | 4-10 |
| NEW operator | 4-10 |
| Index operator | 4-10 |
| Dot operator | 4-11 |
| Scope resolution operator | 4-11 |
| Call, indirection, grouping operator | 4-12 |
| Alias operator | 4-12 |
| Macro operator | 4-13 |
| Non-operational symbols | 4-16 |
| String delimiters. | 4-16 |
| Name/database delimiters. | 4-16 |
| Comment symbols | 4-17 |
| Statement separator, line continuation | 4-17 |
| Codeblock, literal date, literal array symbol | 4-18 |
| Preprocessor directive symbol | 4-19 |

Chapter 5

Core language 5-1

| | |
|---------------------------|-----|
| class Exception | 5-1 |
| class Object | 5-3 |
| ARGCOUNT() | 5-3 |
| ARGVECTOR() | 5-4 |
| CASE | 5-5 |

| | |
|----------------------------|------|
| CATCH | 5-5 |
| CLASS. | 5-5 |
| <i>className</i> | 5-7 |
| CLEAR MEMORY | 5-7 |
| CLEAR PROGRAM | 5-7 |
| CLOSE PROCEDURE | 5-8 |
| DEFINE | 5-9 |
| DO | 5-10 |
| DO CASE | 5-12 |
| DO WHILE | 5-14 |
| DO...UNTIL. | 5-15 |
| ELSE. | 5-17 |
| ELSEIF | 5-17 |
| EMPTY() | 5-17 |
| EXIT | 5-18 |
| FINALLY | 5-19 |
| FINDINSTANCE() | 5-19 |
| FOR...ENDFOR. | 5-20 |
| FUNCTION. | 5-21 |
| IF. | 5-23 |
| IIF() | 5-24 |
| LOCAL | 5-25 |
| LOOP | 5-26 |
| OTHERWISE | 5-26 |
| PARAMETERS. | 5-27 |
| <i>parent</i> | 5-30 |
| PCOUNT() | 5-31 |
| PRIVATE | 5-31 |
| PROCEDURE. | 5-32 |
| PUBLIC | 5-33 |
| QUIT | 5-34 |
| REDEFINE | 5-34 |
| RELEASE | 5-35 |
| <i>release()</i> | 5-36 |
| RELEASE OBJECT | 5-37 |
| RESTORE | 5-37 |
| RETURN | 5-38 |
| SAVE | 5-38 |
| SET LIBRARY | 5-39 |
| SET PROCEDURE | 5-40 |
| SET() | 5-42 |
| SETTO() | 5-43 |
| STATIC | 5-43 |
| STORE. | 5-45 |
| THROW. | 5-46 |
| TRY | 5-46 |
| TYPE() | 5-50 |
| WITH | 5-52 |

Chapter 6 String objects

6-1

| | |
|---------------------------------------|------|
| <code>class String</code> | 6-1 |
| <code>ASC()</code> | 6-4 |
| <code>asc()</code> | 6-4 |
| <code>AT()</code> | 6-5 |
| <code>CENTER()</code> | 6-6 |
| <code>charAt()</code> | 6-7 |
| <code>CHR()</code> | 6-7 |
| <code>chr()</code> | 6-8 |
| <code>DIFFERENCE()</code> | 6-8 |
| <code>getByte()</code> | 6-9 |
| <code>indexOf()</code> | 6-10 |
| <code>ISALPHA()</code> | 6-11 |
| <code>isAlpha()</code> | 6-11 |
| <code>ISLOWER()</code> | 6-11 |
| <code>isLower()</code> | 6-12 |
| <code>ISUPPER()</code> | 6-12 |
| <code>isUpper()</code> | 6-13 |
| <code>lastIndexOf()</code> | 6-13 |
| <code>LEFT()</code> | 6-14 |
| <code>left()</code> | 6-14 |
| <code>leftTrim()</code> | 6-15 |
| <code>LEN()</code> | 6-15 |
| <code>length</code> | 6-15 |
| <code>LIKE()</code> | 6-16 |
| <code>LOWER()</code> | 6-17 |
| <code>LTRIM()</code> | 6-17 |
| <code>PROPER()</code> | 6-17 |
| <code>RAT()</code> | 6-18 |
| <code>REPLICATE()</code> | 6-19 |
| <code>replicate()</code> | 6-20 |
| <code>RIGHT()</code> | 6-20 |
| <code>right()</code> | 6-21 |
| <code>rightTrim()</code> | 6-21 |
| <code>RTRIM()</code> | 6-21 |
| <code>setByte()</code> | 6-22 |
| <code>SOUNDEX()</code> | 6-23 |
| <code>SPACE()</code> | 6-24 |
| <code>space()</code> | 6-25 |
| <code>STR()</code> | 6-25 |
| <code>STUFF()</code> | 6-26 |
| <code>stuff()</code> | 6-27 |
| <code>SUBSTR()</code> | 6-27 |
| <code>substring()</code> | 6-28 |
| <code>toLowerCase()</code> | 6-29 |
| <code>toProperCase()</code> | 6-29 |
| <code>toUpperCase()</code> | 6-29 |
| <code>TRANSFORM()</code> | 6-30 |

| | |
|---------|------|
| TRIM() | 6-30 |
| UPPER() | 6-31 |
| VAL() | 6-31 |

Chapter 7

Math / Money

7-1

| | |
|---------------|------|
| ABS() | 7-1 |
| ACOS() | 7-1 |
| ASIN() | 7-2 |
| ATAN() | 7-2 |
| ATN2() | 7-3 |
| CEILING() | 7-3 |
| COS() | 7-4 |
| DTOR() | 7-4 |
| EXP() | 7-5 |
| FLOOR() | 7-5 |
| FV() | 7-6 |
| INT() | 7-7 |
| LOG() | 7-7 |
| LOG10() | 7-8 |
| MAX() | 7-8 |
| MIN() | 7-9 |
| MOD() | 7-10 |
| PAYMENT() | 7-10 |
| PI() | 7-11 |
| PV() | 7-11 |
| RANDOM() | 7-12 |
| ROUND() | 7-13 |
| RTOD() | 7-14 |
| SET CURRENCY | 7-15 |
| SET DECIMALS | 7-15 |
| SET POINT | 7-16 |
| SET PRECISION | 7-16 |
| SET SEPARATOR | 7-17 |
| SIGN() | 7-18 |
| SIN() | 7-19 |
| SQRT() | 7-19 |
| TAN() | 7-20 |

Chapter 8

Date and time objects

8-1

| | |
|-------------|-----|
| class Date | 8-2 |
| class Timer | 8-4 |
| CDOW() | 8-6 |
| CMONTH() | 8-7 |
| CTOD() | 8-7 |
| DATE() | 8-8 |
| DAY() | 8-8 |
| DMY() | 8-9 |

| | |
|----------------------------|------|
| DOW() | 8-9 |
| DTOC() | 8-10 |
| DTOS() | 8-11 |
| ELAPSED() | 8-11 |
| <i>enabled</i> | 8-12 |
| <i>getDate()</i> | 8-13 |
| <i>getDay()</i> | 8-14 |
| <i>getHours()</i> | 8-14 |
| <i>getMinutes()</i> | 8-15 |
| <i>getMonth()</i> | 8-15 |
| <i>getSeconds()</i> | 8-16 |
| <i>getTime()</i> | 8-16 |
| <i>getTimezoneOffset()</i> | 8-17 |
| <i>getYear()</i> | 8-18 |
| <i>interval</i> | 8-18 |
| MDY() | 8-19 |
| MONTH() | 8-19 |
| <i>onTimer</i> | 8-20 |
| <i>parse()</i> | 8-21 |
| SECONDS() | 8-22 |
| SET CENTURY | 8-22 |
| SET DATE | 8-23 |
| SET DATE TO | 8-24 |
| SET EPOCH | 8-24 |
| SET MARK | 8-25 |
| SET TIME | 8-25 |
| <i>setDate()</i> | 8-25 |
| <i>setHours()</i> | 8-26 |
| <i>setMinutes()</i> | 8-26 |
| <i>setMonth()</i> | 8-26 |
| <i>setSeconds()</i> | 8-27 |
| <i>setTime()</i> | 8-27 |
| <i>setYear()</i> | 8-28 |
| TIME() | 8-28 |
| <i>toGMTString()</i> | 8-29 |
| <i>toLocaleString()</i> | 8-29 |
| <i>toString()</i> | 8-30 |
| UTC() | 8-30 |
| YEAR() | 8-31 |

Chapter 9

Bitwise

9-1

| | |
|-------------|-----|
| BITAND() | 9-1 |
| BITLSHIFT() | 9-2 |
| BITNOT() | 9-3 |
| BITOR() | 9-4 |
| BITRSHIFT() | 9-4 |
| BITSET() | 9-5 |
| BITXOR() | 9-5 |

| | |
|--------------|-----|
| BITZRSHIFT() | 9-6 |
| Htoi() | 9-7 |
| ITOH() | 9-7 |

Chapter 10

Array objects 10-1

| | |
|------------------|-------|
| Array functions | 10-2 |
| class Array | 10-2 |
| class AssocArray | 10-5 |
| ACOPY() | 10-7 |
| add() | 10-8 |
| ALen() | 10-9 |
| count() | 10-9 |
| delete() | 10-10 |
| dimensions | 10-13 |
| dir() | 10-14 |
| dirExt() | 10-16 |
| element() | 10-18 |
| fields() | 10-19 |
| fill() | 10-20 |
| firstKey | 10-21 |
| grow() | 10-22 |
| insert() | 10-25 |
| isKey() | 10-28 |
| nextKey() | 10-29 |
| removeAll() | 10-30 |
| removeKey() | 10-30 |
| resize() | 10-31 |
| scan() | 10-35 |
| size | 10-36 |
| sort() | 10-37 |
| subscript() | 10-40 |

Chapter 11

File/OS 11-1

| | |
|-----------------------------|-------|
| File commands and functions | 11-1 |
| File utility commands | 11-1 |
| File information functions | 11-2 |
| Low-level file functions | 11-2 |
| class File | 11-3 |
| ! | 11-5 |
| accessDate() | 11-6 |
| CD | 11-6 |
| close() | 11-7 |
| copy() | 11-8 |
| COPY FILE | 11-9 |
| create() | 11-9 |
| createDate() | 11-11 |
| createTime() | 11-11 |

| | |
|----------------|-------|
| date() | 11-12 |
| delete() | 11-13 |
| DELETE FILE | 11-14 |
| DIR | 11-14 |
| DISKSPACE() | 11-15 |
| DISPLAY FILES | 11-15 |
| DOS | 11-16 |
| eof() | 11-16 |
| ERASE | 11-17 |
| error() | 11-18 |
| exists() | 11-19 |
| FILE() | 11-19 |
| flush() | 11-20 |
| FNAMEMAX() | 11-20 |
| FUNIQUE() | 11-21 |
| GETDIRECTORY() | 11-22 |
| GETENV() | 11-22 |
| GETFILE() | 11-22 |
| gets() | 11-23 |
| handle | 11-25 |
| HOME() | 11-25 |
| LIST FILES | 11-26 |
| MD | 11-26 |
| MKDIR | 11-27 |
| open() | 11-27 |
| OS() | 11-28 |
| path | 11-29 |
| position | 11-29 |
| PUTFILE() | 11-29 |
| puts() | 11-31 |
| read() | 11-32 |
| readln() | 11-33 |
| RENAME | 11-33 |
| rename() | 11-34 |
| RUN | 11-35 |
| RUN() | 11-35 |
| seek() | 11-36 |
| SET DIRECTORY | 11-37 |
| SET FULLPATH | 11-37 |
| SET PATH | 11-37 |
| shortName() | 11-38 |
| size() | 11-38 |
| time() | 11-39 |
| TYPE | 11-40 |
| VALIDDRIVE() | 11-41 |
| write() | 11-41 |
| writeln() | 11-42 |
| _dbwinhome | 11-43 |

Chapter 12

Xbase

12-1

| | | | |
|---------------------------------------|-------|------------------------------|-------|
| Common command elements | 12-1 | FDECIMAL() | 12-44 |
| Filenames | 12-1 | FIELD() | 12-45 |
| Aliases | 12-2 | FLDCOUNT() | 12-46 |
| Command scope | 12-3 | FLDLIST() | 12-46 |
| ALIAS() | 12-3 | FLENGTH() | 12-47 |
| APPEND | 12-4 | FLOCK() | 12-48 |
| APPEND AUTOMEM. | 12-5 | FLUSH | 12-49 |
| APPEND FROM | 12-7 | FOR() | 12-49 |
| APPEND FROM ARRAY | 12-8 | FOUND() | 12-50 |
| APPEND MEMO. | 12-10 | GENERATE | 12-51 |
| AVERAGE | 12-11 | GO | 12-51 |
| BEGINTRANS() | 12-12 | INDEX | 12-52 |
| BINTYPE() | 12-13 | ISBLANK() | 12-57 |
| BLANK | 12-14 | ISTABLE() | 12-58 |
| BOF() | 12-15 | KEY() | 12-58 |
| BOOKMARK() | 12-16 | KEYMATCH() | 12-59 |
| BROWSE | 12-16 | LIST. | 12-60 |
| CALCULATE | 12-18 | LKSYS() | 12-61 |
| CHANGE() | 12-20 | LOCATE | 12-62 |
| CLEAR AUTOMEM. | 12-21 | LOCK() | 12-64 |
| CLEAR FIELDS | 12-21 | LOOKUP() | 12-64 |
| CLOSE DATABASES | 12-22 | LUPDATE() | 12-65 |
| CLOSE INDEXES | 12-22 | MDX() | 12-65 |
| CLOSE TABLES | 12-22 | MEMLINES() | 12-67 |
| COMMIT() | 12-23 | MLINE() | 12-68 |
| CONTINUE | 12-23 | NDX() | 12-69 |
| COPY | 12-24 | OPEN DATABASE | 12-69 |
| COPY BINARY. | 12-26 | ORDER() | 12-70 |
| COPY MEMO | 12-26 | PACK. | 12-71 |
| COPY STRUCTURE | 12-27 | RECALL | 12-72 |
| COPY STRUCTURE EXTENDED | 12-28 | RECCOUNT() | 12-73 |
| COPY TABLE. | 12-29 | RECNO() | 12-73 |
| COPY TO ARRAY | 12-30 | RECSIZE() | 12-74 |
| COUNT | 12-32 | REFRESH | 12-74 |
| CREATE SESSION. | 12-33 | REINDEX | 12-75 |
| CREATE...FROM. | 12-35 | RELATION() | 12-75 |
| CREATE...STRUCTURE EXTENDED | 12-36 | RELEASE AUTOMEM | 12-76 |
| DATABASE() | 12-37 | RENAME TABLE | 12-76 |
| DBF() | 12-38 | REPLACE | 12-77 |
| DELETE | 12-38 | REPLACE AUTOMEM. | 12-79 |
| DELETE TABLE | 12-39 | REPLACE BINARY. | 12-80 |
| DELETE TAG. | 12-40 | REPLACE FROM ARRAY | 12-81 |
| DELETED() | 12-40 | REPLACE MEMO | 12-83 |
| DESCENDING(). | 12-41 | REPLACE OLE | 12-84 |
| DISPLAY | 12-42 | RLOCK() | 12-84 |
| EDIT. | 12-43 | ROLLBACK() | 12-86 |
| EOF() | 12-43 | SCAN. | 12-86 |
| | | SEEK | 12-88 |
| | | SEEK() | 12-90 |

| | |
|-------------------------|--------|
| SELECT | 12-90 |
| SELECT() | 12-91 |
| SET AUTOSAVE | 12-92 |
| SET DATABASE | 12-92 |
| SET DBTYPE | 12-93 |
| SET DELETED | 12-93 |
| SET EXACT | 12-94 |
| SET EXCLUSIVE | 12-94 |
| SET FIELDS | 12-95 |
| SET FILTER | 12-98 |
| SET HEADINGS | 12-99 |
| SET INDEX | 12-99 |
| SET KEY TO | 12-101 |
| SET LOCK | 12-102 |
| SET MEMOWIDTH | 12-103 |
| SET NEAR | 12-103 |
| SET ODOMETER | 12-104 |
| SET ORDER | 12-104 |
| SET REFRESH | 12-105 |
| SET RELATION | 12-105 |
| SET REPROCESS | 12-108 |
| SET SAFETY | 12-108 |
| SET SKIP | 12-109 |
| SET UNIQUE | 12-110 |
| SET VIEW | 12-110 |
| SKIP | 12-111 |
| SORT | 12-111 |
| STORE AUTOMEM | 12-113 |
| SUM | 12-114 |
| TAG() | 12-115 |
| TAGCOUNT() | 12-116 |
| TAGNO() | 12-116 |
| TARGET() | 12-117 |
| TOTAL | 12-117 |
| UNIQUE() | 12-119 |
| UNLOCK | 12-119 |
| UPDATE | 12-120 |
| USE | 12-121 |
| WORKAREA() | 12-124 |
| ZAP | 12-124 |

Chapter 13

Local SQL 13-1

| | |
|------------------------------|------|
| Naming conventions | 13-2 |
| Tables | 13-2 |
| Columns | 13-2 |
| Operators | 13-3 |
| Reserved words | 13-3 |
| Data definition | 13-4 |

| | |
|---|-------|
| Data manipulation | 13-5 |
| Parameter substitutions in DML statements | 13-5 |
| Aggregate functions | 13-5 |
| String functions | 13-6 |
| Date function | 13-7 |
| Updatable queries | 13-7 |
| Restrictions on live queries | 13-7 |
| Restrictions on live joins | 13-7 |
| Constraints | 13-8 |
| Statements supported | 13-8 |
| ALTER TABLE | 13-8 |
| CREATE INDEX | 13-9 |
| CREATE TABLE | 13-9 |
| DELETE | 13-11 |
| DROP INDEX | 13-12 |
| DROP TABLE | 13-12 |
| INSERT | 13-13 |
| SELECT | 13-13 |
| UPDATE | 13-17 |

Chapter 14

Data access objects 14-1

| | |
|--|-------|
| Understanding the data access object hierarchy | 14-1 |
| Accessing tables | 14-2 |
| Putting the data access objects together . | 14-3 |
| Using stored procedures | 14-3 |
| class Database | 14-3 |
| class DataModule | 14-6 |
| class DataModRef | 14-9 |
| class DbError | 14-10 |
| class DbException | 14-11 |
| class DbfField | 14-12 |
| class Field | 14-13 |
| class LockField | 14-15 |
| class Parameter | 14-15 |
| class PdxField | 14-17 |
| class Query | 14-18 |
| class Rowset | 14-20 |
| class Session | 14-25 |
| class SqlField | 14-27 |
| class StoredProc | 14-27 |
| class UpdateSet | 14-30 |
| <i>abandon()</i> | 14-31 |
| <i>abandonUpdates()</i> | 14-33 |
| <i>access()</i> | 14-33 |
| <i>active</i> | 14-34 |
| <i>addPassword()</i> | 14-35 |
| <i>append()</i> | 14-36 |
| <i>appendUpdate()</i> | 14-37 |

| | | | |
|---------------------------------------|-------|--|--------|
| <i>applyFilter()</i> | 14-37 | <i>fieldName</i> | 14-69 |
| <i>applyLocate()</i> | 14-38 | <i>fields</i> | 14-69 |
| <i>applyUpdates()</i> | 14-39 | <i>filename</i> | 14-69 |
| <i>atFirst()</i> | 14-40 | <i>filter</i> | 14-70 |
| <i>atLast()</i> | 14-41 | <i>filterOptions</i> | 14-71 |
| <i>autoEdit</i> | 14-42 | <i>findKey()</i> | 14-71 |
| <i>beforeGetValue</i> | 14-42 | <i>findKeyNearest()</i> | 14-72 |
| <i>beginAppend()</i> | 14-43 | <i>first()</i> | 14-73 |
| <i>beginEdit()</i> | 14-45 | <i>flush()</i> | 14-73 |
| <i>beginFilter()</i> | 14-46 | <i>getSchema()</i> | 14-74 |
| <i>beginLocate()</i> | 14-46 | <i>goto()</i> | 14-75 |
| <i>beginTrans()</i> | 14-47 | <i>handle</i> | 14-76 |
| <i>bookmark()</i> | 14-49 | <i>indexName [Rowset]</i> | 14-76 |
| <i>bookmarksEqual()</i> | 14-50 | <i>indexName [UpdateSet]</i> | 14-77 |
| <i>cacheUpdates</i> | 14-50 | <i>isolationLevel</i> | 14-77 |
| <i>canAbandon</i> | 14-51 | <i>keyViolationTableName</i> | 14-78 |
| <i>canAppend</i> | 14-52 | <i>last()</i> | 14-78 |
| <i>canChange</i> | 14-53 | <i>length</i> | 14-78 |
| <i>canClose</i> | 14-54 | <i>live</i> | 14-79 |
| <i>canDelete</i> | 14-54 | <i>locateNext()</i> | 14-79 |
| <i>canEdit</i> | 14-55 | <i>locateOptions</i> | 14-80 |
| <i>canGetRow</i> | 14-56 | <i>lock</i> | 14-80 |
| <i>canNavigate</i> | 14-56 | <i>lockRetryCount</i> | 14-81 |
| <i>canOpen</i> | 14-57 | <i>lockRetryInterval</i> | 14-82 |
| <i>canSave</i> | 14-57 | <i>lockRow()</i> | 14-82 |
| <i>changedTableName</i> | 14-58 | <i>lockSet()</i> | 14-84 |
| <i>clearFilter()</i> | 14-59 | <i>login()</i> | 14-85 |
| <i>clearRange()</i> | 14-59 | <i>loginString</i> | 14-86 |
| <i>close()</i> | 14-59 | <i>lookupRowset</i> | 14-86 |
| <i>commit()</i> | 14-60 | <i>lookupSQL</i> | 14-87 |
| <i>constrained</i> | 14-60 | <i>lookupTable</i> | 14-88 |
| <i>copy()</i> | 14-61 | <i>lookupType</i> | 14-88 |
| <i>copyTable()</i> | 14-61 | <i>masterFields</i> | 14-89 |
| <i>copyToFile()</i> | 14-62 | <i>masterRowset</i> | 14-91 |
| <i>count()</i> | 14-62 | <i>masterSource</i> | 14-91 |
| <i>database</i> | 14-63 | <i>maximum</i> | 14-92 |
| <i>databaseName</i> | 14-63 | <i>minimum</i> | 14-92 |
| <i>dataModClass</i> | 14-64 | <i>modified</i> | 14-93 |
| <i>decimalLength</i> | 14-64 | <i>next()</i> | 14-94 |
| <i>default</i> | 14-64 | <i>notifyControls</i> | 14-95 |
| <i>delete()</i> [Rowset] | 14-65 | <i>onAbandon</i> | 14-95 |
| <i>delete()</i> [UpdateSet] | 14-65 | <i>onAppend</i> | 14-96 |
| <i>destination</i> | 14-66 | <i>onChange</i> | 14-97 |
| <i>driverName</i> | 14-66 | <i>onClose</i> | 14-97 |
| <i>dropTable()</i> | 14-66 | <i>onDelete</i> | 14-98 |
| <i>emptyTable()</i> | 14-67 | <i>onEdit</i> | 14-98 |
| <i>endOfSet</i> | 14-67 | <i>onGotValue</i> | 14-99 |
| <i>execute()</i> | 14-68 | <i>onNavigate</i> | 14-99 |
| <i>executeSQL()</i> | 14-68 | <i>onOpen</i> | 14-100 |

| | |
|------------------------------------|--------|
| <i>onProgress</i> | 14-101 |
| <i>onSave</i> | 14-102 |
| <i>open()</i> | 14-102 |
| <i>packTable()</i> | 14-103 |
| <i>params</i> | 14-103 |
| <i>picture</i> | 14-104 |
| <i>precision</i> | 14-104 |
| <i>prepare()</i> | 14-105 |
| <i>problemTableName</i> | 14-106 |
| <i>procedureName</i> | 14-106 |
| <i>readOnly</i> | 14-107 |
| <i>ref</i> | 14-108 |
| <i>refresh()</i> | 14-108 |
| <i>refreshControls()</i> | 14-109 |
| <i>refreshRow()</i> | 14-109 |
| <i>reindex()</i> | 14-109 |
| <i>renameTable()</i> | 14-110 |
| <i>replaceFromFile()</i> | 14-110 |
| <i>requery()</i> | 14-111 |
| <i>requestLive</i> | 14-112 |
| <i>required</i> | 14-112 |
| <i>rollback()</i> | 14-113 |
| <i>rowCount()</i> | 14-113 |
| <i>rowNo()</i> | 14-114 |
| <i>rowset</i> | 14-114 |
| <i>save()</i> | 14-115 |
| <i>scale</i> | 14-115 |
| <i>session</i> | 14-116 |
| <i>setRange()</i> | 14-116 |
| <i>share</i> | 14-117 |
| <i>source</i> | 14-118 |
| <i>sql</i> | 14-118 |
| <i>state</i> | 14-119 |
| <i>tableExists()</i> | 14-120 |
| <i>type [Field]</i> | 14-121 |
| <i>type [Parameter]</i> | 14-121 |
| <i>unidirectional</i> | 14-122 |
| <i>unlock()</i> | 14-122 |
| <i>unprepare()</i> | 14-122 |
| <i>update</i> | 14-123 |
| <i>update()</i> | 14-123 |
| <i>updateWhere</i> | 14-124 |
| <i>user</i> | 14-124 |
| <i>user()</i> | 14-125 |
| <i>value [Field]</i> | 14-125 |
| <i>value [Parameter]</i> | 14-126 |

Chapter 15 Form objects

15-1

| | |
|---|-------|
| Common visual component properties. | 15-1 |
| class <i>ActiveX</i> | 15-3 |
| class <i>Browse</i> | 15-4 |
| class <i>CheckBox</i> | 15-6 |
| class <i>ColumnCheckBox</i> | 15-8 |
| class <i>ColumnComboBox</i> | 15-9 |
| class <i>ColumnEntryfield</i> | 15-10 |
| class <i>ColumnHeadingControl</i> | 15-11 |
| class <i>ColumnSpinBox</i> | 15-12 |
| class <i>ComboBox</i> | 15-13 |
| class <i>Container</i> | 15-15 |
| class <i>Editor</i> | 15-16 |
| class <i>Entryfield</i> | 15-18 |
| class <i>Form</i> | 15-20 |
| class <i>Grid</i> | 15-24 |
| class <i>GridColumn</i> | 15-26 |
| class <i>Image</i> | 15-27 |
| class <i>Line</i> | 15-28 |
| class <i>ListBox</i> | 15-30 |
| class <i>NoteBook</i> | 15-31 |
| class <i>OLE</i> | 15-32 |
| class <i>PaintBox</i> | 15-34 |
| class <i>Progress</i> | 15-36 |
| class <i>PushButton</i> | 15-37 |
| class <i>RadioButton</i> | 15-38 |
| class <i>Rectangle</i> | 15-39 |
| class <i>ReportViewer</i> | 15-41 |
| class <i>ScrollBar</i> | 15-42 |
| class <i>Shape</i> | 15-43 |
| class <i>Slider</i> | 15-44 |
| class <i>SpinBox</i> | 15-46 |
| class <i>TabBox</i> | 15-48 |
| class <i>Text</i> | 15-49 |
| class <i>Treeltem</i> | 15-51 |
| class <i>TreeView</i> | 15-53 |
| <i>abandonRecord()</i> | 15-55 |
| <i>activeControl</i> | 15-56 |
| <i>alias</i> | 15-56 |
| <i>alignHorizontal</i> | 15-57 |
| <i>alignment [Image]</i> | 15-57 |
| <i>alignment [Text]</i> | 15-58 |
| <i>alignVertical</i> | 15-58 |
| <i>allowAddRows</i> | 15-59 |
| <i>allowColumnMoving</i> | 15-59 |
| <i>allowColumnSizing</i> | 15-59 |
| <i>allowEditing</i> | 15-60 |
| <i>allowEditLabels</i> | 15-60 |
| <i>allowEditTree</i> | 15-60 |
| <i>allowRowSizing</i> | 15-61 |

| | | | |
|---------------------------------------|-------|-------------------------------------|--------|
| <i>anchor</i> | 15-61 | <i>dropDownWidth</i> | 15-92 |
| <i>append</i> | 15-62 | <i>editorControl</i> | 15-92 |
| <i>autoCenter</i> | 15-62 | <i>editorType</i> | 15-92 |
| <i>autoDrop</i> | 15-62 | <i>elements</i> | 15-93 |
| <i>autoSize</i> | 15-63 | <i>enabled</i> | 15-94 |
| <i>background</i> | 15-63 | <i>enableSelection</i> | 15-94 |
| <i>before</i> | 15-63 | <i>endSelection</i> | 15-94 |
| <i>beginAppend()</i> | 15-64 | <i>ensureVisible()</i> | 15-95 |
| <i>bgColor</i> | 15-65 | <i>escExit</i> | 15-95 |
| <i>bold</i> | 15-65 | <i>evalTags</i> | 15-95 |
| <i>border</i> | 15-66 | <i>expanded</i> | 15-96 |
| <i>borderStyle</i> | 15-66 | <i>fields</i> | 15-96 |
| <i>bottom</i> | 15-67 | <i>filename</i> | 15-97 |
| <i>buttons</i> | 15-67 | <i>first</i> | 15-98 |
| <i>canChange</i> | 15-67 | <i>firstChild</i> | 15-98 |
| <i>canClose</i> | 15-68 | <i>firstVisibleChild</i> | 15-99 |
| <i>canEditLabel</i> | 15-69 | <i>focus</i> | 15-99 |
| <i>canExpand</i> | 15-69 | <i>focusBitmap</i> | 15-99 |
| <i>canNavigate</i> | 15-71 | <i>fontBold</i> | 15-100 |
| <i>cellHeight</i> | 15-72 | <i>fontItalic</i> | 15-100 |
| <i>checkboxes</i> | 15-72 | <i>fontName</i> | 15-100 |
| <i>checked</i> | 15-72 | <i>fontSize</i> | 15-101 |
| <i>checkedImage</i> | 15-73 | <i>fontStrikeout</i> | 15-101 |
| <i>classId</i> | 15-73 | <i>fontUnderline</i> | 15-101 |
| <i>clearTics()</i> | 15-73 | <i>form</i> | 15-101 |
| <i>clientEdge</i> | 15-74 | <i>frozenColumn</i> | 15-102 |
| <i>close()</i> | 15-74 | <i>function</i> | 15-103 |
| <i>colorHighlight</i> | 15-75 | <i>getTextExtent()</i> | 15-103 |
| <i>colorNormal</i> | 15-75 | <i>gridLineWidth</i> | 15-104 |
| <i>columnCount</i> | 15-81 | <i>group</i> | 15-104 |
| <i>columns</i> | 15-81 | <i>handle</i> | 15-105 |
| <i>copy()</i> | 15-81 | <i>hasButtons</i> | 15-105 |
| <i>count()</i> | 15-82 | <i>hasColumnHeadings</i> | 15-105 |
| <i>cuaTab</i> | 15-82 | <i>hasColumnLines</i> | 15-106 |
| <i>currentColumn</i> | 15-83 | <i>hasIndicator</i> | 15-106 |
| <i>curSel</i> | 15-83 | <i>hasLines</i> | 15-106 |
| <i>cut()</i> | 15-84 | <i>hasRowLines</i> | 15-106 |
| <i>dataLink</i> | 15-84 | <i>hasVScrollHintText</i> | 15-107 |
| <i>dataSource [options]</i> | 15-85 | <i>headingControl</i> | 15-107 |
| <i>dataSource [Image]</i> | 15-87 | <i>height</i> | 15-107 |
| <i>default</i> | 15-88 | <i>helpFile</i> | 15-108 |
| <i>description</i> | 15-89 | <i>helpId</i> | 15-108 |
| <i>designView</i> | 15-89 | <i>hScrollBar</i> | 15-109 |
| <i>disabledBitmap</i> | 15-89 | <i>hWnd</i> | 15-109 |
| <i>disablePopup</i> | 15-90 | <i>hWndClient</i> | 15-110 |
| <i>doVerb()</i> | 15-90 | <i>icon</i> | 15-110 |
| <i>downBitmap</i> | 15-91 | <i>id</i> | 15-110 |
| <i>dragScrollRate</i> | 15-91 | <i>image</i> | 15-111 |
| <i>dropDownHeight</i> | 15-92 | <i>imageScaleToFont</i> | 15-111 |

| | | | |
|------------------------------------|--------|---------------------------------------|--------|
| <i>imageSize</i> | 15-111 | <i>onLostFocus</i> | 15-135 |
| <i>indent</i> | 15-112 | <i>onMiddleDblClick</i> | 15-136 |
| <i>inDesign</i> | 15-112 | <i>onMiddleMouseDown</i> | 15-136 |
| <i>integralHeight</i> | 15-112 | <i>onMiddleMouseUp</i> | 15-136 |
| <i>isRecordChanged()</i> | 15-113 | <i>onMouseMove</i> | 15-137 |
| <i>key</i> | 15-113 | <i>onMove</i> | 15-138 |
| <i>keyboard()</i> | 15-114 | <i>onNavigate</i> | 15-138 |
| <i>left</i> | 15-115 | <i>onOpen</i> | 15-139 |
| <i>level</i> | 15-115 | <i>onPaint</i> | 15-140 |
| <i>lineNo</i> | 15-115 | <i>onRightDblClick</i> | 15-140 |
| <i>linesAtRoot</i> | 15-116 | <i>onRightMouseDown</i> | 15-140 |
| <i>linkFileName</i> | 15-116 | <i>onRightMouseUp</i> | 15-141 |
| <i>lockedColumns</i> | 15-116 | <i>onSelChange</i> | 15-141 |
| <i>maximize</i> | 15-116 | <i>onSelection</i> | 15-141 |
| <i>maxLength</i> | 15-117 | <i>onSize</i> | 15-142 |
| <i>mdi</i> | 15-117 | <i>open()</i> | 15-143 |
| <i>memoEditor</i> | 15-118 | <i>pageCount()</i> | 15-143 |
| <i>menuFile</i> | 15-119 | <i>pageno</i> | 15-144 |
| <i>metric</i> | 15-119 | <i>params</i> | 15-145 |
| <i>minimize</i> | 15-120 | <i>paste()</i> | 15-146 |
| <i>modify</i> | 15-120 | <i>patternStyle</i> | 15-146 |
| <i>mousePointer</i> | 15-121 | <i>pen</i> | 15-147 |
| <i>move()</i> | 15-121 | <i>penStyle</i> | 15-147 |
| <i>moveable</i> | 15-122 | <i>penWidth</i> | 15-148 |
| <i>multiple</i> | 15-122 | <i>phoneticLink</i> | 15-148 |
| <i>multiSelect</i> | 15-123 | <i>picture</i> | 15-148 |
| <i>name</i> | 15-123 | <i>popupEnable</i> | 15-150 |
| <i>nativeObject</i> | 15-124 | <i>popupMenu</i> | 15-150 |
| <i>nextObj</i> | 15-124 | <i>prefixEnable</i> | 15-151 |
| <i>nextSibling</i> | 15-125 | <i>prevSibling</i> | 15-151 |
| <i>noOfChildren</i> | 15-125 | <i>print()</i> | 15-151 |
| <i>oleType</i> | 15-126 | <i>printable</i> | 15-152 |
| <i>onAppend</i> | 15-126 | <i>rangeMax</i> | 15-152 |
| <i>onChange</i> | 15-127 | <i>rangeMin</i> | 15-152 |
| <i>onChar</i> | 15-127 | <i>rangeRequired</i> | 15-153 |
| <i>onCheckBoxClick</i> | 15-128 | <i>readModal()</i> | 15-153 |
| <i>onClick</i> | 15-128 | <i>reExecute()</i> | 15-155 |
| <i>onClose</i> | 15-129 | <i>ref</i> | 15-155 |
| <i>onDesignOpen</i> | 15-130 | <i>refresh()</i> | 15-155 |
| <i>onEditLabel</i> | 15-130 | <i>refreshAlways</i> | 15-156 |
| <i>onExpand</i> | 15-131 | <i>releaseAllChildren()</i> | 15-156 |
| <i>onFormSize</i> | 15-131 | <i>right</i> | 15-157 |
| <i>onGotFocus</i> | 15-131 | <i>rowSelect</i> | 15-157 |
| <i>onHelp</i> | 15-132 | <i>rowset</i> | 15-157 |
| <i>onKeyDown</i> | 15-132 | <i>saveRecord()</i> | 15-158 |
| <i>onKeyUp</i> | 15-133 | <i>scaleFontBold</i> | 15-158 |
| <i>onLeftDblClick</i> | 15-134 | <i>scaleFontName</i> | 15-159 |
| <i>onLeftMouseDown</i> | 15-135 | <i>scaleFontSize</i> | 15-159 |
| <i>onLeftMouseUp</i> | 15-135 | <i>scrollBar</i> | 15-159 |

| | |
|--------------------------------------|--------|
| <i>select()</i> | 15-160 |
| <i>selectAll</i> | 15-160 |
| <i>selected</i> | 15-161 |
| <i>selected()</i> | 15-161 |
| <i>selectedImage</i> | 15-162 |
| <i>serverName</i> | 15-162 |
| <i>setAsFirstVisible()</i> | 15-163 |
| <i>setFocus()</i> | 15-163 |
| <i>setTic()</i> | 15-163 |
| <i>setTicFrequency()</i> | 15-164 |
| <i>shapeStyle</i> | 15-164 |
| <i>showFormatBar()</i> | 15-165 |
| <i>showMemoEditor()</i> | 15-165 |
| <i>showSelAlways</i> | 15-165 |
| <i>showSpeedTip</i> | 15-166 |
| <i>sizeable</i> | 15-166 |
| <i>smallTitle</i> | 15-166 |
| <i>sortChildren()</i> | 15-167 |
| <i>sorted</i> | 15-167 |
| <i>speedBar</i> | 15-167 |
| <i>speedTip</i> | 15-168 |
| <i>spinOnly</i> | 15-168 |
| <i>startSelection</i> | 15-168 |
| <i>statusMessage</i> | 15-169 |
| <i>step</i> | 15-169 |
| <i>style</i> | 15-169 |
| <i>sysMenu</i> | 15-170 |
| <i>tabStop</i> | 15-170 |
| <i>text</i> | 15-171 |
| <i>textLeft</i> | 15-172 |
| <i>tics</i> | 15-172 |
| <i>ticsPos</i> | 15-172 |
| <i>toggle</i> | 15-173 |
| <i>toolTips</i> | 15-173 |
| <i>top</i> | 15-174 |
| <i>topMost</i> | 15-174 |
| <i>trackSelect</i> | 15-174 |
| <i>transparent</i> | 15-175 |
| <i>uncheckedImage</i> | 15-175 |
| <i>undo()</i> | 15-175 |
| <i>upBitmap</i> | 15-176 |
| <i>useTablePopup</i> | 15-176 |
| <i>valid</i> | 15-177 |
| <i>validErrorMsg</i> | 15-178 |
| <i>validRequired</i> | 15-178 |
| <i>value</i> | 15-179 |
| <i>vertical</i> | 15-180 |
| <i>view</i> | 15-180 |
| <i>visible</i> | 15-181 |

| | |
|---------------------------------|--------|
| <i>visibleCount()</i> | 15-181 |
| <i>visualStyle</i> | 15-181 |
| <i>vScrollBar</i> | 15-182 |
| <i>when</i> | 15-182 |
| <i>width</i> | 15-183 |
| <i>windowState</i> | 15-184 |
| <i>wrap</i> | 15-184 |

Chapter 16

Application shell

16-1

| | |
|-----------------------------------|-------|
| <i>_app</i> | 16-1 |
| <i>_app.frameWin</i> | 16-2 |
| <i>class Menu</i> | 16-3 |
| <i>class MenuBar</i> | 16-6 |
| <i>class Popup</i> | 16-8 |
| <i>class ToolBar</i> | 16-9 |
| <i>class ToolButton</i> | 16-11 |
| <i>attach()</i> | 16-13 |
| <i>checked</i> | 16-13 |
| <i>checkedBitmap</i> | 16-14 |
| <i>CLEAR TYPEAHEAD</i> | 16-15 |
| <i>DEFINE COLOR</i> | 16-15 |
| <i>detach()</i> | 16-16 |
| <i>editCopyMenu</i> | 16-17 |
| <i>editCutMenu</i> | 16-18 |
| <i>editPasteMenu</i> | 16-18 |
| <i>editUndoMenu</i> | 16-19 |
| <i>GETCOLOR()</i> | 16-19 |
| <i>GETFONT()</i> | 16-20 |
| <i>INKEY()</i> | 16-21 |
| <i>KEYBOARD</i> | 16-24 |
| <i>MSGBOX()</i> | 16-26 |
| <i>NEXTKEY()</i> | 16-28 |
| <i>ON ESCAPE</i> | 16-29 |
| <i>ON KEY</i> | 16-31 |
| <i>onInitiate</i> | 16-33 |
| <i>onInitMenu</i> | 16-34 |
| <i>onUpdate</i> | 16-34 |
| <i>separator</i> | 16-35 |
| <i>SET CONFIRM</i> | 16-35 |
| <i>SET CUAENTER</i> | 16-36 |
| <i>SET ESCAPE</i> | 16-37 |
| <i>SET FUNCTION</i> | 16-38 |
| <i>SET MESSAGE</i> | 16-40 |
| <i>SET TYPEAHEAD</i> | 16-40 |
| <i>SHELL()</i> | 16-41 |
| <i>shortCut</i> | 16-43 |
| <i>SLEEP</i> | 16-43 |
| <i>trackRight</i> | 16-45 |

| | |
|----------------------------------|-------|
| <i>uncheckedBitmap</i> | 16-45 |
| <i>WAIT</i> | 16-46 |
| <i>windowMenu</i> | 16-47 |

Chapter 17

Report objects 17-1

| | |
|--|-------|
| A simple report example | 17-2 |
| How a report is rendered | 17-4 |
| class <i>Band</i> | 17-5 |
| class <i>Group</i> | 17-6 |
| class <i>PageTemplate</i> | 17-8 |
| class <i>Report</i> | 17-9 |
| class <i>StreamFrame</i> | 17-10 |
| class <i>StreamSource</i> | 17-12 |
| <i>agAverage()</i> | 17-13 |
| <i>agCount()</i> | 17-14 |
| <i>agMax()</i> | 17-15 |
| <i>agMin()</i> | 17-16 |
| <i>agStandardDeviation()</i> | 17-16 |
| <i>agSum()</i> | 17-17 |
| <i>agVariance()</i> | 17-18 |
| <i>autoSort</i> | 17-19 |
| <i>beginNewFrame</i> | 17-20 |
| <i>beginNewFrame()</i> | 17-20 |
| <i>context</i> | 17-20 |
| <i>canRender</i> | 17-21 |
| <i>detailBand</i> | 17-22 |
| <i>drillDown</i> | 17-22 |
| <i>endPage</i> | 17-23 |
| <i>expandable</i> | 17-23 |
| <i>firstOnFrame</i> | 17-23 |
| <i>firstPageTemplate</i> | 17-24 |
| <i>fixed</i> | 17-24 |
| <i>footerBand</i> | 17-25 |
| <i>groupBy</i> | 17-25 |
| <i>headerBand</i> | 17-27 |
| <i>headerEveryFrame</i> | 17-27 |
| <i>isLastPage()</i> | 17-28 |
| <i>leading</i> | 17-29 |
| <i>marginBottom</i> | 17-29 |
| <i>marginHorizontal</i> | 17-30 |
| <i>marginLeft</i> | 17-30 |
| <i>marginRight</i> | 17-30 |
| <i>marginTop</i> | 17-31 |
| <i>marginVertical</i> | 17-31 |
| <i>maxRows</i> | 17-32 |
| <i>nextPageTemplate</i> | 17-32 |
| <i>onPage</i> | 17-32 |
| <i>onRender</i> | 17-33 |

| | |
|--|-------|
| <i>output</i> | 17-34 |
| <i>outputFilename</i> | 17-35 |
| <i>preRender</i> | 17-35 |
| <i>printer</i> | 17-36 |
| <i>render()</i> | 17-37 |
| <i>renderOffset</i> | 17-37 |
| <i>reportGroup</i> | 17-38 |
| <i>reportPage</i> | 17-38 |
| <i>reportViewer</i> | 17-39 |
| <i>rotate</i> | 17-39 |
| <i>startPage</i> | 17-39 |
| <i>streamFrame</i> | 17-40 |
| <i>streamSource</i> | 17-40 |
| <i>suppressIfBlank</i> | 17-41 |
| <i>suppressIfDuplicate</i> | 17-41 |
| <i>title</i> | 17-42 |
| <i>tracking</i> | 17-42 |
| <i>trackJustifyThreshold</i> | 17-42 |
| <i>variableHeight</i> | 17-42 |
| <i>verticalJustifyLimit</i> | 17-43 |

Chapter 18

Text streaming 18-1

| | |
|---|-------|
| <i>?</i> | 18-1 |
| <i>??</i> | 18-5 |
| <i>CHOOSEPRINTER()</i> | 18-6 |
| <i>CLEAR</i> | 18-6 |
| <i>CLOSE ALTERNATE</i> | 18-6 |
| <i>CLOSE PRINTER</i> | 18-7 |
| <i>EJECT</i> | 18-7 |
| <i>EJECT PAGE</i> | 18-8 |
| <i>ON PAGE</i> | 18-8 |
| <i>PCOL()</i> | 18-10 |
| <i>PRINTJOB...ENDPRINTJOB</i> | 18-11 |
| <i>PRINTSTATUS()</i> | 18-12 |
| <i>PROW()</i> | 18-13 |
| <i>SET ALTERNATE</i> | 18-13 |
| <i>SET CONSOLE</i> | 18-15 |
| <i>SET MARGIN</i> | 18-16 |
| <i>SET PCOL</i> | 18-17 |
| <i>SET PRINTER</i> | 18-18 |
| <i>SET PROW</i> | 18-19 |
| <i>SET SPACE</i> | 18-20 |
| <i>_alignment</i> | 18-20 |
| <i>_indent</i> | 18-21 |
| <i>_lmargin</i> | 18-22 |
| <i>_padvance</i> | 18-23 |
| <i>_pageno</i> | 18-24 |
| <i>_pbpage</i> | 18-25 |

| | |
|-------------------------|-------|
| _pcolno | 18-26 |
| _pcopies | 18-27 |
| _pdriver | 18-27 |
| _pject | 18-28 |
| _page | 18-29 |
| _pform | 18-30 |
| _plength | 18-31 |
| _plineno | 18-32 |
| _ploffset | 18-33 |
| _porientation | 18-34 |
| _ppitch | 18-35 |
| _pquality | 18-36 |
| _pspacing | 18-36 |
| _rmargin | 18-37 |
| _tabs | 18-38 |
| _wrap | 18-39 |

Chapter 19

Extending *Visual* dBASE

with DLLs, OLE and DDE **19-1**

| | |
|-------------------------------|-------|
| class DDELink | 19-1 |
| class DDETopic | 19-6 |
| class OleAutoClient | 19-9 |
| <i>advise()</i> | 19-11 |
| <i>execute()</i> | 19-11 |
| EXTERN | 19-12 |
| <i>initiate()</i> | 19-16 |
| LOAD DLL | 19-16 |
| <i>notify()</i> | 19-17 |
| <i>onAdvise</i> | 19-18 |
| <i>onExecute</i> | 19-19 |
| <i>onNewValue</i> | 19-19 |
| <i>onPeek</i> | 19-20 |
| <i>onPoke</i> | 19-20 |
| <i>onUnadvise</i> | 19-21 |
| <i>peek()</i> | 19-21 |
| PLAY SOUND | 19-22 |
| <i>poke()</i> | 19-23 |
| <i>reconnect()</i> | 19-23 |
| RELEASE DLL | 19-24 |
| RESOURCE() | 19-25 |
| RESTORE IMAGE | 19-25 |
| <i>server</i> | 19-26 |
| <i>terminate()</i> | 19-27 |
| <i>timeout</i> | 19-27 |
| <i>topic</i> | 19-28 |
| <i>unadvise()</i> | 19-28 |

Chapter 20

IDE **20-1**

| | |
|-----------------------------|-------|
| BUILD | 20-1 |
| CLEAR ALL | 20-2 |
| CLOSE ALL | 20-2 |
| COMPILE | 20-3 |
| CONVERT | 20-4 |
| CREATE | 20-5 |
| CREATE COMMAND | 20-6 |
| CREATE DATAMODULE | 20-7 |
| CREATE FILE | 20-7 |
| CREATE FORM | 20-8 |
| CREATE LABEL | 20-9 |
| CREATE MENU | 20-10 |
| CREATE POPUP | 20-10 |
| CREATE PROJECT | 20-11 |
| CREATE QUERY | 20-11 |
| CREATE REPORT | 20-12 |
| DEBUG | 20-13 |
| DISPLAY COVERAGE | 20-13 |
| DISPLAY MEMORY | 20-15 |
| DISPLAY STATUS | 20-16 |
| DISPLAY STRUCTURE | 20-17 |
| HELP | 20-19 |
| INSPECT() | 20-19 |
| LIST.... | 20-20 |
| MODIFY.... | 20-20 |
| MODIFY PROJECT | 20-21 |
| MODIFY STRUCTURE | 20-21 |
| SET | 20-23 |
| SET BELL | 20-23 |
| SET BLOCKSIZE | 20-24 |
| SET COVERAGE | 20-25 |
| SET DESIGN | 20-26 |
| SET DEVELOPMENT | 20-27 |
| SET EDITOR | 20-28 |
| SET IBLOCK | 20-29 |
| SET MBLOCK | 20-31 |
| SET TALK | 20-32 |

Chapter 21

Everything Else (Except

Preprocessor) **21-1**

| | |
|--------------------|------|
| ACCESS() | 21-1 |
| ANSI() | 21-1 |
| CANCEL | 21-3 |
| CERROR(). | 21-3 |
| CHARSET(). | 21-4 |

| | | | |
|----------------|-------|---|-------|
| DBERROR() | 21-4 | #else | 22-6 |
| DBMESSAGE() | 21-5 | #endif | 22-6 |
| ERROR() | 21-6 | #if | 22-6 |
| ID() | 21-6 | #ifdef | 22-8 |
| LDRIVER() | 21-7 | #ifndef | 22-9 |
| LINENO() | 21-8 | #include | 22-10 |
| LOGOUT | 21-8 | #pragma | 22-11 |
| MEMORY() | 21-9 | #undef | 22-12 |
| MESSAGE() | 21-10 | __vdb__ | 22-13 |
| NETWORK() | 21-10 | | |
| OEM() | 21-11 | | |
| ON ERROR | 21-12 | Appendix A | |
| ON NETERROR | 21-13 | ASCII character chart | |
| PROGRAM() | 21-14 | (code page 437) | A-1 |
| PROTECT | 21-15 | | |
| RESUME | 21-16 | Appendix B | |
| RETRY | 21-16 | File Structures | B-1 |
| SET ENCRYPTION | 21-17 | Table header and records | B-1 |
| SET ERROR | 21-18 | Table header structure | B-1 |
| SET LDCHECK | 21-18 | Table records | B-2 |
| SET LD_CONVERT | 21-19 | Binary, memo, and OLE fields and .DBT files | B-3 |
| SQLERROR() | 21-19 | | |
| SQLEXEC() | 21-20 | Appendix C | |
| SQLMESSAGE() | 21-22 | Error codes | C-1 |
| SUSPEND | 21-22 | | |
| USER() | 21-23 | | |
| VERSION() | 21-24 | | |
| Chapter 22 | | Index | I-1 |
| Preprocessor | 22-1 | | |
| #define | 22-2 | | |

Introduction

The *Language Reference* describes the classes, objects, properties, events, methods, functions, and preprocessor directives available in the *Visual dBASE™ 7* language.

How this book is organized

- Chapter 2, “Language definition,” covers the basic concepts and components of the *Visual dBASE* language.
- Chapter 3, “Syntax conventions,” describes the conventions used in presenting the syntax of language elements, and provides guidelines for interpreting the syntax notation.
- Chapter 4, “Operators and symbols,” describes the operators and symbols used in the language.
- Chapter 5, “Core language,” is a topical reference to the individual core elements of the language.
- Chapter 6, “String objects,” describes the classes, methods and properties that relate to the use of strings in *Visual dBASE* code.
- Chapter 7, “Math / Money,” lists the classes, methods and properties associated with mathematical operations, including trigonometrical and logarithmic operations.
- Chapter 8, “Date and time objects,” guides you through the elements of date and time objects in *Visual dBASE* code.
- Chapter 9, “Bitwise,” covers the language elements that deal with bit manipulation and base conversion for unsigned 32-bit values.
- Chapter 10, “Array objects,” details *Visual dBASE*’s support for a wide variety of array types.

- Chapter 11, “File/OS,” describes the File class, which provides byte-level access to files, as well as other file and operating system-related functions.
- Chapter 12, “Xbase,” is a reference to legacy dBASE data manipulation and utility commands and functions. Entries include an object-oriented DML equivalent, if one exists.
- Chapter 13, “Local SQL,” summarizes the SQL commands that can be used within *Visual* dBASE when working with BDE Standard dBASE and Paradox tables.
- Chapter 14, “Data access objects,” specifies the various elements that provide access to database tables and are used to link tables to the user interface.
- Chapter 15, “Form objects,” covers the classes, methods, events and properties related to *Visual* dBASE forms.
- Chapter 16, “Application shell,” describes the supporting application elements such as menus, popups, toolbars, standard dialogs, keyboard behavior, and the *_app* object.
- Chapter 17, “Report objects,” helps you understand the elements of *Visual* dBASE’s new reporting capabilities.
- Chapter 18, “Text streaming,” covers the *Visual* dBASE language elements that control text streaming to the Command window, a file, or a printer.
- Chapter 19, “Extending Visual dBASE with DLLs, OLE and DDE,” contains the information you need to extend *Visual* dBASE with OLE Automation, Dynamic Data Exchange, Dynamic Linked Libraries, and other Windows resources and mechanisms.
- Chapter 20, “IDE,” describes language elements that you use within the *Visual* dBASE integrated development environment (IDE) to programmatically create, modify, compile and build applications.
- Chapter 21, “Everything Else (Except Preprocessor),” covers *Visual* dBASE language elements that pertain to errors, security, and locale.
- Chapter 22, “Preprocessor,” describes the separate built-in utility that processes the text of your *Visual* dBASE programs and prepares them for compilation.

Typographical conventions

The *Language Reference* uses specific typographical conventions to help you distinguish among the various language and syntax elements. These conventions are used to make the manual more readable.

| Convention | Applies to | Examples |
|----------------------------------|---|---|
| Italic/Camel cap | Property names, events, methods, arguments | <i>length</i> property, <i>lockRow()</i> method, <i><start expN></i> argument |
| ALL CAPS | Legacy dBASE commands and other language elements from previous versions. Also used in file and directory references. | APPEND BLANK, CUSTOMER.DBF |
| Roman/Initial cap/ Camel camp | Class names (including legacy classes), table names, field names, menu commands | class File, class OleAutoClient, Members table, Price field |
| Monospaced font | Code examples | <code>a = new Array(5, 6)</code> |

In addition to the typographical conventions listed in this table, Chapter 3, “Syntax conventions,” explains the various symbols, conventions and syntactical options used in the language.

Using the online version

The complete *Language Reference* is also available as part of your online Help system. The online version also includes updated entries, expanded examples and other language information that was not available when this printed version went to press.

You can find language elements in the online Help system using any of the standard Help search mechanisms, including the Help contents and index, Help buttons in dialogs, *F1* on windows and controls, and through full text searches.

In addition, you can get instant Help on any property, event or method by selecting it in the Inspector and pressing *F1*. You can also highlight any language element (or any other word or phrase) in the Source editor, Command window, and most other text entry windows, and press *F1*. If the highlighted word or phrase is part of the documentation, Help appears.

For general usage information or an introduction to how the Windows Help system works, choose How to Use Help from the *Visual dBASE* Help menu.

Additional *Visual dBASE* information, technical notes, white papers, resource lists, examples, Help updates and corrections are posted regularly to the dBASE Inc. web site. For details and site addresses, see the README.TXT file on your CD or in your main *Visual dBASE* directory.

Language definition

Visual dBASE is a dynamic object-oriented programming language. It features dozens of built-in classes that represent forms, visual components, reports, and databases in an advanced integrated development environment with Two-Way Tool designers.

This chapter defines the language elements in *Visual* dBASE. After a brief overview of basic language attributes, which is geared toward those with previous programming experience, the language is described from its most fundamental elements, data types, to the most general.

Basic attributes

If you're familiar with another programming language, knowing the following attributes will help orient you to *Visual* dBASE. If *Visual* dBASE is your first programming language, you may not recognize some of the terminology below. Keep the rules in mind; the terminology will be explained later in this chapter.

- *Visual* dBASE is not case-sensitive.

Although language elements are capitalized using certain conventions in the *Language Reference*, you are not required to follow these conventions.

Rules of thumb for how things are capitalized are listed in Chapter 3, "Syntax conventions." You are encouraged to follow these rules when you create your own names for variables and properties.

- *Visual* dBASE is line-oriented.

By default, there is one line per statement, and one statement per line. You may use the semicolon (;) to continue a statement on the next line, or to combine multiple statements on the same line.

- Most structural language elements use keyword pairs.

Most structural language elements start with a specific keyword, and end with a paired keyword. The ending keyword is usually the word starting keyword preceded by the word `END`; for example `IF/ENDIF`, `CLASS/ENDCLASS`, and `TRY/ENDTRY`.

- Literal strings are delimited by single quotes, double quotes, or square brackets.
- *Visual* dBASE is weakly typed with automatic type conversion.

You don't have to declare a variable before you use it. You can change the type of a variable at any time.

- *Visual* dBASE's object model supports dynamic subclassing.

Dynamic subclassing allows you to add new properties on-the-fly, properties that were not declared in the class structure.

Data types

Data is both the means and the end for both programming and databases. Because *Visual* dBASE is designed to manipulate databases, there are three categories of data types:

- Simple data types common to both the language and databases
- Database-specific data types
- Data types used in programming

Simple data types

There are five simple data types common to both *Visual* dBASE and databases:

- String
- Numeric
- Logical or boolean
- Date
- Null

Keep in mind that different table formats support different data types to varying degrees.

For each of these data types, there is a way to designate a value of that type in *Visual* dBASE code. This is known as the *literal* representation.

String data

A string is composed of zero or more characters: letters, digits, spaces, or special symbols. A string with no characters is called an empty string or a null string (not to be confused with the null data type).

The maximum number of characters allowed in a string depends on where that string is stored. In *Visual* dBASE, the maximum is approximately 1 billion characters, if you have enough virtual memory. For DBF (dBASE®) tables, you

may store 254 characters in a character field and an unlimited number in a memo field. For DB (Paradox) tables, the limit is 255 characters in an alpha field, and no limit with memo fields. Different database servers on different platforms each have their own limits.

Literal character strings must be enclosed in matching single or double quotation marks, or square brackets, as shown in the following examples:

```
'text'
"text"
[text]
```

A literal null string, or empty string, is indicated by two matching quotation marks or a set of square brackets with nothing in between.

Numeric data

Visual dBASE supports a single numeric data type. It does not distinguish between integers and non-integers, which are also referred to as floating-point numbers. Table formats vary in the types of numbers they store. Some support short (16-bit) and long (32-bit) integers or currency in addition to a numeric format. When these numbers are read into *Visual* dBASE, they are all treated as plain numbers. When numbers are stored into tables, they are automatically truncated to fit the table format.

In *Visual* dBASE, a numeric literal may contain a fractional portion, or be multiplied by a power of 10. The following are all valid numeric literals:

- 42
- 5e7
- .315
- 19e+4
- 4.6
- 8.306E-2

As the examples show, the “E” to designate a power of 10 may be uppercase or lowercase, and you may include a plus sign to indicate a positive power of 10 even though it is unnecessary.

In addition to decimal literals, you may use octal (base 8) or hexadecimal (base 16) literal integers. If an integer starts with a zero (0), it is assumed to be octal, with digits from 0 to 7. If it starts with 0x or 0X, it is hexadecimal, with the digits from 0 to 9 and the letters A to F, uppercase or lowercase. For example,

| Literal | Base | Decimal value |
|---------|-------------|---------------|
| 031 | Octal | 25 |
| 0x64 | Hexadecimal | 100 |

Logical data

A logical, or boolean, value can be only one of two things: true or false. These two logical values are expressed literally in *Visual* dBASE by the keywords *true* and *false*.

For compatibility with earlier versions of *Visual* dBASE, you may also express true as .T. or .Y., and false as .F. or .N.

Date data

Visual dBASE features native support for dates, including date arithmetic. To specify a literal date, enclose the date in curly braces. The order of the day, month, and year depends on the current setting of SET DATE, which derives its default setting from the Regional Settings in the Windows Control Panel. For example, if SET DATE is MDY (month/day/year), then the literal date:

```
{10/02/97}
```

is October 2nd, 1997. The way *Visual* dBASE handles two-digit years depends on the setting of SET EPOCH. The default is to interpret two-digit years as a year in the 1900s. Curly braces with nothing between them represent a special date value, known as a blank date.

Null values

Visual dBASE supports a special value represented by the keyword *null*. It is its own data type, and is used to indicate a nonexistent or undefined value. A null value is different from a blank or zero value; null is the absence of a value.

The new DBF7 (dBASE) table type support nulls, as do most other tables, including DB (Paradox). Older DBF formats do not. A null value in a field would indicate that no data has been entered into the field, like in a new row, or that the field has been emptied on purpose. In certain summary operations, null fields are ignored. For example, if you are averaging a numeric field, rows with a null value in the field are ignored. If instead a null value was considered to be zero or some other value, it would affect the average.

null is also used in *Visual* dBASE to indicate an empty function pointer, a property or variable that is supposed to refer to a function, but doesn't contain anything.

Database-specific data types

There are a number of data types supported by different databases that do not have a direct equivalent in *Visual* dBASE. The following list is not exhaustive; a new or upgraded table format may introduce new types. In any case, the type is represented by the closest matching *Visual* dBASE data type, with the string type being the catchall, since all data can be represented as a bunch of bytes.

The common database-specific types are:

- Memo
- Binary and OLE

Memo data

As far as *Visual* dBASE is concerned, a memo is just a character string; potentially a very long one. For tables, it is important to distinguish between a character field, which is of fixed and usually small size, and a memo field,

which is unlimited in size. For example, a character field might contain the title of a court decision, and the memo field contain the actual text of that court decision.

Binary and OLE data

Binary and OLE data are similar to memos, except that they are usually meant to be modified by external programs, not *Visual dBASE*. For example, a binary field might contain a graphic bitmap, which *Visual dBASE* can display, but you cannot edit the bitmap with *Visual dBASE*.

Programming data types

There are three data types used specifically for programming:

- Object reference
- Function pointer
- Codeblock

These types are explained later, in the context in which they are used.

Operators and symbols

An *operator* is a symbol, set of symbols, or keyword that performs an operation on data. *Visual dBASE* provides many types of operators, used throughout the language, in the following categories:

| Operator category | Operator symbols |
|----------------------|-------------------------|
| Assignment | = := += -= *= /= %= |
| Comparison | = == <> # > < >= <= \$ |
| String concatenation | + - |
| Numeric | + - * / % ^ ** ++ -- |
| Logical | AND OR NOT |
| Object | . [] NEW :: |
| Call, Indirection | () |
| Alias | -> |
| Macro | & |

All operators require either one or two arguments, called *operands*. Those that require a single operand are called *unary operators*; those requiring two operands are called *binary operators*. For example, the logical NOT operator is a unary operator:

```
not endOfSet
```

The (*) is the binary operator for multiplication, for example,

```
59 * 436
```

If you see a symbol in *Visual* dBASE code, it's probably an operator, but not all symbols are operators. For example, quote marks are used to denote literal strings, but are not operators, since they do not act upon data—they are part of the representation of a data type.

Another common symbol is the end-of-line comment symbol, a double slash. It and everything on the line after it are ignored by *Visual* dBASE. For example,

```
calcAverages()           // Call the function named calcAverages
```

All operators and symbols are described in full in Chapter 4.

Names

Names are given to variables, fields in work areas, properties, events, methods, functions, and classes. The following rules are the naming conventions in *Visual* dBASE:

- A name begins with an underscore or letter, and contains any combination of underscores, letters, spaces, or digits.
- If the name contains spaces, it must be enclosed in colons.
- The letters may be uppercase or lowercase. *Visual* dBASE is not case-sensitive.
- With *Visual* dBASE, only the first 32 characters in a name are significant. There can be more than 32, but the extra characters are ignored. For example, the following two names are considered to be the same:

```
theFirst_32_CharactersAreTheSameButTheRestArent
theFirst_32_CharactersAreTheSameAndTheRestDontMatter
```

The following are some examples of valid names:

```
x
:First name:
DbException
Form
messages1_onOpen
```

Expressions

An *expression* is anything that results in a value. Expressions are built from literal data, names, and operators.

Basic expressions

The simplest expression is a single literal data value; for example,

```
6 // The number 6
"eloign" // The string "eloign"
```

You can use operators to join multiple literals; for example,

```
6 + 456 * 3 // The number 1374
"sep" + "a" + "rat" + "e" // The string "separate"
```

To see the value of an expression in the Command window, precede the expression with the ? symbol:

```
? 6 + 456 * 3 // Displays 1374
```

Variables

Variables are named locations in memory where you store data values: strings, numbers, logical values, dates, nulls, object references, function pointers, and codeblocks. You assign each of these values a name so that you can later retrieve them or change them.

You can use these values to store user input, perform calculations, do comparisons, define values that are used as parameters for other statements, and much more.

Assigning variables

Before a variable can be used, a value must be assigned to it. Use a single equal sign to assign an expression to a variable; for example,

```
alpha = 6 + 456 * 3 // alpha now contains 1374
```

If the variable does not exist, it is created. There are special assignment operators that will assign to existing variables only, and others that combine an arithmetic operation and an assignment.

Using variables and field names in expressions

When a variable is not the target (on the left side) of an assignment operator, its value is retrieved. For example, type the following lines in the Command window, without the comments:

```
alpha = 6 // Assigns 6 to alpha
beta = alpha * 4 // Assigns values of alpha (6) times 4 to beta
? beta // Displays 24
```

In the same way, when the name of a field in a work area is used in an expression, its value for the current record is retrieved. (Note that assignment operators do not work on fields in work areas; you must use the REPLACE command.) Continuing the previous example:

```
use FISH // Open Fish table in current work area
? Name // Display value of Name field in first record
? :Length CM: // Display value of Length CM field in first record
// Colons required around field name because it contains spaces
? :Length CM: * beta // Display value of field multiplied by variable
```

For information on referencing fields in different work areas and resolving name conflicts between variables and field names, see “Alias operator” on page 4-12.

Type conversion

When combining data of two different types with operators, they must be converted to a common type. If the type conversion does not occur automatically, it must be done explicitly.

Automatic type conversion

Visual dBASE features automatic type conversion between its simple data types. When a particular type is expected, either as part of an operation or because a property is of a particular type, automatic conversion may occur. In particular, both numbers and logical values are converted into strings, as shown in the following examples:

```
"There are " + 6 * 2 + " in a dozen"    // The string "There are 12 in a dozen"
"" + 4                                  // The string "4"
"2 + 2 equals 5 is " + ( 2 + 2 == 5 )    // The string "2 + 2 equals 5 is false"
```

As shown above, to convert a number into a string, simply add the number to an empty string. Be careful, though; the following expression doesn’t work as you might expect:

```
"The answer is " + 12 + 1              // The string "The answer is 121"
```

The number 12 is converted to a string and concatenated, then the number 1 is converted and concatenated, yielding “121”. To concatenate the sum of 12 plus 1, use parentheses to force the addition to be performed first:

```
"The answer is " + (12 + 1)            // The string "The answer is 13"
```

Explicit type conversion

In addition to automatic type conversion, there are a number of functions to convert from one type to another:

- String to number: use the VAL() function
- Number to formatted string: use the STR() function
- Date to string: use the DTOC() function
- String to date: use the CTOD() function

Arrays

Visual dBASE supports a rich set of array classes. An array is an *n*-dimensional list of values stored in memory. Each entry in the array is called an element, and each element in an array can be treated like a variable.

To create an array, you can use the object syntax detailed in Chapter 10, “Array objects,” but for a one-dimensional array, you can also use the literal array syntax.

Literal arrays

A literal array declares and populates an array in a single expression. For example,

```
aTest = { 4, "yclept", true }
```

creates an array with three elements:

- The number 4
- The string “yclept”
- The logical value *true*

and assigns it to the variable `aTest`. The three elements are enclosed in curly braces (the same curly braces used for dates) and separated by commas.

Array elements are referenced with the index operator, the square brackets (`[]`). Elements are numbered from one. For example, the third element is element number 3:

```
? aTest[ 3 ]           // Displays true
```

You can assign a new value directly to an element, just like a variable:

```
aTest[ 3 ] = false    // Element now contains false
```

Complex expressions

The following is an example of a complex expression that uses multiple names, operators, and literal data. It is preceded by a question mark so that when it's typed into the Command window, it displays the resulting value:

```
? { "1st", "2nd", "3rd", "4th" } [ ceiling( month( date() ) / 3 ) ] + " quarter"
```

Except for the question mark, the entire line is a single complex expression, made up of many smaller basic expressions. The expression is evaluated as follows:

- A literal array of literal strings is enclosed in braces, separated by commas. The strings are enclosed in double quotation marks.
- The resulting array is referenced using the square brackets as the index operator. Inside the square brackets is a numeric expression.
- The numeric expression uses nested functions, which are evaluated from the inside out. First, the `DATE()` function returns the current date. The `MONTH()` function returns the month of the current date.
- The month is divided by the number 3, then the `CEILING()` function rounds the number up to the nearest integer.
- The string containing the ordinal number for the calendar quarter that corresponds to the month of the current date is extracted from the array, which is then added to the literal string “quarter”.

The value of this complex expression is a string like “4th quarter”.

Statements

A statement is an instruction that directs *Visual* dBASE to perform a single action. This action may be simple or it may be complex, causing other actions to occur. You may type and execute individual statements in the Command window.

Basic statements

There are four types of basic statements:

- *Visual* dBASE commands

These commands make up a significant portion of the entries in the *Language Reference*. For example:

```
clear                // Clears the Command window
erase TEMP.TXT       // Erases a file on the disk
build from FISHBASE  // Creates an executable
? time()             // Displays the current time
```

- Assignment statements

A statement may include only one assignment operator, although the value assigned may be a very complex expression. For example:

```
clear = 14           // Assign 14 to variable named clear
f = new Form()       // NEW and call operator on class name Form, assigned to variable f
```

Note that the first example uses the word “clear”, but because the syntax of the statement a variable is created instead of executing the command. While creating variables with the same name as a command keyword is allowed, it is strongly discouraged.

- *Visual* dBASE expressions

An expression is a valid statement. If the expression evaluates to a number, it is equivalent to a GO command. For example:

```
6                  // Goto record 6
3 + 4              // Goto record 7
date()            // Get today's date and throw it away
f.open()          // Call object f's open() method
```

- Embedded SQL statements

Visual dBASE features native support for SQL statements. You may type an SQL statement in the Command window, or include them in programs. If the command results in an answer table, that table is opened in the current work area. For example:

```
select * from FISH  // Open FISH table in current work area
```

Control statements

Visual dBASE supports a number of *control statements* that can affect the execution of other statements. Control statements fall into the following categories:

- Conditional execution
 - IF
 - DO CASE
- Looping
 - FOR
 - DO WHILE
 - DO...UNTIL
- Object manipulation
 - WITH
- Exception handling
 - TRY

These control statements are fully documented in Chapter 5, “Core language.”

Functions and codeblocks

In addition to the built-in functions, you may create your own. A function is a code module—a set of statements—to which a name is assigned. The statements can be called by the function name as often as needed. Functions also provide a mechanism whereby the function can take one or more parameters that are acted upon by the function.

A function is called by following the function name with a set of parentheses, which act as the call operator. When discussing a function, the parentheses are included to help distinguish functions from other language elements like variables.

For example, the function LDoM() takes a date parameter *dArg* and returns the last day of the month of that date.

```
function LDoM( dArg )
  local dNextMonth
  dNextMonth = dArg - date( dArg ) + 45  // Day in the middle of next month
  return dNextMonth - day( dNextMonth )
```

Functions are identified by the keyword **FUNCTION** in a program file; they cannot be typed into the Command window. While many functions use **RETURN** to return a value, they are not required to do so.

Function pointers

The name of a function that you create is actually a pointer to that function. Applying the call operator () to a function pointer calls that function. (Built-in functions work differently; there is no function pointer.)

Function pointers are a distinct data type, and can be assigned to other variables or passed as parameters. The function can then be called through that function pointer variable.

Function pointers enable you to assign a particular function to a variable or property. The decision can be made up front and changed as needed. Then that function can be called as needed, without having to decide which function to call every time.

Codeblocks

While a function pointer points to a function defined in a program, a codeblock is compiled code that can be stored in a variable or property. Codeblocks do not require a separate program; they actually contain code. Codeblocks are another distinct data type that can be stored in variables or properties and passed as parameters, just like function pointers.

Codeblocks are called with the same call operator that functions use, and may receive parameters.

There are two types of codeblocks:

- Expression codeblocks
- Statement codeblocks

Expression codeblocks return the value of a single expression. Statement codeblocks act like functions; they contain one or more statements, and may return a value.

In terms of syntax, both kinds of codeblocks are enclosed in curly braces ({}) and

- Cannot span multiple lines.
- Must start with either two pipe characters (| |) or a semicolon (;)
 - If ; it must be a statement codeblock with no parameters
 - If | | it may be either an expression or statement codeblock
- The | | are used for parameters to the codeblock, which are placed between the two pipe characters. They may also have nothing in-between, meaning no parameters for either an expression or statement codeblock.
- Parameters inside the | |, if any, are separated by commas.
- For an expression codeblock, the | | must be followed by one and only one expression, with no ; These are valid expression codeblocks:

```
{| | false}
{| | date() }
```



```
{|x| x * x}
```

- Otherwise, it is a statement codeblock. A statement codeblock may begin with `||` (again, with or without parameters in-between).
- Each statement in a statement codeblock must be *preceded* by a `;` symbol. These are valid statement codeblocks (the first two are functionally the same):

```
{; clear}
{||; clear}
{|x|; ? x}
{|x|; clear; ? x}
```

- You may use a RETURN inside a statement codeblock, just like with any other function. (A RETURN is implied with an expression codeblock.) For example,

```
{|n|; for i=2 to sqrt(n); if n % i == 0; return false; endif; endfor; return true}
```

Because codeblocks don't rely on functions in programs, you can create them in the Command window. For example,

```
square = {|x| x * x} // Expression codeblock
? square( 4 ) // Displays 16
// A statement codeblock that returns true if a number is prime
p = {|n|; for i=2 to sqrt(n); if n % i == 0; return false; endif; endfor; return true}
? p( 23 ) // Displays true
? p( 25 ) // Displays false
```

As mentioned previously, curly braces are also used for literal dates and literal arrays. Compare the following:

```
{10} // A literal array containing one element with the value 10
{10/5} // A literal array containing one element with the value 2
{10/5/97} // A literal date
{||10/5} // An expression codeblock that returns 2
```

Codeblocks vs. functions

A codeblock is a convenient way to create a small anonymous function and assign it directly to a variable or property. The code is physically close to its usage and easy to see. In contrast, a function pointer refers to a function defined elsewhere, perhaps much later in the same program file, or in a different program file.

Functions are easier to maintain. Their syntax is not cramped like codeblocks, and it's easier to include readable comments in the code. In a class definition, all FUNCTION definitions are all together at the bottom. Codeblocks are scattered throughout the constructor. If you want to run the same code from multiple locations, using function pointers that point to the same function means that changing the code requires changing the function once; multiple codeblocks would require changing each codeblock individually.

You can create a codeblock at runtime by constructing a string that looks like a codeblock and using the macro operator to evaluate it.

Objects and classes

An object is a collection of properties. Each of these properties has a name. These properties may be simple data values, such as numbers or strings, or references to code, such as function pointers and codeblocks. A property that references code is called a method. A method that is called by *Visual* dBASE in response to a user action is called an event.

Objects are used to represent abstract programming constructs, like arrays and files, and visual components, like buttons and forms. All objects are initially based on a class, which acts as a template for the object. For example, the `PushButton` class contains properties that describe the position of the button, the text that appears on the button, and what the button should do when it is clicked. All these properties have default values. Individual button objects are *instances* of the `PushButton` class that have different values for the properties of the button.

Visual dBASE contains many built-in, or *stock*, classes, which are documented throughout the *Language Reference*. You can extend these stock classes or build your own from scratch with a new CLASS definition.

While the class acts as a formal definition of an object, you can always add properties as needed. This is called *dynamic subclassing*.

Dynamic subclassing

To demonstrate dynamic subclassing, start with the simplest object: an instance of the `Object` class. The `Object` class has no properties. To create an object, use the `NEW` operator, along with the class name and the call operator, which would include any parameters for the class (none are used for the `Object` class).

```
obj = new Object()
```

This statement creates a new instance of the `Object` class and assigns an *object reference* to the variable `obj`. Unlike variables that contain simple data types, which actually contain the value, an object reference variable contains only a reference to the object, not the object itself. This also means that making a copy of the variable:

```
copy = obj
```

does not duplicate the object. Instead, you now have two variables that refer to the same object.

To assign values to properties, use the dot operator. For example,

```
obj.name = "triangle"
obj.sides = 3
obj.length = 4
```

If the property does not exist, it is added; otherwise, the value of the property is simply reassigned. This behavior can cause simple bugs in your programs. If you mistype a property name during an assignment, for example,

```
obj.wides = 4    // should be s, not w
```

a new property is created instead of changing the value of the existing property you intended. To catch these kinds of problems, use the assignment-only `:=` operator when you know you are not initializing a property or variable. If you attempt to assign a value to a property or variable that does not exist, an error occurs instead of creating the property or variable. For example:

```
obj.wides := 4 // Error if wides property does not already exist
```

Methods

A method is a function or codeblock assigned to a property. The method is then called through the object via the dot and call operators. Continuing the example above:

```
obj.perimeter = {|| this.sides * this.length}
? obj.perimeter() // Displays 12
```

As you may have deduced by now, the object referred to by the variable *obj* represents a regular polygon. The perimeter of such a polygon is the product of the length of each side and the number of sides.

The reference *this* is used to access these values. In the method of an object, the reference *this* always refers to the object that called the method. By using *this*, you can write code that can be shared by different objects, and even different classes, as long as the property names are the same.

A simple class

Here is a class representing the polygon:

```
class RegPolygon
  this.sides = 3 // Default number of sides
  this.length = 1 // and default length

  function perimeter()
    return this.sides * this.length
endclass
```

The top of the CLASS definition, up to the first FUNCTION, is called the class *constructor*, which is executed when an instance of the class is created. In the constructor, the reference *this* refers to the object being created. The *sides* and *length* properties are added, just as they were before.

The function in the class definition is considered a method, and the object automatically has a property with the same name as the method that points to the method. The code is the same, but now instead of a codeblock, the method is a function in the class. Methods have the advantage of being easier to maintain and subclass.

Programs

A program contains any combination of the following items:

- Statements to be executed
- Functions and classes that may be called
- Comments

The *Visual* dBASE compiler also supports a standard language preprocessor, so a program that is run by *Visual* dBASE may contain preprocessor directives. These directives are not part of the *Visual* dBASE language; instead they form a separate simple language that can affect the code compilation process, and are explained later.

Program files

A program file may have any file-name extension, although there are a number of defaults:

- A program containing a form is .WFM
- A program containing a report is .REP
- Any other program is .PRG

These file-name extensions are assumed by the Navigator and the Source Editor.

When a program is compiled into byte code by *Visual* dBASE, it stores the byte code in a file with the same name and extension, but it changes the last character of the extension to the letter "O": .PRG becomes .PRO, .WFM becomes .WFO, and .REP becomes .REO.

Program execution

Use the DO command to run a program file, or double-click the file in the Navigator. If you run the program through the Navigator, the equivalent DO command will be streamed out to the Command window and executed. You can also call a .PRG program by name with the call operator, the parentheses, in the Command window; for example,

```
sales_report()
```

will attempt to execute the file SALES_REPORTS.PRG. Since the operating system is not case-sensitive about file names when searching for files, neither is *Visual* dBASE.

A basic program simply contains a number of *Visual* dBASE statements, which are executed once in the order that they appear in the program file, from the top down. For example, the following four statements remember the current directory, switch to another directory, execute a report, and switch back to the previous directory:

```
cDir = set( "DIRECTORY" )  
cd C:\SALES
```

```
do DAILY.REP
cd &cDir
```

Control statements, discussed earlier, are acted upon as they occur; they may affect the execution of the code that they contain. Some statements may be executed only when a certain condition is true and other statements may be executed more than once in a loop. But even within these control statements, the execution is still basically the same, from the top down.

When and if there are no more statement to execute, the program ends, and control returns to where the program was called. For example, if the program was executed from the Command window, then control returns to the Command window and you can do something else.

Functions and classes

Functions and classes affect execution in two ways. First, when a function or class definition is encountered in the straight top-down execution of a program, execution in that program is terminated.

The second effect is that when a function, class constructor, or method is called, execution jumps into that function or class, executes that code in the usual top-down fashion, then goes back to where the call was made and continues where it left off.

Comments

Use comments to include notes to yourself or others. The contents of a comment do not follow any *Visual* dBASE rules; include anything you want. Comments are stripped out at the beginning of the program compilation process.

A program will typically contain a group of comments at the beginning of the file, containing information like the name of the program, who wrote it and when, version information, and instructions for using it. But the most important use for comments is in the code itself, to explain the code—not obvious things like this:

```
n++    // Add one to the variable n
```

(unless you're writing example code to explain a language) but rather things like what you're doing in the overall scheme of the program, or why you decided to do something in a particular way. Decisions that are obvious to you when you write a statement will often completely bewilder you a few months later. Write comments so that they can be read by others, and put them in as you code, since there's rarely time to add them in after you're done, and you may have forgotten what you did by then anyway.

Preprocessor directives

A preprocessor directive must be on its own line, and starts with the number sign (#).

Because preprocessor directives are not part of the *Visual* dBASE language, you cannot execute them in the Command window.

For more information about using preprocessor directives, see Chapter 22, “Preprocessor.”

A simple program

Here is a simple program that creates an instance of the RegPolygon class, changes the length of a side, and displays the perimeter:

```
// Polygon.prg
// A simple program example
//
local poly
poly = new RegPolygon()
poly.length = 4
? poly.perimeter() // Displays 12

class RegPolygon
    this.sides = 3 // Default number of sides
    this.length = 1 // and default length

    function perimeter()
        return this.sides * this.length
endclass
```

Syntax conventions

The *Language Reference* uses specific symbols and conventions in presenting the syntax of *Visual* dBASE language elements. This chapter describes the symbols used in syntax and provides information on interpreting the syntax conventions.

Syntax notation

Statements, methods, and functions are described with syntax diagrams. These syntax diagrams consist of a least one fixed language element—the one being documented—and may include arguments, which are enclosed in angle brackets (< >).

The *Visual* dBASE language is not case-sensitive.

The following table describes the symbols used in syntax:

| Symbol | Description |
|--------|--|
| < > | Indicates an argument that you must supply |
| [] | Indicates an optional item |
| | Indicates two or more mutually exclusive options |
| ... | Indicates an item that may be repeated any number of times |

Arguments are often expressions of a particular type. The description of an expression argument will indicate the type of argument expected, as listed in the following table:

| Descriptor | Type |
|-------------|------------------------|
| <i>expC</i> | A character expression |
| <i>expN</i> | A numeric expression |

| Descriptor | Type |
|-------------|---|
| <i>expL</i> | A logical or boolean expression; that is, one that evaluates to <i>true</i> or <i>false</i> |
| <i>expD</i> | A date expression |
| <i>exp</i> | An expression of any type |
| <i>oRef</i> | An object reference |

All the arguments and optional elements are described in the syntax description.

Unlike legacy dBASE command and function keywords, which are shown in uppercase letters, property names are capitalized differently. Property names are camel-capped, that is, they contain both uppercase and lowercase letters if the name consists of more than one word. If the property is a method, the name is followed by parentheses. Examples of properties include *onAppend*, *onRightMouseDown*, *checked*, and *close()*.

These conventions help you differentiate the language elements; for example,

- DELETE is a command
- *delete* is a property
- DELETED() is a function
- *delete()* is a method

These typographical conventions are for readability only. When writing code, you can use any combination of uppercase and lowercase letters.

Note In *Visual* dBASE 7, you must refer to classes and properties by their full names. However, you can still abbreviate some keywords in the dBASE language to the first four characters, though for reasons of readability and clarity such abbreviation is not recommended.

Syntax example

The syntax entries for the EXTERN statement illustrate all of the syntax symbols:

```
EXTERN [ CDECL | PASCAL | STDCALL ] <return type> <function name>
      ([<parameter type> [, <parameter type> ... ]])
      <filename>
```

- The square brackets enclosing the calling convention, [CDECL | PASCAL | STDCALL], means the item is optional. The pipe character between the three calling conventions is an "or" indicator. In other words, if you want to use a calling convention, you must choose one of the three.
- <return type> and <function name> are both required arguments.

- The parentheses are fixed language elements, and thus also required. Inside the parentheses are optional *<parameter type>* arguments, as indicated by the square brackets.
 - The location of the comma inside the second square bracket indicates that the comma is needed only if more than one *<parameter type>* is specified.
 - The ellipsis (...) at the end means that any number of parameter type arguments may be specified (with a comma delimiter, if more than one is used).
- *<filename>* is a required argument.

A simple EXTERN statement with neither of the two optional elements would look like this:

```
extern CINT angelsOnAPin() ANSWER.DLL
```

The *<return type>* argument is CINT, and the *<function name>* is *angelsOnAPin*.

A more complicated EXTERN statement with a calling convention and parameters would look like this:

```
extern PASCAL CLONG wordCount( CPTR, CLOGICAL ) ANSWER.DLL
```

Capitalization guidelines

The following guidelines describe the standard capitalization of various language elements. Although *Visual dBASE* is not a case-sensitive language, you are encouraged to follow these guidelines in your own scripts.

- Commands and built-in functions are shown in uppercase in descriptions so that they stand out, but are all lowercase in code examples.
- Class names start with a capital letter. Multiple-word class names are joined together without any separators between the words, and each word starts with a capital letter. For example,

```
Form
PageTemplate
```

- Property, event, and method names start with a lowercase letter. If they are multiple-word names, the words are joined together without any separators between the words, and each word (except the first) starts with a capital letter. They also appear italicized in the *Language Reference*. For example,

```
color
dataLink
showMemoEditor( )
```

- Variable and function names are capitalized like property names.
- Manifest constants created with the #define preprocessor directive are all uppercase, with underscores between words. For example,

```
ARRAY_DIR_NAME
```

NUM_REPS

- Field names and table names from DBF tables are in all uppercase in code so that they stand out.

SET command defaults

If a SET... command has a default setting, it is shown in uppercase in its syntax entry; the other options are shown in lowercase. For example, with:

SET DELETED ON | off

SET DELETED may be either ON or OFF. It is ON by default.

In some cases, a setting may depend on the Regional Settings in the Windows Control Panel, so there is no explicit default. For example:

SET CURRENCY left | right

Note that the SET command saves all settings to the VDB.INI file. Those settings then become the default when you start *Visual* dBASE or issue CREATE SESSION.

Operators and symbols

An *operator* is a symbol, set of symbols, or keyword that specifies an operation to be performed on data. Data is supplied in the form of arguments, or *operands*.

For example, in the expression “total = 0”, the equal sign is the operator and “total” and “0” are the operands. In this expression, the numeric operator “=” takes two operands, which makes it a *binary* operator. Operators that require just one operand (such as the numeric increment operator “++”) are known as *unary* operators.

Operators are categorized by type. *Visual* dBASE’s operators are classified as follows:

| Operator symbols | Operator category |
|------------------------|----------------------|
| = := += -= *= /= %= | Assignment |
| = == <> # > < >= <= \$ | Comparison |
| + - | String concatenation |
| + - * / % ^ ** ++ -- | Numeric |
| AND OR NOT | Logical |
| . [] NEW :: | Object |
| () | Call, Indirection |
| -> | Alias |
| & | Macro |

Most symbols you see in *Visual* dBASE code are operators, but not all. Quotation marks, for example, are used to denote literal strings and thus are part of the representation of a data type. Since they don’t act upon data, they’re a “non-operational” symbol.

You can use the following non-operational symbols in *Visual* dBASE code:

| Symbols | Name/meaning |
|-------------------|--|
| ; | Statement separator, line continuation |
| // && | End-of-line comment |
| * | Full-line comment |
| /* */ | Block comment |
| { } { ; } { } | Literal date/literal array/codeblock markers |
| " " ' ' [] | Literal strings |
| :: | Name/database delimiters |
| # | Preprocessor directive |

Finally, the following symbols are used as *Visual* dBASE commands when they are used to begin a statement:

| Symbols | Name/meaning |
|---------|--|
| ? ?? | Displays streaming output (page 18-1) |
| ! | Runs program or operating system command (page 11-5) |

Operator precedence

Visual dBASE applies strict rules of precedence to compound expressions. In expressions that contain multiple operations, parenthetical groupings are evaluated first, with nested groupings evaluated from the “innermost” grouping outward. After all parenthetical groupings are evaluated, the rest of the expression is evaluated according to the following operator precedence:

| Order of precedence (highest to lowest) | Operator description or category |
|---|---|
| & | Macro |
| (expression) | Parenthetical grouping, all expressions |
| -> | Alias |
| () [] . NEW :: | Object operators: call; member (square bracket or dot); <i>new</i> ; scope resolution |
| + - ++ -- | Unary plus/minus, increment/decrement |
| ^ ** | Exponentiation |
| * / % | Multiply, divide, modulus |
| + - | Addition, subtraction |
| = == <> # < <= > >= \$ | Comparison |
| NOT | Logical Not |
| AND | Logical And |
| OR | Logical Or |
| = := += -= *= /= %= | Assignment |

In compound expressions that contain operators from the same precedence level, evaluation is conducted on a literal left-to-right basis. For example, no operator precedence is applied in the expressions $21/7*3$ and $3*21/7$ (both return 9).

Here's another example:

```
4+5*(6+2*(8-4)-9)%19>=11
```

This example is evaluated in the following order:

```
8-4=4
2*4=8
6+8=14
14-9=5
5*5=25
25%19=6
4+6=10
```

The result is the logical value *false*.

Assignment operators

Assign/create operator: =

Assignment-only operator: :=

Arithmetic assignment operators: += -= *= /= %=

Syntax

```
x = n
y = x
x += y
```

Description

Assignment operators are binary operators that assign the value of the operand on the right to the operand on the left.

The standard assignment operator is the equal sign. For example, $x = 4$ assigns the value 4 to the variable x , and $y = x$ assigns the value of the variable x (which must already have an assigned value) to the variable y . If the variable or property on the left of the equal sign does not exist, it is created.

To prevent the creation of a variable or property if it does not exist, use the assignment-only $:=$ operator. This operator is particularly useful when assigning values to properties. If you inadvertently misspell the name of the property with the $=$ operator, a new property is created; your code will run without error, but it will not behave as you intended. By using the $:=$ operator, if the property (or variable) does not exist, an error occurs.

The arithmetic assignment operators are shortcuts to self-updating arithmetic operations. For example, the expression $x += y$ means that x is assigned its own value plus that of y ($x = x + y$). Both operands must already have assigned values, or an error results. Thus, if the operand x has already been assigned the value 4 and y has been assigned the value 6, the expression $x += y$ returns 10.

+ (“plus”) operator

Numeric, String

Addition, concatenation, unary positive operator.

Syntax `n + m`
`date + n`
`"str1" + "str2"`
`"str" + x`
`x + "str"`
`+n`

Description The “plus” operator performs a variety of additive operations:

- It adds two numeric values together.
- You may add a number to a date (or vice-versa). The result is the day that many days in the future (or the past if the number is negative). Adding any number to a blank date always results in a blank date.
- It concatenates two strings.
- You may concatenate any other data type to a string (or vice versa). The other data type is converted into its display representation:
 - Numbers become strings with no leading spaces. Integer values eight digits or less have no decimal point or decimal portion. Integer values larger than eight digits and non-integer values have as many decimals places as indicated by SET DECIMALS.
 - The logical values *true* and *false* become the strings “true” and “false”.
 - Dates (primitive dates and Date objects) are converted using DTOC().
 - Object references to arrays are converted to the word “Array”.
 - References to objects of all other classes are converted to the word “Object”.
 - Function pointers take on the form “Function: “ followed by the function name.

Note Adding the value *null* to anything (or anything to *null*) results in the value *null*.

The plus sign may also be used as a unary operator to indicate no change in sign, as opposed to the unary minus operator, which changes sign. Of course, it is generally superfluous to indicate no change in sign; the unary plus is rarely used.

Example These examples demonstrate addition and concatenation.

```
"this &" + " that"      // = the string "this & that"  
5 + 5                   // = the number 10
```

```

"this & " + 5 + " more" // = the string "this & 5 more"
5 + "-5"                // = the string "5-5"
date() + 7               // = same day next week
"" + Form::open          // = the string "Function: FORM::OPEN"
? 3 + 4 + "abc" + false // = the string "7abcfalse"
? false + 3 + 4 + "abc" // Error: unexpected type

```

The last two examples demonstrate the standard left-to-right precedence of the + operator. In the first example, 4 is added to 3, which yields 7. The string "abc" is added, so the number is converted to its display representation, resulting in the string "7abc". Then the value *false* is added, which is also converted to string, yielding "7abcfalse". But in the second example, the first addition attempts to add 3 to the value *false*, which is not allowed; an error occurs.

See also - ("minus") operator

- ("minus") operator

Subtraction, concatenation, unary negative operator.

Syntax

```

n - m
date - n
date - date
"str1" - "str2"
"str" - x
x - "str"
-n

```

Description The "minus" operator is similar to the "plus" operator. It subtracts two numbers, and subtracts days from a date. You may also subtract one date from another date; the result is the number of days between the two dates. If you subtract a blank date from another date, the result is always zero.

The minus symbol is also used as the unary negation operator, to change the sign of a numeric value.

You may concatenate two strings, or a string with any other data type, just like with the plus operator. The difference is that with the minus operator, the trailing blanks from the first operand are removed before the concatenation, and placed at the end of the result. This means that the concatenation with either the plus or minus results in a string with the same length, but with the minus operator, the trailing blanks are combined at the end of the result.

If you want to trim field values when creating an expression index for a DBF table, use the minus operator.

Example Suppose you have a DBF table with last name and first name fields, both 16 characters wide. Compare the result of the plus and minus operators:

```
"Bailey-Richter" + "Gwendolyn" ==> "Bailey-Richter Gwendolyn"
```

```
"Bailey-Richter " - "Gwendolyn" ==> "Bailey-RichterGwendolyn"
```

It may be more useful to include a comma between the last name and first name:

```
"Bailey-Richter " - ", " - "Gwendolyn" ==> "Bailey-Richter,Gwendolyn"
```

The last name and comma are concatenated, moving the trailing blanks after the comma, then that is concatenated to the first name, moving the trailing blanks after the last name. By separating the last name and first name, the comma ensures that the names are sorted correctly, and it makes searching—particularly interactive incremental searching—easier. The command to create such an index tag would look like:

```
index on upper( LAST_NAME - ", " - FIRST_NAME ) tag FULL_NAME
```

The minus operator results in index keys that are all the same length, something that you wouldn't get by using the TRIM() function.

See also + (“plus”) operator

Numeric operators

Binary numeric operators: + − ∗ / % ^ **

Unary numeric operators: ++ --

Syntax

```
n + m
n++
n--
++n
```

Description Perform standard arithmetic operations on two operands, or increment or decrement a single operand.

All of these operators take numeric values as operands. The + (plus) and - (minus) symbols can also be used to concatenate strings.

As binary numeric operators, the +, −, ∗, and / symbols perform the standard arithmetic operations addition, subtraction, multiplication and division.

The modulus operator returns the remainder of an integral division operation on its two operands. For example, 50%8 returns 2, which is the remainder after dividing 50 by 8.

You may use either ^ or ** for exponentiation. For example, 2^5 is 32.

The increment/decrement operators ++ and -- take a variable or property and increase or decrease its value by one. The operator may be used before the variable or property as a *prefix* operator, or afterward as *postfix* operator. For example,

```
n = 5      // Start with 5
? n++     // Get value (5), then increment
? n       // Now 6
```



```
? ++n    // Increment first, then get value (7)
? n      // Still 7
```

If the value is not used immediately, it doesn't matter whether the ++/-- operator is prefix or postfix, but the convention is postfix.

Logical operators

Binary logical operators: AND OR

Unary logical operator: NOT

Syntax a AND b
a OR b
NOT b

Description The AND and OR logical operators return a logical value (*true* or *false*) based on the result of a comparison of two operands. In a logical AND, both expressions must be *true* for the result to be *true*. In a logical OR, if either expression is *true*, or both are *true*, the result is *true*; if both expressions are *false*, the result is *false*.

When *Visual* dBASE evaluates an expression involving AND or OR, it uses *short-circuit evaluation*:

- *false* AND *<any expL>* is always *false*
- *true* OR *<any expL>* is always *true*

Because the result of the comparison is already known, there is no need to evaluate *<any expL>*. If *<any expL>* contains a function or method call, it is not called; therefore any side effects of calling that function or method do not occur.

The unary NOT operator returns the opposite of its operand expression. If the expression evaluates to *true*, then NOT *exp* returns *false*. If the expression evaluates to *false*, NOT *exp* returns *true*.

You may enclose the logical operators in dots, that is: .AND., .OR., and .NOT. The dots are required in earlier versions of dBASE.

Comparison operators

Comparison operators compare two expressions. The comparison returns a logical *true* or *false* value. Comparing logical expressions is allowed, but redundant; use logical operators instead.

Visual dBASE automatically converts data types in a comparison, using the following rules:

- 1 If the two operands are the same type, they are compared as-is.
- 2 If either operand is a numeric expression, the other operand is converted to a number:

- If a string contains a number only (leading spaces are OK), that number is used, otherwise it is interpreted as an invalid number.
- The logical value *true* becomes one; *false* becomes zero.
- All other data types are invalid numbers.

All comparisons between a number and an invalid number result in *false*.

3 If either operand is a string, the other operand is converted to its display representation:

- Numbers become strings with no leading spaces. Integer values eight digits or less have no decimal point or decimal portion. Integer values larger than eight digits and non-integer values have as many decimals places as indicated by SET DECIMALS.
- The logical values *true* and *false* become the strings “true” and “false”.
- Dates (primitive dates and Date objects) are converted using DTOC().
- Object references to arrays are converted to the word “Array”.
- References to objects of all other classes are converted to the word “Object”.
- Function pointers take on the form “Function: ” followed by the function name.

4 All other comparisons between mismatched data types return *false*.

These are the comparison operators:

| Operator | Description |
|----------|------------------------------------|
| == | Exactly equal to |
| = | Equal to or Begins with |
| <> or # | Not equal to or Doesn't begin with |
| < | Less than |
| > | Greater than |
| <= | Less than or equal to |
| >= | Greater than or equal to |
| \$ | Contained in |

When comparing dates, a blank date comes after (is greater than) a non-blank date.

When comparing Date objects, the date/time they represent are compared; they may be earlier, later, or exactly the same. For all other objects, only the equality tests makes sense. It tests whether two object references refer to the same object.

String equality comparisons are case-sensitive and follow the rules of SET EXACT. The == operator always compares two strings as if SET EXACT is ON. The other equality operators (=, <>, #) use the current setting of SET EXACT. When SET EXACT is ON, trailing blanks in either string are ignored in the

comparison. When SET EXACT is OFF (the default), the = operator act like a “begins with” operator: the string on the left must begin with the string on the right. The <> and # operators act like “does not begin with” operators. Note that there is no single genuinely exactly equal comparison for strings in *Visual dBASE*.

It is recommended that you leave SET EXACT OFF so that you have the flexibility of doing an “exact” comparison or a “begins with” comparison as needed. By definition, all strings “begin with” an empty string, so when checking if a string is empty, always put the empty string on the left of the equality operator.

Warning For compatibility with earlier versions of dBASE, if the string on the right of the = operator is (or begins with) CHR(0) and SET EXACT is OFF, then the comparison always returns *true*. When checking for CHR(0), always use the == operator.

The \$ operator determines if one string is contained in, or is a substring of, another string. By definition, an empty string is *not* contained in another string.

Example To see these examples in action, type them into the Command window.

```
// The usual numeric and string comparisons
? 3 < 4                // true
? "cat" > "dog"        // false

// String comparisons demonstrate SET EXACT rules
set exact off
? "abc" == "abc"       // Obviously true
? "abc" == "abc "      // Trailing space in second operand is ignored: true
? "abc" = "ab"         // 1st operand begins with the characters in the 2nd: true
? "abc" = ""           // true by definition
? "" = "abc"           // false
? "abc" # ""           // false
? "" # "abc"           // true

// Logical comparisons are redundant
valid = true
? valid == true        // true, but so is
? valid               // this
? valid == false       // false, but it's simpler to
? not valid            // use the logical NOT operator

// Date objects compare the date/time they represent
x = new Date()
y = new Date()         // Should be a few seconds later
? x < y                // true, date/time in x is before y
x = new Date( "25 Sep 1996" )
y = new Date( "25 Sep 1996" )
? x == y               // true: objects are different, but date/time is the same

// Other objects test for equality only
a = new Form()
b = new Form()
c = b
? a == b               // false, different objects
```

```
? b == c // true, references to same object
```

Object operators

Object operators are used to create and reference objects, properties, and methods. Here are the Object operators:

| Operator | Description |
|------------|--|
| NEW | Creates a new instance of an object |
| [] | Index operator, which accesses the contents of an object through a numeric or string value |
| . (period) | Dot operator, which accesses the contents of an object through an identifier name |
| :: | Scope resolution operator, to reference a method in a class or call a method from a class. |

NEW operator

The NEW operator creates an object or instance of a specified class.

The following is the syntax for the NEW operator:

```
[<object reference> =] new <class name>([<parameters>])
```

The *<object reference>* is a variable or property in which you want to store a reference to the newly created object.

Note that the reference is optional syntactically; you may create an object without storing a reference to it. For most classes, this results in the object being destroyed after the statement that created it is finished, since there are no references to it.

The following example shows how to use the NEW operator to create a Form object from the Form class. A reference to the object is assigned to the variable *customerForm*:

```
customerForm = new Form()
```

This example creates and immediately uses a Date object. The object is discarded after the statement is complete:

```
? new Date().toGMTString()
```

Index operator

The index operator, [], accesses an object's properties or methods through a value, which is either a number or a character string. The following shows the syntax for using the index operator (often called the array index operator):

```
<object reference>[<exp>]
```

You typically use the index operator to reference elements of array objects, as shown in the following example:

```
aScores = new Array(20) // Create a new array object with 20 elements
aScores[1] = 10         // Change the value of the 1st element to 10
? aScores[1]           // Displays 10 in results pane of Command window
```

Dot operator

The dot operator, (“.”), accesses an object’s properties, events, or methods through a name. The following shows the syntax for using the dot operator:

```
<object reference>[.<object reference> ...].<property name>
```

Objects may be nested: the property of an object may contain a reference to another object, and so on. Therefore, a single property reference may include many dots.

The following statements demonstrate how you use the dot operator to assign values:

```
custForm = new Form()           // Create a new form object
custForm.title = "Customers"    // Set the title property of custForm
custForm.height = 14           // Set the height property of custForm
```

If an object contains another object, you can access the child object’s properties by building a path of object references leading to the property, as the following statements illustrate:

```
custForm.addButton = new Button(custForm) // Create a button in the custForm form
custForm.addButton.text = "Add"           // Set the text property of addButton
```

Scope resolution operator

The scope resolution operator (::, two colons, no space between them) lets you reference methods directly from a class or call a method from a class.

The scope resolution operator uses the following syntax:

```
<class name>|class|super::<method name>
```

The operator must be preceded by either an explicit class name, the keyword CLASS or the keyword SUPER. CLASS and SUPER may be used only inside a class definition. CLASS refers to the class being defined and SUPER refers to the base class of the current class, if any.

<method name> is the method to be referenced or called.

Scope resolution searches for the named method, starting at the specified class and back through the class’s ancestry. Because SUPER starts searching in a class’s base class, it is used primarily when overriding methods.

Call, indirection, grouping operator

Parentheses are used to call functions and methods, and to execute codeblocks. For example:

```
MyClass::MyMethod
```

is a function pointer to a method in the class, while

```
MyClass::MyMethod()
```

actually calls that method. Any parameters to include in the call are placed inside the parentheses. Multiple parameters are separated by commas. Here is an example using a codeblock:

```
rootn = {|x,n| x^(1/n)} // Create expression codeblock with two parameters
? rootn( 27, 3 )       // Displays cube root of 27: 3
```

Some commands expect the names of files, indexes, aliases, and so forth to be specified directly in command—“bare”—not in a character expression. Therefore, you cannot use a variable directly. For example, the ERASE command erases a file from disk. The following code will not work:

```
cFile = getfile( " *.*", "Erase file" ) // Store filename to variable
erase cFile                          // Tries to erase file named "cFile"
```

because the ERASE command tries to erase the file with the name of the variable, not the contents of the variable. To use the variable name in the file, enclose the variable in parentheses. In these commands, the parentheses evaluate the *indirect file reference*, and when used in this way, they are referred to as *indirection operators*:

```
erase ( cFile ) // Spaces inside parentheses optional
```

Macro substitution also works in these cases, but macro substitution can be ambiguous. Indirection operators are recommended in commands where they are allowed.

Finally, parentheses are also used for grouping in expressions to override or emphasize operator precedence. Emphasizing precedence simply means making the code more readable by explicitly grouping expressions in the normal order they are evaluated, so that you don't need to remember all the precedence rules to understand an expression. Overriding precedence uses the parentheses to change the order of evaluation. For example:

```
? 3 + 4 * 5 // Multiplication first, result is 23
? ( 3 + 4 ) * 5 // Do addition first, result is 35
```

See also Macro operator

Alias operator

Designates a field name in a specific work area, or a private or public variable.

Syntax alias->name

Description When using a name that may be a variable or the name of a field in the current work area, the name is matched in the following order:

- 1 Local or static variable
- 2 Field name
- 3 Private or public variable

To resolve the ambiguity, or to refer to a field in another work area, use the alias operator. Aliases are not case-sensitive.

Private and public variables are referenced by the alias M. Use the alias of the specific work area to identify a particular field. Local and static variables cannot use the alias operator; you must use the variable alone.

Example The following program opens two tables that both have a field named City and creates both private and local variables named City:

```
use CUSTOMER                      // Open in current work area
use CUSTOMER                      // Open in current work area
use VENDOR2 in select() alias VENDOR // Open in another work area with alias
private city                      // Names are not case-sensitive
city = "Peoria" // Alias not required because assignment does not assign to fields
? "No alias:", city                // Field from current table
exerciseAliasOp()
use in VENDOR                      // Close tables
use

function exerciseAliasOp()
  local city
  city = "Louisville"
  ? "Local defined, no alias:", city // Local variable
  ? "M alias:", m->city              // Private variable hidden by local
  ? "Customer:", customer->city      // Field from table
  ? "Vendor:", vendor->city          // Field from other table
```

Macro operator

Substitutes the contents of a private or public string variable during the evaluation of a statement.

Syntax &<character variable>[.]

Description Macro substitution with the & operator allows you to change the actual text of a program statement at runtime. This capabilities allows you to overcome certain syntactic and architectural limitations in *Visual* dBASE.

The mechanics of macro substitution are as follows. When compiling a statement, in a program or for immediate execution in the Command window, *Visual* dBASE looks for any single & symbols in the statement. (Double ampersands [&&] denote end-of-line comments.) If something that looks like it

could be a variable name—that is, a word made up of letters, numbers, and underscores—immediately follows the & symbol, its location is noted during compilation. If a period (.) happens to immediately follow the word, that period is considered to be a macro terminator.

When the statement is executed, *Visual* dBASE searches for a private or public variable with that name. If that variable exists, and that variable is a character variable, the contents of that variable are substituted in the statement in the place of the & symbol, the variable name, and the terminating period, if any. This is referred to as *macro substitution*. If no private or public variable with that name can be found, or if the variable is not a character variable, nothing happens; the statement is executed as-is.

Note The & character is also used as the pick character in the *text* property of some form and menu components. For example, if you use the string "&Close" to designate the letter C as the pick character, if you happen to have a private or public variable named *close*, it will be substituted.

If macro substitution occurs, one of two things can happen:

- Some commands expect certain kinds macro substitution. If the substitution is one of those cases, the command can immediately use the substituted value. For example, SET commands which expect either ON or OFF as the final word in the statement are optimized in this way.
- If the substituted value is not an expected case, or if the command or statement does not expect macro substitution, the entire statement in its new form is recompiled on-the-fly and executed.

Recompiling the statement takes a small amount of time that is negligible unless you are constantly recompiling in a loop. Also, local and static variables may be out-of-scope when a recompiled statement is executed.

You cannot use the & operator immediately after a dot operator. You also cannot have the & and dot operators on the left side of an assignment operator; that is, you cannot assign to a property that is partially resolved with macro substitution. If you do either of these, a compile-time error occurs. You can assign to a property that is completely resolved with macro substitution, or use the STORE command instead of an assignment operator.

The macro terminator (a period, the same character as the dot operator) is required if you want to abut the macro variable name with a letter, number, underscore or dot operator. Compare the following examples:

```
&f.text    // The macro variable f.text
&f.text    // The macro variable f followed by the word text
&f..text   // The macro variable f followed by the dot operator and the word text
```

Example The first example stores the value of a setting at the beginning of a function, and restores it at the end.

```
function findMatch( xArg )
  local lRet
  private cExact           // Can't be local for macro substitution
  cExact = set( "EXACT" )   // Store "ON" or "OFF" to character variable
```



```

set exact on
lRet = seek( xArg )           // Does exact match exist?
set exact &cExact             // Either "set exact ON" or "set exact OFF"
return lRet

```

The second example converts the value of a control in a form into a literal value to be used in a filter. The filter cannot refer to the control directly, because the value of the form reference varies depending on what form has focus at any given moment.

```

function setFilter_onClick()
  private cFltr
  cFltr = form.cityCombobox.value // Store string in private variable
  set filter to CITY == "&cFltr"

```

Note the use of macro substitution inside a quoted string. For example, if the value of the combobox is “Dover”, then the variable is assigned the value “Dover”. The result of the macro substitution would then be the statement:

```

set filter to CITY == "Dover"

```

The third example uses macro substitution to refer to a control by name through a variable.

```

local f
private c, g
f = new Form()
g = f // g and f both point to same form
f.button1 = new PushButton(f)
c = "1"
? g.button&c..className // Displays "PUSHBUTTON"
? f.button&c..className // Error: Variable undefined: F

```

This creates a form with a single button. Two variables refer to this form, one private variable, and one local variable. In the first macro substitution statement, the value of the variable *c* is substituted. There are two periods following the variable. The first is to terminate the macro, and the second is the actual dot operator. This results in:

```

? g.button1.className

```

The second macro substitution works the same, but the statement fails because the local variable *f* is not in scope when the statement is executed. However, this approach is actually unnecessary because you can refer to controls by name with a variable through the form’s *elements* array:

```

? f.elements[ "button" + c ].className

```

Without macro substitution, you avoid potential problems with local variables.

Finally, continuing the previous example, compare these statements that attempt to assign to a property using macro substitution:

```

local cText
cText = "OK"
g.button&c..text := cText // Has both & and . to left of assignment; won't compile
private cVar
cVar = "g.button" + c + ".text"

```

```
&cVar := cText    // Compiles, but fails at runtime; local variable out-of-scope
private cStmt
cStmt = cVar + " := cText"           // Try macro substituting entire statement
&cStmt                                // Fails; local variable out-of-scope
cStmt = cVar + [ := "]" + cText + ["] // Build entire statement with no locals
&cStmt                                // This works
g.elements[ "button" + c ].text = cText // But this is still easier to use
```

Non-operational symbols

Though they don't act upon data or hold values in themselves, non-operational symbols have their own purpose in *Visual* dBASE code and in the interpretation of programs. The following is a summary of these symbols and their usage.

String delimiters

Enclose literal strings in either:

- A set of single quote marks,
- A set of double quote marks, or
- A set of square brackets

The following example simply assigns the string "literal text" to the variable *xString*:

```
xString = "literal text"
```

To use a string delimiter in a literal string, use a different set of delimiters to delimit the string. For example:

```
? [There are three string delimiters: the ', the ",] + " and the []"
```

Note that the literal string had to be broken up into two separate strings, because all three kinds of delimiters were used.

Name/database delimiters

If the name of a variable or a field in a work area contains a space, you may enclose the name in colons, for example:

```
local :a var:
:a var: = 4
? :a var:      // Displays 4
```

Creating variables with spaces in them is strongly discouraged, but for some table types, it is not unlikely to get field names with spaces. If you create automem variables for that table, those variables will also have spaces.

However, if you're using the data access objects instead of the Xbase DML, the fields are contained in a *fields* array and are referenced by name. The field name is a character expression, so you don't have to do anything different if the field name contains a space. The colons are *not* used.

You may also use colons when designating a table in a database. The name of the database is enclosed in colons before the name of the table, in the form:

```
:database:table
```

For example:

```
use :IBLOCAL:EMPLOYEE // IBLOCAL is sample Interbase database
```

Comment symbols

Two forward slashes (//, no space between them) indicate that all text following the slashes (until the next carriage return) is a comment. Comments let you provide reference information and notes describing your code:

```
x = 4 * y // multiply the value of y by four and assign the result to variable x
```

Two ampersands (&&) can also be used for an end-of-line comment, but they are usually seen in older code.

If an asterisk (*) is the first character in a statement, the entire line is considered a comment.

A pair of single forward slashes with “inside” asterisks (/* */) encloses a block comment that can be used for a multi-line comment block:

```
/* this is the first line of a comment block
this is more of the comment
this is the last line of the comment block */
```

You can also use the pair for a comment in the middle of a statement:

```
x = 1000000 /* a million! */ * y
```

Comment blocks cannot be nested. This example shows improper usage:

```
/* this is the first line of a comment block
this is more of the the same /* this nested comment will cause problems*/
this is the last line of the comment block */
```

After the opening block marker, *Visual* dBASE ends the comment at the next closing block marker it finds, which means that only the section of the comment from “this is the first line” to the word “problems” will be interpreted as a comment. The unenclosed remainder of the block will generate an error.

Statement separator, line continuation

There is normally one statement per line in a program file. Use the semicolon to either:

- Combine multiple statements on a single line, or
- Create a multi-line statement

For example, a DO...UNTIL loop usually takes more than two lines: one for the DO, one for the UNTIL condition, and one or more lines in the loop body. But

suppose all you want to do is loop until the condition is *true*; you can combine them using the semicolon as the statement separator:

```
do ; until rlock() // Wait for record lock
```

Long statements are easier to read if break them up into multiple lines. Use the semicolon as the last non-comment character on the line to indicate that the statement continues on the next line. When the program is compiled, the comments are stripped; then any line that ends with a semicolon is tacked onto the beginning of the next line. For example, the program:

```
? "abc" + ; // A comment
  "def" + ;
    ghi
```

is compiled as

```
? "abc" + "def" + ghi
```

on line 3 of the program file. Note that the spaces before the semicolons and the spaces used to indent the code are not stripped. If an error occurs because there is no variable named *ghi*, the error will be reported on line 3.

Codeblock, literal date, literal array symbol

Braces ({ }) enclose codeblocks, literal dates, and literal array elements. They must always be paired. The following examples show how braces may be used in *Visual* dBASE code.

Literal dates are interpreted according to the current settings of SET DATE and SET EPOCH:

```
dMoon = {07/20/69} // July 20, 1969 if SET DATE is MDY and SET EPOCH is 1900
```

To enclose arrays

```
a = {1,2,3}
? a[2] // displays 2
```

To assign a statement codeblock to an object's event handling property

```
form.onOpen = {;msgbox("Warning: You are about to enter a restricted area.")}
```

To assign an expression codeblock to a variable, and pass parameters to it

```
c = {|x| x*9}
? c(4) // returns 36

// or

q = {|n| {"1st","2nd","3rd"}[n]}
? q(2) // displays "2nd"
```

To assign an expression codeblock to a variable, without passing parameters

```
c = {|| 4*9} // pipes (||) must be included in an expression codeblock,
// even if a parameter is not being passed
? c() // returns 36
```

Preprocessor directive symbol

The number sign (#) marks preprocessor directives, which provide instructions to the *Visual* dBASE compiler. Preprocessor directives may be used in programs only.

Use directives in your *Visual* dBASE code to perform such compile-time actions as replacing text throughout your program, perform conditional compilations, include other source files, or specify compiler options.

The symbol must be the non-blank first character on a line, followed by the directive (with no space), followed by any conditions or parameters for the directive.

For example, you might use this statement:

```
#include "IDENT.H"
```

to include a source file named IDENT.H (the “H” extension is generally used to identify the file as a “header” file) in the compilation. The included file might contain its own directives, such as constant definitions:

```
//file IDENT.H: constant definitions for MYPROG
#define COMPANY_NAME  "Nobody's Business"
#define NUM_EMPLOYEES 1
#define COUNTRY       "Liechtenstein"
```

For a complete listing of all *Visual* dBASE preprocessor directives, along with syntax and examples for each, see Chapter 16.

Core language

This chapter describes the core features of the *Visual* dBASE programming language, primarily:

- Structural elements
- Function linking/loading
- Program flow
- Variable scoping
- Global properties and methods

Basic understanding of programming concepts such as loops and variables is assumed.

class Exception

An object that describes an exception condition.

Syntax [*<oRef>* =] new Exception()

<oRef> A variable or property in which to store a reference to the newly created Exception object.

Properties The following table lists the properties of the Exception class. (No events or methods are associated with this class.)

| Property | Default | Description |
|------------------|-----------|--|
| <i>className</i> | EXCEPTION | Identifies the object as an instance of the Exception class |
| <i>code</i> | 0 | A numeric code to identify the type of exception |
| <i>filename</i> | | The name of the file in which a system-generated exception occurs. |

| Property | Default | Description |
|----------------|---------|---|
| <i>lineNo</i> | 0 | The line number in the file in which a system-generated exception occurs. |
| <i>message</i> | | Text to describe the exception |

Description

| Event | Parameters | Description |
|-------|------------|-------------|
| none | | |

An Exception object is automatically generated by *Visual* dBASE whenever an error occurs. The object's properties contain information about the error.

You can also create an Exception object manually, which you can fill with information and THROW to manage execution or to jump out of deeply nested statements.

You may subclass the Exception class to create your own custom exception objects. A TRY block may be followed by multiple CATCH blocks, each one looking for a different exception class.

Example Suppose you are using exceptions to manage execution in a deeply nested set of conditional statements and loops. You create your own exception class:

```
class JumpException of Exception
endclass
```

Then in the code, you create the JumpException object and THROW it if needed:

```
try
  local j
  j = new JumpException()
  // Lots of nested code
  :
  if lItsNoGood
    throw j // Deep in the code, you want out
  endif
  :
catch ( JumpException e )
  // Do nothing; JumpException is OK
catch ( Exception e )
  // Normal error
  logError( new Date(), e.message ) // Record error message
  // and continue
endtry
```

If there is a normal error, the second CATCH block saves it to a log file, using a function you wrote, and execution continues.

See also THROW, TRY...ENDTRY

class Object

An empty object.

Syntax [*<oRef>* =] new Object()

<oRef> A variable or property in which to store a reference to the newly created object.

Properties An object of the Object class has no initial properties, events, or methods.

Description Use the Object class to create your own simple objects. Once the new object is created, you may add properties and methods through assignment. You cannot add events.

This technique of adding properties and methods on-the-fly is known as *dynamic subclassing*. In *Visual dBASE*, dynamic subclassing supplements *formal subclassing*, which is achieved through CLASS definitions.

The Object class is the only class in *Visual dBASE* that does not have the read-only *className* property.

Example The following statements create a simple object with a few properties—some referenced by name and some referenced by number—and a codeblock as a method.

```
o = new Object()
o.title = "Summer"
o[ 2000 ] = "Sydney"
o[ 1996 ] = "Atlanta"
o.cityInYear = {|y| this[ y ]}
? o.cityInYear( 2000 ) // Displays "Sydney"
```

See also CLASS

ARGCOUNT()

Returns the number of parameters passed to a routine.

Syntax ARGCOUNT()

Description Use ARGCOUNT() to determine how many parameters, or arguments, have been passed to a routine. You may alter the behavior of the routine based on the number of parameters. If there are fewer parameters than expected, you may provide default values.

ARGCOUNT() returns 0 if no parameters are passed.

The function PCOUNT() is identical to ARGCOUNT(). Neither function recognizes parameters passed to codeblocks. If called within a codeblock, the

ARGVECTOR()

function will return the parameter information for the currently executing FUNCTION or PROCEDURE.

Example The following function returns someone's age. The first required parameter is the birthdate. The second optional parameter is the date to calculate the age. If the second parameter is not specified, the current date is used.

```
function age( dBirth, dCheck )
  if argcount() < 2
    dCheck = date()
  endif
  return floor( ( val( dtos( dCheck )) - val( dtos( dBirth )) ) / 10000 )
```

See Also ARGVECTOR(), DO, FUNCTION, PARAMETERS

ARGVECTOR()

Returns the specified parameter passed to a routine.

Syntax ARGVECTOR(<parameter expN>)

<parameter expN> The number of the parameter to return. 1 returns the first parameter, 2 returns the second parameter, etc.

Description Use ARGVECTOR() to get a copy of the value of a parameter passed to a routine. Because it is a copy, there is no danger of modifying the parameter, even if it was a variable that was passed by reference. For more information on parameter passing, see PARAMETERS.

ARGVECTOR() can be used in a routine that receives a variable number of parameters, where declaring the parameters would be difficult.

Example The following function returns the mean average of all the parameters passed to it, skipping any with a *null* value:

```
function mean()
  local nRet, nArg, nCnt
  nTot = 0
  nCnt = 0
  for nArg = 1 to argcount()
    if argvector( nArg ) # null
      nTot += argvector( nArg )
      nCnt++
    endif
  endfor
  return nTot / nCnt
```

See Also ARGCOUNT(), DO, FUNCTION, PARAMETERS

CASE

Designates a block of code in a DO CASE block.

Description See DO CASE for details.

CATCH

Designates a block of code to execute if an exception occurs inside a TRY block.

Description See TRY...ENDTRY for details.

CLASS

A class declaration including constructor code, which typically creates member properties, and class methods.

Syntax

```

CLASS <class name>[(<parameters>)]
    [OF <superclass name>(<parameters>)]
    [CUSTOM]
    [FROM <filename expC> ]
    [PROTECT <propertyList>]
    [<constructor code>]
    [<methods>]
ENDCLASS

```

<class name> The name of the class.

OF <superclass name> Indicates that the class is a derived class that inherits the properties defined in the superclass. The superclass constructor is called before the <constructor code> in the current CLASS is called, which means that any properties created in the superclass are inherited by the class.

<parameters> Optional parameters to pass to the class, and through to the superclass.

CUSTOM Identifies the class as a custom component class, so that its predefined properties are not streamed out by the visual design tools.

FROM <filename> <filename> specifies the file containing the definition code for the <superclass>, if the <superclass> is not defined in the same file as the class.

PROTECT <propertyList> <propertyList> is a list of properties and/or methods of the class which are to be accessible only by other members of the class, and by classes derived from the class.

<constructor code> The code that is called when a new instance of the class is created with the NEW operator or a DEFINE statement. The constructor consists of all the code at the top of the class declaration up to the first method.

<methods> Any number of functions designed for the class.

ENDCLASS A required keyword that marks the end of the CLASS structure.

Description Use CLASS to create a new class.

A class is a specification, or template, for a type of object. *Visual* dBASE provides many stock classes, such as Form and Query; for example, when you create a form, you are creating a new Form object that has the standard properties and methods from the Form class. However, when you declare a class with CLASS, *you* specify the properties and methods that objects derived from the new class will have.

A CLASS declaration formalizes the creation of an object and its methods. Although you can always add properties to an object and assign methods dynamically, a CLASS simplifies the task and allows you to build a clear class hierarchy.

Another benefit is polymorphism. Every FUNCTION (or PROCEDURE) defined in the CLASS becomes a method of the class. An object of that class automatically has a property with the same name as each FUNCTION that contains a reference to that FUNCTION. Because a method is part of the CLASS, different functions may use the same name as long as they are methods of different classes. For example, you can have multiple *copy()* functions in different classes, with each one applying to objects of that class. Without classes, you would have to name the functions differently even if they performed the same task conceptually.

Before the first statement in the constructor is executed, if the CLASS extends another class, the constructor for that superclass has already been executed, so the object contains all the superclass properties. Any properties that refer to methods, as described in the previous paragraph, are assigned. This means that if the CLASS contains a method with the same name as a method in a superclass, the method in the CLASS overrides the method in the superclass. The CLASS constructor, if any, then executes.

In the constructor, the variable *this* refers to the object being created. Typically, the constructor creates properties by assigning them to *this* with dot notation. However, the constructor may contain any code at all, except another CLASS—you can't nest classes—or a FUNCTION, since that FUNCTION would become a method of the class and indicate the end of the constructor.

Properties and methods can be protected to prevent the user of the class from reading or changing the protected property values, or calling the protected methods from outside of the class.

See also class Object, *className*, FUNCTION

className

Identifies of which class the object is a member.

Property of All classes except Object.

Description The *className* property identifies the class constructor that originally created the object. Although you may dynamically subclass the object by adding new properties, the *className* property does not change.

The *className* property is read-only.

See also CLASS, FINDINSTANCE()

CLEAR MEMORY

Clears all user-defined memory variables.

Syntax CLEAR MEMORY

Description Use CLEAR MEMORY to release all memory variables (except system memory variables), including those declared PUBLIC and STATIC and those initialized in higher-level routines. CLEAR MEMORY has no effect on system memory variables.

Note CLEAR MEMORY does not explicitly release objects. However, if the only reference to an object is in a memory variable, releasing the variable with CLEAR MEMORY will in turn release the object.

Issuing RELEASE ALL in the Command window has the same effect as CLEAR MEMORY. However, issuing RELEASE ALL in a program clears only memory variables created at the same program level as the RELEASE ALL statement, and has no effect on higher-level, public, or static variables. CLEAR MEMORY, whether issued in a program or in the Command window, always has the same effect, releasing all variables.

To clear only selected memory variables, use RELEASE.

See Also RELEASE

CLEAR PROGRAM

Clears from memory all program files that aren't currently executing and aren't currently open with SET PROCEDURE or SET LIBRARY.

Syntax CLEAR PROGRAM

Description Program files are loaded into memory when they are executed with DO, and when they are loaded as library or procedure files with SET LIBRARY and SET PROCEDURE. When *Visual* dBASE is done with the program—the execution is complete, or the file is unloaded—the program file is not automatically cleared from memory. This allows these files to be quickly reloaded without having to reread them from disk. *Visual* dBASE's internal dynamic memory management will clear these files if it needs more memory; for example, when you create a very large array.

You may use CLEAR PROGRAM to force the clearing of all inactive program (object code) files from memory. The command doesn't clear files that are currently executing or files that are currently open with SET PROCEDURE or SET LIBRARY. However, if you close a file (for example, with CLOSE PROCEDURE), a subsequent CLEAR PROGRAM clears the closed file from memory.

CLEAR PROGRAM is rarely used in a deployed application. Because of the event-driven nature of *Visual* dBASE, program files must remain open to handle events; these files are not affected by CLEAR PROGRAM anyway. Also, the amount of memory used by dormant program files is small compared to the total amount of memory available. You are more likely to use CLEAR PROGRAM during development, for example to ensure that you are running the latest version of a program file, and not one that is stuck in memory.

See Also DO, CLEAR MEMORY, CLOSE PROCEDURE, SET LIBRARY, SET PROCEDURE

CLOSE PROCEDURE

Closes one or more procedure files, preventing further access and execution of its functions, classes, and methods.

Syntax CLOSE PROCEDURE [<filename list>]

<filename list> A list of procedure files you want to close, separated by commas. If you specify a file without including its extension, *Visual* dBASE assumes PRG. If you omit <filename list>, all procedure files are closed, regardless of their load count.

Description CLOSE PROCEDURE reduces the load count of each specified program file by one. If that reduces its load count to zero, then that program file is closed, and its memory is marked as available for reuse.

When you specify more than one file in <filename list>, they are processed in reverse order, from right to left. If a specified file is not open as a procedure file, an error occurs, and no more files in the list are processed.

Closing a program file does not automatically remove the file from memory. If a request is made to open that program file, and the file is still in memory and its source code has not been updated, it will be reopened without having to reread

the file from disk. Use CLEAR MEMORY to release a closed program file from memory.

In a deployed application, it is not unusual to open program files as procedure files and never close them. Because of the event-driven nature of *Visual* dBASE, program files must remain open to respond to events. The memory used by a procedure file is small in comparison to the amount of system memory.

See SET PROCEDURE for a description of the reference count system used to manage procedure files. You may issue SET PROCEDURE TO or CLOSE PROCEDURE with no <filename list> to close all open procedure files, regardless of their load count.

See Also CLEAR PROGRAM, SET LIBRARY, SET PROCEDURE

DEFINE

Creates an object from a class.

Syntax DEFINE <class name> <object name>
 [OF <container object>]
 [FROM <row, col> TO <row, col> > | <AT <row, col>]
 [PROPERTY <stock property list>]
 [CUSTOM <custom property list>]

<class name> The class of the object to create.

<object name> The identifier for the object you create. <object name> will become an object reference variable, or a named property of the container if a <container object> is specified.

OF <container object> Identifies the object that contains the object you define.

FROM <row>, <col> TO <row>, <col> | AT <row>, <col> Specifies the initial location and size of the object within its container. FROM and TO specify the upper left and lower right coordinates of the object, respectively. AT specifies the position of the upper left corner.

PROPERTY <stock property list> Specifies values you assign to the built-in properties of the object.

CUSTOM <custom property list> Specifies new properties you create for the object and the values you assign to them.

Description Use DEFINE to create an object in memory. DEFINE provides an alternate, shorthand syntax for creating objects that directly maps to using the NEW operator. The equivalence depends on whether the object created with DEFINE is created inside a container object. With no container,

define <class name> <object name>

is equivalent to:

```
<object name> = new <class name>()
```

With a container,

```
define <class name> <object name> of <container object>
```

is equivalent to:

```
new <class name>( <container object>, "<object name>" )
```

where *<object name>* becomes an all-uppercase string containing the specified name. These two parameters, the container object reference and the object name, are the two properties expected by the class constructors for all stock control classes such as `PushButton` and `Entryfield`. For example, these two sets of statements are functionally identical (and you can use the first statement in one set with the second statement of the other set):

```
define Form myForm
define PushButton cancelButton of myForm

myForm = new Form()
new PushButton( myForm, "CANCELBUTTON" )
```

The FROM or AT clause of the DEFINE command provide a way to specify the *top* and *left* properties of an object, and the TO coordinates are used to calculate the object's *height* and *width*.

The PROPERTY clause allows assignment to existing properties only.

Attempting to assign a value to a non-existent property generates an error at runtime. This will catch spelling errors in property names, when you want to assign to an existing property; it prevents the creation of a new property with the misspelled name. Using the assignment-only (`:=`) operator has the same effect when assigning directly to a property in a separate assignment statement. In contrast, the CUSTOM clause will create the named property if it doesn't already exist.

While the DEFINE syntax offers some amenities, it is not as flexible as using the NEW operator and a WITH block. In particular, with DEFINE you cannot pass any parameters to the class constructor other than the two properties used for control containership, and you cannot assign values to the elements of properties that are arrays.

See Also CLASS, REDEFINE, WITH

DO

Runs a program or function.

Syntax DO <filename> | ? | <filename skeleton> |
 <function name>
 [WITH <parameter list>]

<filename> | ? | <filename skeleton> The program file to execute. The ? and <filename skeleton> options display a dialog box from which you can select a file.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the *search path* in *search order*. See "Search path and order" later in this section for more information.

If you specify a file without including its extension, *Visual* dBASE assumes a .PRO extension (a compiled object file). If *Visual* dBASE can't find a .PRO file, it looks for a .PRG file (a source file), which, if found, it compiles. By default, *Visual* dBASE creates the .PRO in the same directory as the .PRG, which might not be the current directory.

<function name> The function name in an open program file to execute. The function must be in the program file containing the DO command that calls it, or in a separate open file on the search path. The search path is described later in this section.

WITH <parameter list> Specifies memory variable values, field values, or any other valid expressions to pass as parameters to the program or function. See the description of PARAMETERS for information on parameter passing.

Description Use DO to run program files from the Command window or to run other programs from a program. If you enter DO in the Command window, control returns to the Command window when the program or function ends. If you use DO in a program to execute another program, control returns to the program line following the DO statement when the program ends.

Although you may use DO to execute functions, common style dictates the use of the call operator (the parentheses) when calling functions, and the DO command when running a program file. The DO command supports the use of a file path and extension, and the ? and <filename skeleton> options. The call operator supports calling a function by name only. In the not-recommended situation where you have a program file that has the same name as a function loaded into memory, the DO command will execute the program file, and the call operator will execute the loaded function. Other than these differences, the two calling methods behave the same, and follow the same search rules described later in this section.

You may nest routines; that is, one routine may call another routine, which may call another routine, and so on. This series of routines, in the order in which they are called, is referred to as the *call chain*.

When *Visual* dBASE executes or loads a program file, it will automatically compile the program file into object code when either:

- There is no object code file, or
- SET DEVELOPMENT is ON, and program file is newer than the object code file (the source code file's last update date and time is later than the object code file's)

When *Visual* dBASE encounters a function call in a program file, it looks in that file for a FUNCTION or PROCEDURE of the specified name. If the current program file contains a FUNCTION and a PROCEDURE with the same name, *Visual* dBASE executes the first one declared. If *Visual* dBASE doesn't find a

FUNCTION or PROCEDURE definition of the specified name in the same program file, it looks for a program file, FUNCTION, or PROCEDURE of the specified name on the *search path* in *search order*.

Search path and order If the name you specify with DO doesn't include a path or a file-name extension, it can be a file, FUNCTION, or PROCEDURE name. To resolve the ambiguity, *Visual* dBASE searches for the name in specific places (the search path) in a specific order (the search order) and runs the first program or function of the specified name that it finds. The search path and order *Visual* dBASE uses is as follows:

- 1 The executing program's object file (.PRO)
- 2 Other open object files (.PRO) in the call chain, in most recently opened order
- 3 The file specified by SYSPROC = <filename> in VDB.INI
- 4 Any files opened with SET PROCEDURE, SET PROCEDURE...ADDITIVE, or SET LIBRARY statements, in the order in which they were opened
- 5 The object file (.PRO) with the specified name in the search path
- 6 The program file (.PRG) with the specified name in the search path, which *Visual* dBASE automatically compiles

The search path is controlled with the SET PATH command. It is *not* used when you are running a compiled EXE (a deployed application)—all program files must be linked into the executable. All path information is lost during linking and ignored during execution, which means that you cannot have more than one file with the same name, even if they originally came from different directories.

Because program files must be compiled into object code to be linked into a compiled EXE, the last search step, #6, does not apply when running a compiled EXE.

See Also CLEAR PROGRAM, COMPILE, RETURN, SET DEVELOPMENT, SET ESCAPE, SET LIBRARY, SET PATH, SET PROCEDURE

DO CASE

Conditionally processes statements by evaluating one or more conditions and executing the statements following the first condition that evaluates to true.

Syntax

```
DO CASE
CASE <condition expL 1>
    <statements>
[CASE <condition expL 2>
    <statements>...]
[OTHERWISE
    <statements>]
ENDCASE
```

CASE <condition expl> If the condition is true, executes the set of commands between CASE and the next CASE, OTHERWISE, or ENDCASE command, and then transfers control to the line following ENDCASE. If the condition is false, control transfers to the next CASE, OTHERWISE, or ENDCASE command.

<statements> Zero or more statements to execute if the preceding CASE statement evaluates to *true*.

OTHERWISE Executes a set of statements if all the CASE statements evaluate to *false*.

ENDCASE A required keyword that marks the end of the DO CASE structure.

Description

DO CASE is similar to IF...ELSE...ENDIF. As with IF conditions, *Visual* dBASE evaluates DO CASE conditions in the order they're listed in the structure. DO CASE acts on the first true condition in the structure, even if several apply. In situations where you want only the first true instance to be processed, use DO CASE instead of a series of IF commands.

Also, use DO CASE when you want to program a number of exceptions to a condition. The CASE <condition> statements can represent the exceptions, and the OTHERWISE statement the remaining situation.

Starting with the first CASE condition, *Visual* dBASE does the following.

- Evaluates each CASE condition until it encounters one that's *true*
- Executes the statements between the first true CASE statement and the next CASE, OTHERWISE, or ENDCASE (if any)
- Exits the DO CASE structure without evaluating subsequent CASE conditions
- Moves program control to the first line after the ENDCASE command

If none of the conditions are true, *Visual* dBASE executes the statements under OTHERWISE if it's included. If no OTHERWISE statement exists, *Visual* dBASE exits the structure without executing any statements and transfers program control to the first line after the ENDCASE command.

DO CASE is functionally identical to an IF...ELSEIF...ENDIF structure. Both specify a series of conditions and an optional fallback (OTHERWISE and ELSE) if none of the conditions are *true*. Common style dictates the use of DO CASE when the conditions are dependent on the same variable, for example what key was pressed, while IF...ELSEIF...ENDIF is used when the conditions are not directly related. In addition, DO CASE usually involves more indenting of code.

Example

The following is a *key* event handler for a custom entryfield that handles dates. It recognizes special keystrokes to move the date forward or backward one day, to the first and last day of the month, and so forth:

```
function key( nChar, nPosition )
    local c1
    c1 = upper( chr( nChar ) )
    do case
        case c1 == "T"                // Today
```

```

        this.value := date()
    case c1 == "-" or c1 == "_" // Next day
        this.value--
    case c1 == "+" or c1 == "=" // Previous day
        this.value++
    case c1 == "M" // First day of the month
        this.value := FDoM( iif( this.lastKey == "M", --this.value, this.value ) )
    case c1 == "H" // Last day of the month
        this.value := LDoM( iif( this.lastKey == "H", ++this.value, this.value ) )
    case c1 == "Y" // First day of the year
        this.value := FDoY( iif( this.lastKey == "Y", --this.value, this.value ) )
    case c1 == "R" // Last day of the year
        this.value := LDoY( iif( this.lastKey == "R", ++this.value, this.value ) )
    otherwise
        this.lastKey := null // Clear stored keystroke
        return true // Handle key normally
    endcase
    this.lastKey := c1 // Store as property for comparison next time
    return false // Ignore key

```

The functions FDoM(), LDoM(), FDoY(), and LDoY() are defined elsewhere.

See Also IF, IIF()

DO WHILE

Executes the statements between DO WHILE and ENDDO while a specified condition is *true*.

Syntax DO WHILE <condition expl>
 [<statements>]
 ENDDO

<condition expl> A logical expression that is evaluated before each iteration of the loop to determine whether the iteration should occur. If it evaluates to *true*, the statements are executed. Once it evaluates to *false*, the loop is terminated and execution continues with the statement following the ENDDO.

<statements> Zero or more statements executed in each iteration of the loop.

ENDDO A required keyword that marks the end of the DO WHILE loop.

Description Use a DO WHILE loop to repeat a statement or block of statements while a condition is *true*. If the condition is initially *false*, the statements are never executed.

You may also exit the loop with EXIT, or restart the loop with LOOP.

Example The following loop deletes all the orders for a particular customer, using the customer ID number to find their orders in the Order table:

```
function deleteAllOrders
```

```
do while form.orders1.rowset.findKey( form.custID.value )
    form.orders1.rowset.delete()
enddo
```

Note that if there are no orders in the table initially, the DO WHILE condition will fail, and nothing will happen.

See also DO...UNTIL, EXIT, FOR...ENDFOR, LOOP

DO...UNTIL

Executes the statements between DO and UNTIL at least once while a specified condition is false.

Syntax DO
 [<statements>]
 UNTIL <condition expL>

<statements> Zero or more statements executed in each iteration of the loop.

UNTIL <condition expL> The statement that marks the end of the DO...UNTIL loop. <condition expL> is a logical expression that is evaluated after each iteration of the loop to determine whether the iteration should occur again. If it evaluates to *false*, the statements are executed. Once it evaluates to *true*, the loop is terminated and execution continues with the statement following the UNTIL.

Description Use a DO...UNTIL loop to repeat a block of statements until a condition is *true* (in other words, while the condition is *false*). Because the condition is evaluated at the end of the loop, a DO...UNTIL loop always executes at least once, even when the condition is initially *true*.

You may also exit the loop with EXIT, or restart the loop with LOOP.

DO...UNTIL is rarely used. In most condition-based loops, you don't want to execute the loop at all if the condition is initially invalid. DO WHILE loops are much more common, because they check the condition before they begin.

In a DO WHILE loop, the condition fails—that is, the loop should not be executed—when it evaluates to *false*; in a DO...UNTIL loop, the condition fails when it evaluates to *true*. This is simply the result of the wording of the looping commands. You can easily reverse any logical condition by using the logical NOT operator or the opposite comparison operator (for example, less than instead of greater than or equal, or not equal instead of equal).

Example The first example shows a loop that goes through all the checkboxes on a form and sets their *value* to *false*. An object reference to the form's first control is assigned to a variable, and the reference is updated at the end of the loop to point to the next control in the tab order.

```
local oCtrl
oCtrl = form.first
do
```

```

    if oCtrl.className == "CHECKBOX"
        oCtrl.value := false
    endif
    oCtrl := oCtrl.before
until oCtrl == form.first

```

Because the DO...UNTIL checks the condition at the end of the loop, after the object reference has been updated, you can simply test if the reference has looped back to the beginning. To use the same test with a DO WHILE loop, you would have to maintain an extra flag to allow the loop to proceed the first time through.

The next example shows a basic loop that traverses all the rows in a rowset, referenced by the variable *r*:

```

if r.first()
do
    // Something to do to each row
until not r.next()
endif

```

To traverse the rowset, you must start at the first row. The *first()* method attempts to reposition the row cursor to the first row in the rowset, returning *true* to indicate success. It would return *false* if there are no rows in the rowset—no rows at all, or no rows that match any active filter conditions—in which case the IF fails and the DO...UNTIL loop is not executed at all. If it returns *true*, then there must be at least one row, and the DO...UNTIL loop body is executed.

After the loop body is executed, the rowset's *next()* method is called. It returns *true* unless it reaches the end-of-set. As long as it returns *true*, the logical NOT operator reverses the logical condition so that the UNTIL condition evaluates to *false*, and the loop continues. When it reaches the end-of-set, *next()* returns *false*, which gets reversed to *true*, satisfying the UNTIL condition and terminating the loop.

Compare the DO...UNTIL loop with the equivalent structure using DO WHILE:

```

r.first()
do while not r.endOfSet
    // Something to do to each row
    r.next
enddo

```

The rowset's *endOfSet* property is *true* if the rowset is at the end-of-set. The return value from the *first()* method is not checked, because the *endOfSet* property is checked at the beginning of the DO WHILE loop. If *first()* fails, it leaves the rowset cursor at the end-of-set. The return value of *next()* is also not checked, for the same reason. However, this loop is slightly less efficient because it goes through the extra step of checking the *endOfSet* property instead of simply using the return value of *next()*, which must be called to move to the next row.

This next example may look a bit odd:

```

do
until form.rowset.rlock()

```

but it simply retries the `rlock()` until it is successful. Note that the loop body is empty. You may want put a comment in the loop so you won't have to think about it in the future:

```
do
    // Wait for lock
until form.rowset.rlock()
```

or you can use the semicolon to put two statements on the same line:

```
do; until form.rowset.rlock()
```

See Also DO WHILE, EXIT, FOR...ENDFOR, LOOP

ELSE

Designates an alternate statement to execute if the condition in an IF statement is *false*.

Description See IF for details.

ELSEIF

Designates an alternate condition to test if the condition in an IF statement is *false*.

Description See IF for details.

EMPTY()

Returns *true* if a specified expression is empty.

Syntax EMPTY(<exp>)

<exp> An expression of any type.

Description Use EMPTY() to determine if an expression is empty. The definition of empty depends on the type of the expression:

| Expression type | Empty if value is |
|------------------|--|
| Numeric | 0 (zero) |
| String | empty string ("") or a string of just spaces (" ") |
| Date | blank date ({ / / }) |
| Logical | <i>false</i> |
| Null | <i>null</i> is always considered empty |
| Object reference | Reference points to object that has been released |

Note that event properties that have not been assigned handlers have a value of *null*, and are therefore considered empty. In contrast, an object reference pointing to an object that has been released is not *null*; you must use `EMPTY()`.

`EMPTY()` is similar to `ISBLANK()`. However, `ISBLANK()` is intended to test field values; it differentiates between zero and blank values in numeric fields, while `EMPTY()` does not. `EMPTY()` understands null values and object references, while `ISBLANK()` does not. For more information, see `ISBLANK()`.

See Also `ISBLANK()` (page 12-57), `TYPE()`

EXIT

Immediately terminates the current loop. Execution continues with the statement after the loop.

Syntax `EXIT`

Description Normally, all of the statements in the loop are executed in each iteration of the loop; in other words, the loop always exits after the last statement in the loop. Use `EXIT` to exit a loop from the middle of a loop, due to some extra or abnormal condition.

In most cases, you don't have to resort to using `EXIT`; you can code the condition that controls the loop to handle the extra condition. The condition is tested between loop iterations, after the last statement, but that usually means that there are some statements that should not be executed because of this condition. Those statements would have to be conditionalized out with an `IF` statement. Therefore, often it's simpler to `EXIT` out of a loop immediately once the condition occurs.

Example The following function counts the number of words in a string by counting spaces between words. Multiple spaces between two words are counted as a single space and therefore a single word:

```
function wordCount( cArg )
    local nRet, cRemain, nPos
    nRet    = 0
    cRemain = ltrim( trim( cArg ) )
    do while " " # cRemain
        nRet++
        nPos = at( " ", cRemain )
        if nPos == 0
            exit
        else
            cRemain := ltrim( substr( cRemain, nPos ) )
        endif
    enddo
    return nRet
```


The condition in the DO WHILE loop is really needed only once, the first time the loop is entered. It makes sure that there is some text to search through. If the argument is an empty string or all spaces, the loop is not executed and the word count is zero. After the first loop, it is used simply to keep the loop going, since there would always be text to check.

The loop is terminated when there are no more spaces in the string. This is determined by the return value of the AT() function. Because the position returned is out of range for the SUBSTR() function, it should be called if there are no more spaces in the string. By using EXIT, the loop is immediately terminated once no more spaces are found. Execution continues with the RETURN statement following the DO WHILE loop.

See also DO WHILE, DO...UNTIL, LOOP, FOR...ENDFOR

FINALLY

Designates a block of code that always executes after a TRY block, even if an exception occurs.

Description See TRY...ENDTRY for details.

FINDINSTANCE()

Returns an object of the specified class from the object heap.

Syntax FINDINSTANCE(<classname expC> [, <previous oRef>])

<classname expC> The name of the class you want to find an instance of. <classname expC> is not case-sensitive.

<previous oRef> When omitted, FINDINSTANCE() returns the first instance of the specified class. Otherwise, it returns the instance following <previous oRef> in the object heap.

Description Use FINDINSTANCE() to find any instance of a particular class, or to find all instances of a class in the object heap.

Objects are stored in the object heap in no predefined order. Creating a new instance of a class or destroying an instance may reorder all other instances of that class. A newly created object is not necessarily last in the heap.

Sometimes you will want to make sure there is only one instance of a class, and reuse that instance; a particular toolbar is the prime example. To see if there is an instance of that class, call FINDINSTANCE() with the class name only. If the return value is *null*, there is no instance of that class in memory.

Other times, you may want to iterate through all instances of a class to perform an action. For example, you may want to close all data entry forms, which are

all instances of the same class. Call FINDINSTANCE() with the class name only to find the first instance of the class. Then call FINDINSTANCE() in a loop with the class name and the object reference to get the next instance in the object heap. When FINDINSTANCE() returns *null*, there are no more instances.

Example The first example checks if there is already an instance of the the EditToolbar class. If not, one is created.

```
function attachEditToolbar( formObj )
  local t
  t = findinstance( "EditToolbar" )
  if empty( t )                                // If null, no instance exists, so
    t = new EditToolbar()                      // Create one (defined in this file)
    set procedure to program(1) additive       // Load this file as procedure file
  endif
  t.attach( formObj )
```

The second example finds all instances of the OrderForm class and closes them.

```
function closeAllOrders()
  local f
  f = findinstance( "OrderForm" )
  do while not empty( f )
    f.close()
    f := findinstance( "OrderForm", f )
  enddo
```

See also *className*

FOR...ENDFOR

Executes the statements between FOR and ENDFOR the number of times indicated by the FOR statement.

Syntax FOR <memvar> = <start expN> TO <end expN> [STEP <step expN>]
 [<statements>]
 ENDFOR | NEXT

<memvar> The *loop counter*, a memory variable that's incremented or decremented and then tested each time through the loop.

<start expN> The initial value of <memvar>.

<end expN> The final allowed value of <memvar>.

STEP <step expN> Defines a step size (<step expN>) by which Visual dBASE increments or decrements <memvar> each time the loop executes. The default step size is 1.

When <step expN> is positive, Visual dBASE increments <memvar> until it is greater than <end expN>. When <step expN> is negative, Visual dBASE decrements <memvar> until it is less than <end expN>.

<statements> Zero or more statements executed in each iteration of the loop.

ENDFOR | NEXT A required keyword that marks the end of the FOR loop. You may use either ENDFOR (more dBASE-ish) or NEXT.

Description Use FOR...ENDFOR to execute a block of statements a specified number of times. When *Visual dBASE* first encounters a FOR loop, it sets *<memvar>* to *<start expN>*, and reads the values for *<end expN>* and *<step expN>*. (If *<end expN>* or *<step expN>* are variables and are changed inside the loop, the loop will not see the change and the original values will still be used to control the loop.)

The loop counter is checked at the beginning of each iteration of the loop, including the first iteration. If *<memvar>* evaluates to a number greater than *<end expN>* (or less than *<end expN>* if *<step expN>* is negative), *Visual dBASE* exits the FOR loop and executes the line following ENDFOR (or NEXT). Therefore, it's possible that the loop body is not executed at all.

If *<memvar>* is in the range from *<start expN>* through *<end expN>*, the loop body is executed. After executing the statements in the loop, *<step expN>* is added to *<memvar>*, and the loop counter is checked again. The process repeats until the loop counter goes out of range.

You may also exit the loop with EXIT, or restart the loop with LOOP.

The *<memvar>* is usually used inside the loop to refer to numbered items, and continues to exist after the loop is done, just like a normal variable. If you do not want the variable to be the default private scope, you should declare the scope of the variable before the FOR loop.

Example The following event handler creates a new row, carrying over the values in the current row. The values in the current row are copied to a temporary array, a new row is created, and the values are copied from the array.

```
function newButton_onClick
  local a, n
  a = new Array()
  for n = 1 to form.rowset.fields.size
    a.add( form.rowset.fields[ n ].value )
  endfor
  form.rowset.beginAppend()
  for n = 1 to form.rowset.fields.size
    form.rowset.fields[ n ].value := a[ n ]
  endfor
```

See also DO WHILE, EXIT, LOOP

FUNCTION

Defines a function in a program file including variables to represent parameters passed to the function.

Syntax FUNCTION <function name>[([<parameter list>])]
[<statements>]

<function name> The name of the function. Although *Visual* dBASE imposes no limit to the length of function names, it recognizes only the first 32 characters.

<parameter list> Variable names to assign to data items (or *parameters*) passed to the function by the statement that called it. The variables in <parameter list> are local in scope, protecting them from modification in lower-level subroutines. For more information about the local scope, see LOCAL.

The number of variables assigned can be different from the number of parameters passed. You can use PCOUNT() to identify the number of parameters a procedure has received. You can include up to 255 variable names in <parameter list>.

<statements> Any statements that you want the function to execute. You can call functions recursively.

Description Use functions to create code modules. By putting commonly used code in a function, you can easily call it whenever needed, pass parameters to the function, and optionally return a value. You also create more modular code, which is easier to debug and maintain.

When a FUNCTION is defined inside a CLASS definition, the FUNCTION is considered a method of that CLASS. You cannot nest functions.

The keywords FUNCTION and PROCEDURE are interchangeable in *Visual* dBASE.

A single program file can contain a total of 184 functions and methods. Each class also counts as one function (for the class constructor). To access more functions simultaneously, use SET PROCEDURE...ADDITIVE. The maximum size of a function is limited to the maximum size of a program file.

When a function is called via an object, usually as a method or event handler, the variable *this* refers to the object that called the function.

Function naming restrictions Do not give a function the same name as the file in which it's contained. Statements at the beginning of the file, before any FUNCTION, PROCEDURE, or CLASS statement, are considered to be a function (not counted against the total limit) with the same name as the file. (This function is sometimes referred to as the "main" procedure in the program file.) Multiple functions with the same name do not cause an error, but the first function with that name is the only one that is ever called.

Don't give the function the same name as a built-in *Visual* dBASE function. You cannot call such a function with the DO command, and if you call the function with the call operator (parentheses), *Visual* dBASE always executes its built-in function instead.

Also do not give the function a name that matches a *Visual* dBASE command keyword. For example, you should not name a function OTHER() because that

matches the beginning of the keyword OTHERWISE. When you call the OTHER() function, the compiler will think it's the OTHERWISE keyword and will generate an error, unless you happen to be in a DO CASE block, in which case it will be treated like the OTHERWISE keyword, instead of calling the function.

These function naming restrictions do not apply to methods, because calling a method through the dot or scope resolution operator clearly indicates what is being called. However, you may run into problems calling methods inside a WITH block. See WITH for details.

Making procedures available You can include a procedure in the program file that uses it, or place it in a separate program file you access with SET PROCEDURE or SET LIBRARY. If you include a procedure in the program file that uses it, you should place it at the end of the file and group it with other procedures.

When you call a procedure, *Visual* dBASE searches for it in the *search path* in *search order*. If there is more than one procedure available with the same name, *Visual* dBASE runs the first one it finds. For this reason, avoid using the same name for more than one procedure. See the description of DO for an explanation of the search path and order *Visual* dBASE uses.

See also RETURN

IF

Conditionally executes statements by evaluating one or more conditions and executing the statements following the first condition that evaluates to *true*.

Syntax

```
IF <condition expL 1>
    [ <statements> ]
[ELSEIF <condition expL 2>
    <statements>
[ELSEIF <condition expL 3>
    <statements>...]]
[ELSE
    [ <statements> ]]
ENDIF
```

<condition expL> A logical expression that determines if the set of statements between IF and the next ELSE, ELSEIF, or ENDIF command execute. If the condition is *true*, the statements execute. If the condition is false, control passes to the next ELSE, ELSEIF, or ENDIF.

<statements> One or more statements that execute depending on the value of <condition expL>.

ELSEIF <condition expl> <statements> Specifies that when the previous IF or ELSEIF condition is false, control passes to this ELSEIF <condition expl>. As with IF, if the condition is true, only the set of statements between this ELSEIF and the next ELSEIF, ELSE, or ENDIF execute. If the condition is false, control passes to the next ELSEIF, ELSE, or ENDIF.

You can enter this option as either ELSEIF or ELSE IF. The ellipsis (...) in the syntax statement indicates that you can have multiple ELSEIF statements.

ELSE <statements> Specifies statements to execute if all previous conditions are *false*.

ENDIF A required keyword that marks the end of the IF structure.

Description Use IF to evaluate one or more conditions and execute only the set of statements following the first condition that evaluates to *true*. For the first true condition, *Visual* dBASE executes the statements between that program line and the next ELSEIF, ELSE, or ENDIF, then skips everything else in the IF structure and executes the program line following ENDIF. If no condition is true and an associated ELSE command exists, *Visual* dBASE executes the set of statements after ELSE and then executes the program line following ENDIF.

Use IF...ENDIF to test one condition and IF...ELSEIF...ENDIF to test two or more conditions. If you have more than three conditions to test, consider using DO CASE instead of IF. Compare the example in this section with the example for DO CASE.

If you're evaluating a condition to decide which value you want to assign to a variable or property, you may be able to use the IIF() function, which involves less duplication (you don't have to type the target of the assignment twice).

You can nest IF statements to test multiple conditions; however, the ELSEIF option is an efficient alternative. When you use ELSEIF, you don't need to keep track of which ELSE applies to which IF, nor do you have to put in an ending ENDIF.

You can put many statements for each condition. If the number of statements in a set makes the code hard to read, consider putting them in a function and calling the function from the IF statement instead.

See also DO CASE, IIF()

IIF()

Returns one of two values depending on the result of a specified logical expression.

Syntax IIF(<expl>, <exp1>, <exp2>)

<expL> The logical expression to evaluate to determine whether to return <exp1> or <exp2>.

<exp1> The expression to return if <expL> evaluates to *true*.

<exp2> The expression to return if <expL> evaluates to *false*. The data type of <exp2> doesn't have to be the same as that of <exp 1>.

Description IIF() stands for "immediate IF" and is a shortcut to the IF...ELSE...ENDIF programming construct. Use IIF() as an expression or part of an expression where using IF would be cumbersome or not allowed. In particular, if you're evaluating a condition to decide which value you want to assign to a variable or property, using IIF() involves less duplication (you don't have to type the target of the assignment twice).

If <exp1> and <exp2> are *true* and *false*, in either order, using IIF() is redundant because <expL> must evaluate to either *true* or *false* anyway.

See Also IF

LOCAL

Declares memory variables that are visible only in the routine where they're declared.

Syntax LOCAL <memvar list>

<memvar list> The list of memory variables to declare local.

Description Use LOCAL to declare a list of memory variables available only to the routine in which the command is issued. Local variables differ from those declared PRIVATE in the following ways:

- Private variables are available to lower-level subroutines, while local variables are not. Local variables are accessible *only* to the routine—the program or function—in which they are declared.
- TYPE() does not “see” local variables. If you want to determine the TYPE() of a local variable, you must copy it to a private (or public) variable and call TYPE() with that variable name in a string.
- You cannot use a local variable for macro substitution with the & operator. Again, you must copy it to a private (or public) variable first.

Despite these limitations, local variables are generally preferred over private variables because of their limited visibility. You cannot accidentally overwrite them in a lower-level routine, which would happen if you forget to hide a public variable; nor can you inadvertently use a variable created in a higher-level routine, thinking that it's one declared in the current routine, which would happen if you misspell the variable name in the current routine.

Note The special variables *this* and *form* are local.

You must declare a variable `LOCAL` before initializing it to a particular value. Declaring a variable `LOCAL` doesn't create it, but it does hide any higher-level variable with the same name. After declaring a variable `LOCAL`, you can create and initialize it to a value with `STORE` or `=`. (The `:=` operator will not work at this point because the variable hasn't been created yet.) Local variables are erased from memory when the routine that creates them finishes executing.

For more information, see `PUBLIC` for a table that compares the scope of public, private, local, and static variables.

See Also `CLEAR MEMORY`, `PARAMETERS`, `PRIVATE`, `PUBLIC`, `RELEASE`, `STATIC`, `STORE`

LOOP

Skips the remaining statements in the current loop, causing another loop iteration to be attempted.

Syntax `LOOP`

Description Conditional statements are often used inside a loop to control which statements are executed in each loop iteration. For example, in a loop that processes the rows in an employee table, you might want to increase the monthly salary of non-managers and the annual bonus for managers, all in the same loop.

There can be many different sets of statements in the loop, each with a different combination of conditions dictating whether they should be executed. Sometimes you can be in the middle of a loop, and none of the remaining statements apply. The condition that determines this may be nested a few levels deep. While it would be possible to code the rest of the loop with conditional statements to take this condition into account, often it's simpler to use a `LOOP` statement when this condition is encountered. This causes the remaining statements in the loop to be skipped, and the next iteration of the loop to be attempted.

See also `DO WHILE`, `DO...UNTIL`, `EXIT`, `FOR...ENDFOR`

OTHERWISE

Designates a block of code in a `DO CASE` block to execute if there are no matching `CASE` blocks.

Description See `DO CASE` for details.

PARAMETERS

Assigns data passed from a calling routine to private variables.

Syntax PARAMETERS <*parameter list*>

<*parameter list*> The memory variable names to assign, separated by commas.

Description There are three ways to access values passed to program or function:

- Variable names may be declared on the FUNCTION (or PROCEDURE) line in parentheses. These variables are local to that routine.
- Variable names may be declared in a PARAMETERS statement. These variables are private in scope.
- The values may be retrieved through the ARGVECTOR() function.

Passed values may be assigned to variables only once in a routine. You may either create local variables on the FUNCTION line or use the PARAMETERS statement, and you may only use the PARAMETERS statement once.

The ARGVECTOR() function returns copies of the passed values, and has no effect nor is affected by the other two techniques.

In general, local variables are preferred because they cannot be accidentally overwritten by a lower-level routine. Reasons to use PARAMETERS instead include:

- Using values passed to a program file: a program file may contain statements that are not part of a function or class, like the statements in the Header of a WFM file. Because there is no FUNCTION or PROCEDURE line, there is no place to declare local parameters. A PARAMETERS statement must be used instead.
- You specifically want the parameters to be private, so they can for example be modified by a lower-level routine, or be used in a macro substitution.

For more information on the difference between local and private variable scope, see LOCAL.

If you specify more variables in the <*parameter list*> than values passed to the routine, the extra variables assume a value of *false*. If you specify fewer variables, the extra values do not get assigned.

The PARAMETERS statement should be at or near the top of the routine. This is good programming style; there is no rule requiring this.

Passing mechanisms There are two ways to pass parameters, *by reference* or *by value*. This section uses the term "variable" to refer to both memory variables and properties.

- If you pass variables by reference, the called function has direct access to the variable. Its actions can change (overwrite) the value in that variable. Pass

variables by reference if you want the called function to manipulate the values stored in the variables it receives as parameters.

- If you pass variables by value, the called function gets a copy of the value contained in the variable. Its actions can't change the contents of the variable itself. Pass variables by value if you want the called function to use the values in the variables without changing their values—on purpose or by accident—in the calling subroutine.

The following rules apply to parameter passing mechanisms:

- Literal values (like 7) and calculated expression values (like `xVar + 3`) must be passed by value—there is no reference for the called function to manipulate, nor is there any way to tell that the parameter has been changed.
- Memory variables and properties may be passed by reference or by value. The default is pass-by-reference.
- The scope declaration of a variable (local, private, etc.) *does not* have any effect on whether the variable is passed by reference or by value. The scope declaration protects the name of the variable. That name is used inside the calling routine; the called function assigns its own name (which is often different but sometimes happens to be the same) to the parameter, making the scope declaration irrelevant.
- To pass a variable or property by value, enclose it in parentheses when you pass it.

Passing objects Because an object reference is itself a reference, passing one as a parameter is a bit more complicated:

- Passing a variable (or property) that contains an object reference by reference means that you can change the contents of that variable, so that it points to another object, or contains any other value.
- Even if you pass an object reference by value, you can access that object, and change any of its properties. This is because the value of a object reference is still a reference to that object.

Passing *this* and *form* When passing the special object references *this* and *form* as parameters to the method of another object, they must be enclosed in parentheses to be passed by value. If not, the value of the *this* and *form* parameters take on the corresponding values for the target object, and no longer refer to the calling objects.

Passing fields in XBase DML With the XBase DML, fields are accessed directly by name (instead of a Field object's *value* property). When used as parameters, they are always passed by value, so the called function can't change their contents.

There are two ways to alter the contents of an XBase field with a function:

- Store its contents to a memory variable and call the function with that variable. When control returns to the calling routine, REPLACE the field contents with the memory variable contents.

- Design the function to accept a field name. Pass the name of the field, and have the function REPLACE the contents of the named field, using macro substitution to convert the field name to a field reference.

Protecting parameters from change Because the decision whether to pass by reference or by value is made by the caller, the called function doesn't know whether it's safe to modify the parameter. It's a good idea to copy parameters to work variables and to use those variables instead if their values are going to be changed, unless the intent of the function is specifically to modify the parameters.

Example The following contrived examples demonstrate the various aspects of the parameter passing mechanism. With the following program file, DOUBLE.PRG:

```
parameters arg
arg *= 2      // Double passed parameter
```

from the Command window, typing the following statements results in the values shown in the comments:

```
x = 7
double( x )      // Call as variable
? x              // Displays 14, pass by reference
double( x + 0 )  // Call as an expression
? x              // Displays 14, pass by value
double( (x) )    // Call with parentheses around variable name
? x              // Displays 14, pass by value

o = new Object()
o.x = 5
double( o.x )    // Call as property
? o.x           // Displays 10, pass by reference
double( (o.x) )  // Call with parentheses around property name
? o.x           // Displays 10, pass by value
```

With the following program DOUBLEX.PRG, designed specifically to modify the property *x* of the passed object:

```
parameters oArg
oArg.x *= 2
```

typing the following statements in the Command window results in the values shown in the comments:

```
doublex( o )      // Pass by reference
? o.x             // Displays 10, property modified
doublex( (o) )    // Pass by value
? o.x             // Displays 20, property still modified
```

With the following program ILIKEAPP.PRG:

```
parameter oArg
oArg := _app
```

passing by value will prevent the object reference itself from being changed:

```
f = new Form()
```

```

ilikeapp( f )      // Pass by value
? f.className     // Displays FORM
ilikeapp( f )      // Pass by reference
? f.className     // Displays APPLICATION, object reference changed
g = "test"        // Another variable, this one with a string
ilikeapp( g )      // Pass by reference
? g.className     // Displays APPLICATION, variable changed to an object reference

```

Note that you when assigning to a variable that was passed by reference, you are free to change the type of the variable.

This example demonstrates what happens if you don't enclose the special object reference *this* in parentheses when it is passed to the method of another object. (Codeblocks are used for the methods; codeblocks declare their parameters in-between pipe characters instead of using a PARAMETERS statement.)

```

f = new Form( "F" )      // text property is "F"
g = new Form( "G" )      // text property is "G"
f.test1 = {; g.meth( this )} // Pass-by-reference
f.test2 = {; g.meth( (this) )} // Pass-by-value
g.meth = {|o|; ? o.text} // Display text property of passed object
f.test1()                // Pass-by-reference displays "G"
f.test2()                // Pass-by-value displays "F"

```

Whenever an object's method is called, the value of *this* is automatically updated to point to that object. If the parameter *this* is passed by reference from the caller, the value of *this* changes to the called object before it is assigned to the parameter variable. By enclosing the parameter *this* in parentheses to make it pass-by-value, *this* does not change, and the parameter value is passed as expected.

See Also ARGVECTOR(), FUNCTION, LOCAL, PRIVATE

parent

The immediate container of an object.

Property of Most data access, form, and report objects

Description Many objects are related in a *containership hierarchy*. If the container object—referred to as the parent—is destroyed, all the objects it contains—referred to as child objects—are also destroyed. Child objects may be parents themselves and contain other objects. Destroying the highest-level parent destroys all the descendant child objects.

An object's *parent* property refers to its parent object.

For example, a form contains both data access objects and visual components. A Query object in a form has the form as its parent. The Query object contains a rowset, which contains an array of fields, which in turn contains Field objects. Each object in the hierarchy has a *parent* property that refers back up the chain, up to the form, which has no parent. A button on the form also has a *parent*

property that refers to the form. If the form is destroyed, all of the objects it contains are destroyed.

The *parent* property is often used to refer to sibling objects—other objects that are contained by the parent. For example, one Field object can refer to another by using the *parent* reference to go one level up in the hierarchy, then use the name of the other field to go back down one level to the sibling object.

The *parent* property is read-only.

See also n/a

PCOUNT()

Returns the number of parameters passed to a routine.

Syntax PCOUNT()

Description PCOUNT() is identical to ARGCOUNT().

PRIVATE

Declares variables that you can use in the routine where they're declared and in all lower-level subroutines.

Syntax PRIVATE <memvar list> |
ALL
[LIKE <memvar skeleton 1>]
[EXCEPT <memvar skeleton 2>]

<memvar list> The list of memory variables you want to declare private, separated by commas.

ALL Makes private all memory variables declared in the subroutine.

LIKE <memvar skeleton 1> Makes private the memory variables whose names are like the memory variable skeleton you specify for <memvar skeleton 1>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 1>.

EXCEPT <memvar skeleton 2> Makes private all memory variables except those whose names are like the memory variable skeleton you specify for <memvar skeleton 2>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 2>. You can use LIKE and EXCEPT in the same statement, for example, PRIVATE ALL LIKE ?_* EXCEPT c_*.

Description Use PRIVATE in a function to avoid accidentally overwriting a variable with the same name that was declared in a higher-level routine. Normally, variables are visible and changeable in lower-level routines. In effect, PRIVATE hides any

existing variable with the same name that was not created in the current routine. It's a good practice to always use `LOCAL` or `PRIVATE`. For example, if you write a function that someone else might use, you probably won't know what variables they're using. If you don't use `LOCAL` or `PRIVATE`, you might accidentally change the value of one of their variables when they call your function.

Although they have some limitations, local variables are generally preferred over private variables because of their more limited visibility. You cannot accidentally overwrite them in a lower-level routine, which would happen if you forget to hide a public variable; nor can you inadvertently use a variable created in a higher-level routine, thinking that it's one declared in the current routine, which would happen if you misspell the variable name in the current routine. Also, private variables may be macro-substituted inadvertently with the `&` operator. For example, if you specify the *text* of a menu item as `"&Close"` to designate the letter C as the pick character and you happen to have a private variable named `close`, the variable will be macro-substituted when the menu is created. If the variable was declared local, this wouldn't happen.

You must declare a variable `PRIVATE` before initializing it to a particular value. Declaring a variable `PRIVATE` doesn't create it, but it does hide any higher-level variable with the same name. After declaring a variable `PRIVATE`, you can create and initialize it to a value with `STORE` or `=`. (The `:=` operator will not work at this point because the variable hasn't been created yet.) Private variables are erased from memory when the routine that creates them finishes executing.

Unless declared otherwise, variables you initialize in programs are private. If you initialize a variable that has the same name as a variable created in the Command window or declared `PUBLIC` or `PRIVATE` in an earlier routine—in other words, a variable that is visible to the current routine—and don't declare the variable `PRIVATE` first, it is *not* created as a private variable. Instead, the routine uses and alters the value of the existing variable. Therefore, you should always declare your private variables, even though that is the default.

For more information, see `PUBLIC` for a table that compares the scope of public, private, local, and static variables.

See also `LOCAL`, `PUBLIC`, `STATIC`

PROCEDURE

Defines a function in a program file including variables to represent parameters passed to the function.

Description `PROCEDURE` is identical to `FUNCTION`. While earlier versions of dBASE differentiated between the two, these differences have been removed. The descriptive terms "function" and "procedure" are used interchangeably in *Visual*

dBASE. (The term “procedure file” refers to a program file opened with the SET PROCEDURE command, which is not restricted to a file that contains PROCEDURES only.)

See FUNCTION for details.

PUBLIC

Declares global memory variables.

Syntax PUBLIC <memvar list>

<memvar list> The memory variables to make public.

Description A variable’s *scope* is determined by two factors: its *duration* and its *visibility*. A variable’s duration determines when the variable will be destroyed, and its visibility determines in which routines the variable can be seen.

Use PUBLIC to declare a memory variable that has an indefinite duration and is available to all routines and to the Command window.

You must declare a variable PUBLIC before initializing it to a particular value. Declaring a variable PUBLIC creates it and initializes it to *false*. Once declared, a public variable will remain in memory until it is explicitly released.

By default, variables you initialize in the Command window are public, and those you initialize in programs without a scope declaration are private. (Variables initialized in the Command window when a program is suspended are private to that program.) The following table compares the characteristics of variables declared PUBLIC, PRIVATE, LOCAL and STATIC in a routine called CreateVar.

| | PUBLIC | PRIVATE | LOCAL | STATIC |
|--|--------|---------|-------|--------|
| Created when it is declared and initialized to a value of <i>false</i> | Y | N | N | Y |
| Can be used and changed in CreateVar | Y | Y | Y | Y |
| Can be used and changed in lower-level routines called by CreateVar | Y | Y | N | N |
| Can be used and changed in higher-level routines that call CreateVar | Y | N | N | N |
| Automatically released when CreateVar ends | N | Y | Y | N |

Public variables are rarely used in programs. To maintain global values, it’s better to create properties of the *_app* object. As properties, they will not conflict with variables that you might have with the same name, and they can communicate with each other more easily.

See Also CLEAR MEMORY, LOCAL, PRIVATE, RELEASE, RESTORE, SAVE, STATIC, STORE

QUIT

Closes all open files and terminates *Visual* dBASE.

Syntax QUIT [WITH <expN>]

WITH <expN> Passes a return code, <expN>, to the operating system when you exit *Visual* dBASE.

Description Use QUIT to end your *Visual* dBASE work. It has the same effect as closing the *Visual* dBASE application.

If you include QUIT in a program file, *Visual* dBASE halts the program's execution and exits *Visual* dBASE. To end a program's execution without leaving *Visual* dBASE, use CANCEL or RETURN.

Use QUIT WITH <expN> to pass a return code to Windows or to another application.

Example At the end of a long day, suppose you want to exit *Visual* dBASE and run the latest 3-D shoot-em-up game, which requires 128 MB of RAM. Your hands are already on the home keys of the keyboard, so instead of reaching to press *Alt-F4* or using the mouse to click the close button, you type the following in the Command window.

```
quit
```

REDEFINE

Assigns new values to an object's properties.

Syntax REDEFINE <class name> <object name>
 [OF <container object>]
 [FROM <row, col> TO <row, col> > | <AT <row, col>]
 [PROPERTY <changed property list>]
 [CUSTOM <new property list>]

<class name> The class of the object you want to redefine.

<object name> The identifier for the object you want to modify. <object name> is either an object reference variable, or a named property of the container if a <container object> is specified.

OF <container object> Identifies the object that contains the object you want to redefine.

FROM <row>, <col> TO <row>, <col> | AT <row>, <col> Specifies the new location and size of the object within its container. FROM and TO specify the upper left and lower right coordinates of the object, respectively. AT specifies the position of the upper left corner.

PROPERTY <changed property list> Specifies new values you assign to the existing properties of the object.

CUSTOM <new property list> Specifies new properties you create for the object and the values you assign to them.

Description Use REDEFINE to assign new values to the properties of an existing object. While the REDEFINE syntax offers some amenities (like DEFINE), it is not as flexible as assigning values in a WITH block. In particular, with REDEFINE you cannot assign values to the elements of properties that are arrays.

See Also CLASS, DEFINE, WITH

RELEASE

Deletes specified memory variables.

Syntax RELEASE <memvar list> |
ALL
[LIKE <memvar skeleton 1>]
[EXCEPT <memvar skeleton 2>]

<memvar list> The specific memory variables to release from memory, separated by commas.

ALL Removes all variables in memory (except system memory variables).

LIKE <memvar skeleton 1> Removes from memory all memory variables whose names are like the memory variable skeleton you specify for <memvar skeleton 1>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 1>.

EXCEPT <memvar skeleton 2> Removes from memory all memory variables except those whose names are like the memory variable skeleton you specify for <memvar skeleton 2>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 2>. You can use LIKE and EXCEPT in the same statement, for example, RELEASE ALL LIKE ?_* EXCEPT c_*.

Description Use RELEASE to clear memory variables. To remove large groups of variables, use the option ALL [LIKE <memvar skeleton 1>] [EXCEPT <memvar skeleton 2>]. If you issue RELEASE ALL [LIKE <memvar skeleton 1>] [EXCEPT <memvar skeleton 2>] in a program or function, Visual dBASE releases only the local and private variables declared in that routine. It doesn't release public or static variables, or variables declared in higher-level routines.

`release ()`

To release a variable by name, that variable must be in scope. For example, you may release a private variable declared in a higher-level routine by name, because the private variable is still visible; but you cannot release a local variable the same way because the local variable is not visible outside its routine.

Note RELEASE does not explicitly release objects. However, if the only reference to an object is in a memory variable, releasing the variable with RELEASE will in turn release the object. In contrast, RELEASE OBJECT will explicitly release an object, but it does not release any variables that used to point to that object.

When control returns from a subroutine to its calling routine, *Visual dBASE* clears from memory all variables initialized in the subroutine that weren't declared PUBLIC or STATIC. Thus, you don't have to release a routine's local or private variables explicitly with RELEASE before the routine terminates.

See Also CLEAR MEMORY, LOCAL, PRIVATE, PUBLIC, QUIT, RELEASE OBJECT, RESTORE, RETURN, SAVE, STATIC

release()

Explicitly releases an object from memory.

Syntax `<oRef>.release()`

<oRef> An object reference to the object you want to release.

Property of All form objects; all report objects except Band and StreamFrame.

Description *Visual dBASE* manages memory and resources used by objects automatically. When there are no more variables or properties that reference an object and that object is not visible onscreen, the object is destroyed. Any components that are contained in the object, such as the components of a form, are also destroyed when the container is destroyed. Because of this automatic object management, there is usually no reason to call *release()*.

release() explicitly releases an object from memory, returning *true* if successful. Any references that point to that object become invalid; attempting to use such a reference results in an error. If these references are tested with `EMPTY()`, it returns *true*.

For example, you might want to get rid of a single component in a form. You could *release()* that component, but in most cases you could just as easily hide the component by setting its *visible* property to *false*.

See also RELEASE OBJECT

RELEASE OBJECT

Explicitly releases an object from memory.

Syntax `RELEASE OBJECT <oRef>`

<oRef> An object reference to the object you want to release.

Description `RELEASE OBJECT` functions identically to the `release()` method. See `release()` for details.

Because `release()` is a method, its use is preferred, especially when called from a method. But `release()` is not a method in all classes. Use `RELEASE OBJECT` when the object does not have a `release()` method, or to release an object regardless of its class.

If `<oRef>` is a variable, `RELEASE OBJECT` does not release that variable, or any other variables that point to the just-released object. Testing these variables with `EMPTY()` will return `true` once the object has been released.

See Also `release()`

RESTORE

Copies the memory variables stored in the specified disk file to active memory.

Syntax `RESTORE FROM <filename> | ? | <filename skeleton>`
`[ADDITIVE]`

<filename> | ? | <filename skeleton> The file of memory variables to restore. `RESTORE FROM ?` and `RESTORE FROM <filename skeleton>` display a dialog box, from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with `SET PATH`. If you specify a file without including its extension, *Visual* dBASE assumes `MEM`.

ADDITIVE Preserves existing memory variables when `RESTORE` is executed.

Description Use `RESTORE` with `SAVE` to retrieve and store important memory variables. All local and private variables are cleared at the end of execution of the routine that created them, while all public and static variables are cleared when you exit *Visual* dBASE. To preserve these values for future use, store them in a memory file by using `SAVE`. You can then retrieve these values later by using `RESTORE`.

`SAVE` saves simple variables only—those containing numeric, string, logical, or null values—and objects of class `Array`. It ignores all other object reference variables. Therefore you can neither `SAVE` nor `RESTORE` objects (other than arrays).

Without the ADDITIVE option, RESTORE clears all existing user memory variables before returning to active memory the variables stored in a memory file. Use ADDITIVE when you want to restore a set of variables while retaining those already in memory.

Note If you use ADDITIVE and a restored variable has the same name as an existing variable, the restored variable will replace the existing one.

If you issue RESTORE in the Command window, *Visual* dBASE makes all restored variables public. When *Visual* dBASE encounters RESTORE in a program file, it makes all restored variables private to the currently executing function.

See Also CLEAR MEMORY, RELEASE, SAVE, STORE

RETURN

Ends execution of a program or function, returning control to the calling routine—program or function—or to the Command window.

Syntax RETURN [<return exp>]

<return exp> The value a function returns to the calling routine or the Command window.

Description Programs and functions return to their callers when there are no more statements to execute. When ended this way, they do not return a value.

Use RETURN in a program or function to return a value, or to return before the end of the program or function.

If the RETURN is inside a TRY block, the corresponding FINALLY block, if any, is executed before returning. If there is a RETURN inside that FINALLY block, whatever it returns is returned instead.

See also CANCEL, FUNCTION

SAVE

Stores memory variables to a file on disk.

Syntax SAVE TO <filename> | ? | <filename skeleton>
 [ALL]
 [LIKE <memvar skeleton 1>]
 [EXCEPT <memvar skeleton 2>]

TO <filename> | ? | <filename skeleton> Directs the memory variable output to be saved to the target file <filename>. By default, *Visual* dBASE assigns a MEM extension to <filename> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

ALL Stores all memory variables to the memory file. If you issue SAVE TO <filename> with no options, *Visual* dBASE also saves all memory variables to the memory file.

LIKE <memvar skeleton 1> Stores in the target file the memory variables whose names are like the memory variable skeleton you specify for <memvar skeleton 1>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 1>.

EXCEPT <memvar skeleton 2>] Stores in the target file all memory variables except those whose names are like the memory variable skeleton you specify for <memvar skeleton 2>. Use characters of the variable names and the wildcards * and ? to create <memvar skeleton 2>.

Description Use SAVE with RESTORE to store and retrieve important memory variables. Local and private variables are cleared at the end of the routine that created them, while public and static variables are cleared when you exit *Visual* dBASE. To preserve these values for future use, store them in a memory file with SAVE. Use RESTORE to retrieve them.

If SET SAFETY is ON and a file exists with the same name as the target file, *Visual* dBASE displays a dialog box asking if you want to overwrite the file. If SET SAFETY is OFF, any existing file with the same name is overwritten without warning.

Note SAVE saves simple variables only—those containing numeric, string, logical, or null values—and objects of class Array. It ignores all other object reference variables. Therefore you can neither SAVE nor RESTORE objects (other than arrays). SAVE also does not save function pointer, bookmark, or system memory variables.

See Also RELEASE, RESTORE, STORE

SET LIBRARY

Opens a *Visual* dBASE program file as the library file, making its functions, classes, and methods available for execution.

Syntax SET LIBRARY TO [<filename> | ? | <filename skeleton>]

<filename> | ? | <filename skeleton> The program file to open. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. If you

specify a file without including its extension, *Visual* dBASE assumes .PRO (a compiled object file). If *Visual* dBASE can't find a .PRO file, it looks for a .PRG file (a source file). If *Visual* dBASE finds a .PRG file, it compiles it.

Description SET LIBRARY is similar to SET PROCEDURE. Both commands open a program file, allowing access to the functions, classes, and methods the file contains. The difference is that while SET PROCEDURE can add a program file to a list of procedure files, there can be only one library file open at any time. The library file cannot be closed with the SET PROCEDURE command.

Otherwise, the library file is treated like a procedure file. The library and procedure files are searched in the order they were opened. You may want to designate a stable program file with core functionality as the library file, and all other program files as procedure files.

Issue SET LIBRARY TO without a file name to close the open library file.

See Also DO, FUNCTION, PROCEDURE, SET(), SET PROCEDURE

SET PROCEDURE

Opens a *Visual* dBASE program file as a procedure file, making its functions, classes, and methods available for execution.

Syntax SET PROCEDURE TO
[<filename> | ? | <filename skeleton>] [ADDITIVE]

<filename> | ? | <filename skeleton> The procedure file to open. The ? and <filename skeleton> options display a dialog box, from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. If you specify a file without including its extension, *Visual* dBASE assumes .PRO (a compiled object file). If *Visual* dBASE can't find a .PRO file, it looks for a .PRG file (a source file). If *Visual* dBASE finds a .PRG file, it compiles it.

ADDITIVE Opens the procedure file(s) without closing any you've opened with previous SET PROCEDURE statements. SET PROCEDURE TO <filename> (**without** the ADDITIVE option) closes all procedure files you've opened with previous SET PROCEDURE statements.

Description To execute a function or method, that function must be loaded in memory. To be more precise, a simple pointer to that function must be in memory. The contents of the function itself are not necessarily in memory at any given time; if not, the contents get loaded into memory automatically when the function is executed. But if that function's pointer is in memory, it is considered to be loaded.

Whenever you execute a program file with DO (or with the call operator), it is loaded implicitly; pointers to all of the functions, classes, and methods in that

file are loaded into memory. Therefore, code in a program file may always call any other functions or methods in the same file.

To access functions, classes, and methods in other program files, load the program file with SET PROCEDURE first. Its function pointers stay in memory until the program file is unloaded with CLOSE PROCEDURE or SET PROCEDURE TO (with no options).

Visual dBASE uses a reference count system to manage program files in memory. Each loaded program file has a counter for the number of times it has been loaded, either explicitly with SET PROCEDURE or implicitly. As long as the count is more than zero, the file stays loaded. Calling CLOSE PROCEDURE reduces the count by one. Therefore, if you issue SET PROCEDURE twice, you need to issue CLOSE PROCEDURE twice to close the program file.

A program file's load count has no impact on memory; it is simply a counter. Loading a program file 10 times uses the same amount of memory as loading it once.

Whenever a function is called, *Visual* dBASE looks for the routine in specific places in a specific order. After searching the program files in the call chain, *Visual* dBASE looks in files opened with SET PROCEDURE. See the DO command for an explanation of the search path and order.

To make the file containing the currently executing routine a procedure file—for example, after creating an object, to make the object's methods which are defined in the same file available to it—execute the following statement:

```
set procedure to program(1) additive
```

Some operations, such as assigning a *menuFile* to a form or opening a form defined in a WFM file, automatically open the associated file as a procedure file, and that statement is not necessary.

If you issue SET PROCEDURE TO with no options, *Visual* dBASE closes all procedure files you've opened with SET PROCEDURE. If you want to close only specific procedure files, use CLOSE PROCEDURE. The maximum number of open procedure files depends on available memory.

Note A common mistake is to forget the ADDITIVE clause when opening a procedure file. This will close all other open procedure files.

When *Visual* dBASE executes or loads a program file, it will automatically compile the program file into object code when either:

- There is no object code file, or
- SET DEVELOPMENT is ON, and program file is newer than the object code file (the source code file's last update date and time is later than the object code file's)

If a file is opened as a procedure file and the file is changed in the Source editor, the file is automatically recompiled so that the changed code takes effect immediately.

Use TYPE() to detect whether a function, class, or method is loaded into memory. If so, TYPE() will return "FP" (for function pointer), as shown in the following IF statements:

```
if type( "myfunc" ) # "FP"           // Function name
if type( "myclass::myclass" ) # "FP" // Class constructor name
if type( "myclass::mymethod" ) # "FP" // Method name
```

See Also CLOSE PROCEDURE, COMPILE, DO, FUNCTION, SET(), SET LIBRARY

SET()

Returns the current setting of a SET command or function key.

Syntax SET(<expC> [, <expN>])

<expC> A character expression that is the SET command or function key whose setting value to return.

<expN> The *n*th such setting to return.

Description Use SET() to get a SET or function key setting so that you can change it or save it. For example, you can issue SET() at the beginning of a routine to get current settings. You can then save these settings in memory variables, change the settings, and restore the original settings from the memory variables at the end of the routine.

When *Visual* dBASE supports a SET and a SET...TO command that use the same keyword, SET() returns the ON|OFF setting and SETTO() returns the SET...TO setting. For example, you can issue SET FIELDS ON, SET FIELDS OFF, or SET FIELDS TO <field list>. SET("FIELDS") returns "ON" or "OFF" and SETTO("FEILDS") returns the field list as a character expression.

If *Visual* dBASE supports a SET...TO command but not a corresponding SET command, SET() and SETTO() both return the SET...TO value. For example, SET("BLOCKSIZE") and SETTO("BLOCKSIZE") both return the same value.

When <expC> is a function key name, such as "F4", SET() returns the function key setting. To return the value of a Ctrl+function key setting, add 10 to the function key number; to return the value of a Shift+function key setting, add 20 to the function key number. That is, to return the value of *Ctrl+F4*, use SET("F14"), and to return the value of *Shift+F4*, use SET("F24").

If a procedure file is open, SET("PROCEDURE") returns the name of the procedure file. If more than one procedure file is open, SET("PROCEDURE") returns the name of the first one loaded. To return the name of another open procedure file, enter a number as the second argument; for example, SET("PROCEDURE",2) returns the name of the second procedure file that was loaded. If no procedure files are open, SET("PROCEDURE") returns an empty string ("").

The command you specify for *<expC>* can be abbreviated to four letters in most cases, following the same rules as those for abbreviating keywords. For example, SET("DECI") and SET("DECIMALS") have the same meaning. The *<expC>* argument is not case-sensitive.

Example The following example stores the value of a setting at the beginning of a function, and restores it at the end.

```
function findMatch( xArg )
  local lRet
  private cExact           // Can't be local for macro substitution
  cExact = set( "EXACT" )   // Store "ON" or "OFF" to character variable
  set exact on
  lRet = seek( xArg )       // Does exact match exist?
  set exact &cExact         // Either "set exact ON" or "set exact OFF"
  return lRet
```

See Also DISPLAY STATUS, SET, SET FUNCTION, SETTO()

SETTO()

Returns the current setting of a SET...TO command or function key.

Syntax SETTO(*<expC>* [,*<expN>*])

<expC> A character expression that is the SET...TO command whose setting value to return.

<expN> The *n*th such setting to return.

Description Use SETTO() to get a SET or function key setting so that you can change it or save it. For example, you can issue SETTO() at the beginning of a routine to get current settings. You can then save these settings in memory variables, change the settings, and restore the original settings from the memory variables at the end of the routine.

When *Visual* dBASE supports a SET and a SET...TO command that use the same keyword, SET() returns the SET setting and SETTO() returns the SET...TO setting. For example, you can issue SET FIELDS ON, SET FIELDS OFF, or SET FIELDS TO *<field list>*. SET("FIELDS") returns the ON or OFF setting and SETTO("FIELDS") returns the field list as a character expression.

SETTO() is almost identical to SET(). For more information, see SET().

See Also DISPLAY STATUS, SET, SET(), SET FUNCTION

STATIC

Declares memory variables that are local in visibility but public in duration.

Syntax `STATIC <variable 1> [= <value 1>] [, <variable 2> [= <value>] ...]`

<variable> The variable to declare static.

<value> The value to assign to the variable.

Description Use `STATIC` to declare memory variables that are visible only to the routine where they're declared but are not automatically cleared when the routine ends. Static variables are different from other scopes of memory variables in two important ways:

- You can declare and assign a value to a static variable in a single statement, referred to as an *in-line* assignment.
- Static variables initialized in a single statement are assigned the initialization value whenever the variable is undefined, including the first time the routine is executed and after the variable is cleared.

You must declare a variable `STATIC` before initializing it to a particular value. Declaring a variable `STATIC` without an in-line assignment creates it and initializes it to *false*. Once declared, a static variable will remain in memory until it is explicitly released (usually with `CLEAR MEMORY`).

Because static variables are not released when the routine in which they are created ends, you can use them to retain values for subsequent times that routine runs. To do this, use an in-line assignment. The first time *Visual* dBASE encounters the `STATIC` declaration, the variable is initialized to the in-line value. If the subroutine is run again, the variable is not reinitialized; instead, it retains whatever value it had when the routine last ended.

Because *Visual* dBASE is a dynamic object-oriented language, you usually assign new properties to an object to retain values between method calls. For example, if you're calculating a running total in a report, you can create a property of the Report or Group object to store that number.

Static variables are only useful for truly generic functions that are not associated with objects, functions that might be called from different objects that need to share a persistent value, or for values that are maintained by a class—not each object. In this last case, the variables are referred to as *static class variables*.

For more information, see `PUBLIC` for a table that compares the scope of public, private, local, and static variables.

TOPIC~>

Example The following is a stopwatch function that returns the number of seconds since the last time it was called.

```
function stopwatch()
  local thisTime, nSecs
  thisTime = new Date().getTime()
  static lastTime = thisTime
  nSecs = ( thisTime - lastTime ) / 1000
  lastTime := thisTime
  return nSecs
```

The function uses a Date object's *getTime()* method, which keeps time in milliseconds. Whenever the function is called, the variable *firstTime* is set to the current time in milliseconds. The first time through the function, the *lastTime* variable is set to that same time. The difference is calculated, and then the value of *thisTime* is saved in the static variable *lastTime* for the next function call.

To reset the timer, call the function; you may ignore the return value. Then the next time you call the function, you will get the elapsed time. If you're measuring a series of intervals, call the function once between intervals. For example:

```
stopwatch()          // Reset timer
// Process 1
time1 = stopwatch() // Time for first process
// Process 2
time2 = stopwatch() // Time for second process
// etc.
```

The static variable *lastTime* maintains its value between function calls. The inline assignment makes sure it has a value the first time the function is called (the function will return zero the first time), and is ignored from then on, unless the variable is explicitly released. The static variable is hidden to all other functions, so you can't accidentally overwrite it.

See Also CLEAR MEMORY, LOCAL, PRIVATE, PUBLIC, RELEASE

STORE

Stores an expression to specified memory variables or properties.

Syntax STORE <exp> TO <memvar list>
<exp> The expression to store.

TO <memvar list> The list of memory variables and/or properties to store <exp>, separated by commas.

Description Use STORE to store any valid expression to a one or more variables or properties.

Common style dictates the use of STORE only when storing a single value to multiple locations. When storing to a single variable or property, an assignment operator, either = or :=, is preferred.

To specify the *scope* of a variable, use LOCAL, PRIVATE, PUBLIC, or STATIC before assigning a value to the variable.

See Also LOCAL, PRIVATE, PUBLIC, RESTORE, SAVE, STATIC

THROW

Generates an exception.

Syntax `THROW <exception oRef>`
<exception oRef> A reference to the Exception object you want to pass to the CATCH handler.

Description Use THROW to manually generate an exception. THROW must pass a reference to an existing Exception object that describes the exception.

Example Suppose you are using exceptions to manage execution in a deeply nested set of conditional statements and loops. You create your own exception class:
 If there is a normal error, the second CATCH block saves it to a log file, using a function you wrote, and execution continues.

See also class Exception, TRY...ENDTRY

TRY

A control statement used to handle exceptions and other deviations of program flow.

Syntax `TRY`
 `<statement block 1>`
 `[CATCH(<exception type1> <exception oRef1>)`
 `<statement block 2>]`
 `[CATCH(<exception type2> <exception oRef2>)`
 `<statement block 3>]`
 `[CATCH ...]`
 `[FINALLY`
 `<statement block 4>]`
 `ENDTRY`

TRY <statement block 1> A statement block for which the following CATCH or FINALLY block—or both—will be used if an exception occurs during execution. A TRY block must be followed by either a CATCH block, a FINALLY block, or both.

CATCH <statement block 2> A statement block that is executed when an exception occurs.

<exception type> The class name of the exception to look for—usually, *Exception*.

<exception oRef> A formal parameter to receive the Exception object passed to the CATCH block.

CATCH... Catch blocks for other types of exceptions.

FINALLY <statement block 4> A statement block that is always executed after the TRY block, even if an exception or other deviation of program flow occurs. If there is both a CATCH and a FINALLY, the FINALLY block executes after the CATCH block.

ENDTRY A required keyword that marks the end of the TRY structure.

Description

An *exception* is a condition that is either generated by *Visual* dBASE, usually in response to an error, or by the programmer. By default, *Visual* dBASE handles an exception by displaying an error dialog and terminating the currently executing program. You can use FINALLY to make sure some code gets executed even if there is an exception, and CATCH to handle the exception yourself, in the following combinations:

- For a block of code that may generate an exception, place the code inside a TRY block. To prevent the exception from generating a standard error dialog and terminating execution, place exception handling code in a CATCH block after the TRY. If an exception occurs, execution immediately jumps to the CATCH block; no more statements in the TRY block are executed. If no exception occurs, the CATCH block is not executed.
- If there's some code that should always be executed at the end of a process, whether or not the process completes successfully, place that code in a FINALLY block. With TRY and FINALLY but no CATCH, if an exception occurs during the TRY block, execution immediately jumps to the FINALLY block; no more statements in the TRY block are executed. Since there was no CATCH, you would still have an exception, which if not handled by a higher-level CATCH as described later, *Visual* dBASE would handle as usual, after executing the FINALLY block. If no exception occurs, the FINALLY block is executed after the TRY.
- If you have all three—TRY, CATCH, and FINALLY—if an exception occurs, execution immediately jumps to the CATCH block; after the CATCH block executes, the FINALLY block is executed. If there is no exception during the TRY, then the CATCH block is skipped, and the FINALLY block is executed.

The code that is covered by TRY doesn't have to be inside the statement block physically; the coverage exists until that entire block of code is executed. For example, you may have a function call inside a TRY block, and if an exception occurs while that function is executing—even if that function is defined in another programfile—execution jumps back to the corresponding CATCH or FINALLY.

A TRY block may be followed by multiple CATCH blocks, each with its own *<exception type>*. When an exception occurs, *Visual* dBASE compares the *<exception type>* with the *className* property of the Exception object. If they match, that CATCH block is executed and all others are skipped. If the *className* does not match, *Visual* dBASE searches the class hierarchy of that object to find a match. If no match is found, the next CATCH block is tested.

Class name matches are not case-sensitive. For example, the `DbException` class is a subclass of the `Exception` class. If the blocks are arranged like this:

```
try
  // Statements
catch ( DbException e )
  // Block 1
catch ( Exception e )
  // Block 2
endtry
```

and a `DbException` occurs, execution goes to Block 1, because that's a match. If an `Exception` occurs, execution goes to Block 2, because Block 1 doesn't match, but Block 2 does. If the blocks are arranged the other way around, like this:

```
try
  // Statements
catch ( Exception e )
  // Block 1
catch ( DbException e )
  // Block 2
endtry
```

then all exceptions always go to Block 1, because all `Exceptions` are derived from the `Exception` class. Therefore, when using multiple `CATCH` blocks, list the most specific exception classes first.

You can generate exceptions on purpose with the `THROW` statement to control program flow. For example, if you enter deeply nested control structures or subroutines from a `TRY` block, you can `THROW` an exception from anywhere in the nested code. This would cause execution to jump back to the corresponding `CATCH` or `FINALLY`, instead of having to exit each control structure or subroutine one-by-one.

You may also nest `TRY` structures. An exception inside the `TRY` block causes execution to jump to the corresponding `CATCH` or `FINALLY`, but an exception in a `CATCH` or `FINALLY` is simply treated as an exception. Also, if you have a `TRY` and `FINALLY` but no `CATCH`, that leaves you with an unhandled exception. If the `TRY/CATCH/FINALLY` is itself inside a `TRY` block, then that exception would be handled at that next higher level, as illustrated in the following code skeleton:

```
try
  // exception level 1
  try
    // exception level 2
    catch ( Exception e )
      // handler for level 2
    // but exception level 1
  finally
    // level 2
  endtry
catch ( Exception e )
  // handler for level 1
endtry
```

Note that if an exception occurs in the level 2 CATCH, the level 2 FINALLY is still executed before going to the level 1 CATCH, because a FINALLY block is always executed after a TRY block.

In addition to exceptions, other program flow deviations—specifically EXIT, LOOP, and RETURN—are also caught by TRY. If there is a corresponding FINALLY block, it's executed before control is transferred to the expected destination. (CATCH catches only exceptions.)

Example The following example illustrates how to code a transaction, which is an all-or-nothing attempt at multiple database changes. If any of the changes should fail—for example, attempting to write a new record to disk, which would fail if there was no more disk space—the entire transaction must be rolled back:

```
try
    form.rowset.parent.database.beginTrans() // Begin the transaction
    //
    // make changes
    //
    form.rowset.parent.database.commit()      // If you got this far, there were no
                                              // errors, so commit the changes
catch ( Exception e ) // The parameter receives the Exception object that describes
                      // the error (not used in this example, but required)
    form.rowset.parent.database.rollback()    // Undo any changes that did take
    // display an error message
endtry
```

This example runs a process in a subdirectory, the name of which is passed as the parameter *cDir*. It uses two TRY blocks to create the subdirectory if necessary, and return to the previous directory even if there is an error in the process:

```
try
    // Instead of bothering to see if the directory already exists
    md &cDir // Go ahead and try to create the directory
catch ( Exception e ) // If there's an error creating the directory,
                      // execution goes here.
    // Do nothing -- this assumes the error is because the directory already exists.
    // By using a CATCH, the error is ignored.
finally
    cd &cDir // Now try and go to that directory
    // At this point, if you can't go to the directory, then that's a real error.
    // That would be handled normally, since the error occurred in the FINALLY and
    // is not nested inside another TRY.
endtry

try
    //
    // Run the process
    //
    // No CATCH, so if there's an error, there will be a dialog
finally
    // But because of this FINALLY, the previous directory will be restored regardless.
    // This makes the code easier to re-test, since you don't have to switch back to
    // your main directory manually after an error.
```

TYPE()

```
cd ..
endtry
```

Note that in the first TRY block, the statement to switch to the subdirectory doesn't have to be in a FINALLY block. Unlike the second TRY block, where the FINALLY will switch back to the parent directory even if there is an error, the switch to the subdirectory would work just as well between the two TRY blocks. It's shown in the FINALLY as an example of what would happen if there is an exception in a FINALLY block.

See also EXIT, LOOP, RETURN, THROW

TYPE()

Returns a character string that indicates a specified expression's data type.

Syntax TYPE(<expC>)
<expC> A character string containing the expression whose type to evaluate and return.

Description Use TYPE() to determine the data type of an expression, including whether a variable is undefined.

TYPE() expects a character string containing the expression. This allows you to specify a variable name that may not exist. (If you were to use an actual expression with an undefined variable instead of putting the expression in a string, the expression would cause an error.) The <expC> may be any valid character expression, including a variable or a literal string representing the expression to evaluate.

TYPE() returns a character string containing a one- or two- letter code indicating the data type. The following table lists the values TYPE() returns.

| Expression type | TYPE() code |
|---|--------------|
| Array object | A |
| DBF or Paradox binary field (BLOB) | B |
| Bookmark | BM |
| Character field or string value, Paradox alphanumeric field | C |
| Codeblock | CB |
| Date field or value, Paradox date field | D |
| Float field, Paradox numeric or currency field | F |
| Function pointer | FP |
| OLE (general) field | G |
| Logical field or value | L |
| DBF or Paradox memo field | M |
| DBF numeric field or value | N |

| Expression type | TYPE() code |
|---|--------------|
| Object reference (other than Array) | O |
| Undefined variable, field, invalid expression, or <i>null</i> | U |

Note that an object of class Array is a special case. Unlike other objects, its code is "A" (this is for backward compatibility with earlier versions of dBASE).

TYPE() cannot "see" variables declared as local or static. If there is a public or private variable hidden by a local or static variable of the same name, then TYPE() will return the code for that hidden variable. Otherwise, that variable and any expression using that variable is considered undefined.

Use TYPE() to detect whether a function, class, or method is loaded into memory. If so, TYPE() will return "FP" (for function pointer), as shown in the following IF statements, which detect if the named function is *not* loaded (this is done to determine if the specified function needs to be loaded):

```
if type( "myfunc" ) # "FP"           // Function name
if type( "myclass::myclass" ) # "FP" // Class constructor name
if type( "myclass::mymethod" ) # "FP" // Method name
```

Example The following statements demonstrate that TYPE() expects a character string containing an expression:

```
dVar = date()           // Create a date variable
? type( dVar )          // Error: Data type mismatch. Expecting: Character
? type( "dVar" )        // Displays "D" for Date, as do the next two statements
? type( "D" + "var" )    // Any expression containing the variable name works
                        // (and variable names are not case-sensitive)
? type( "date()" )      // String can contain any expression, not just single variable
```

The following routine is used to read the data in a generated text file into the corresponding fields of a table. Character fields in the text file are the same length as in the table. Dates are formatted in six characters as MMDDYY (which matches the current SET DATE format). Numbers are always twelve characters and represent currency stored in cents, so it needs to be divided by 100.

```
function decodeLine( cLine, aDest )
#define YEAR_LEN 2
#define NUM_LEN 12
local nPtr, nFld, cFld, nLen
nPtr = 1 // Pointer into string
for nFld = 1 to fldcount()
cFld = field( nFld ) // Store name of field in string variable for reuse
do case
case type( cFld ) == "C"
aDest[ nFld ] = substr( cLine, nPtr, length( nFld ) )
nPtr += length( nFld )
case type( cFld ) == "D"
aDest[ nFld ] = ctod( substr( cLine, nPtr, 2 ) + "/" + ;
                      substr( cLine, nPtr + 2, 2 ) + "/" + ;
                      substr( cLine, nPtr + 4, YEAR_LEN ) )
nPtr += 2 + 2 + YEAR_LEN
case type( cFld ) == "N"
```

WITH

```
        aDest[ nFld ] = val( substr( cLine, nPtr, NUM_LEN ) ) / 100
        nPtr += NUM_LEN
    endcase
endfor
```

An array is passed to the routine along with the line to read. The field values are stored in the array, which is appended to the table with APPEND FROM ARRAY in the calling routine (not shown here). The function defines some manifest constants for the size of a numeric field and whether the year is two or four digits in case this changes in the future. A FOR loop goes through each field in the table. The name of each field is stored in a variable for convenience; it's used repeatedly in the DO CASE structure. The variable is string containing the name of the field, which TYPE() expects. In contrast, TYPE("cFLD") would always return "C".

The following function is a slight variation on the TYPE() function. It returns the type of a value, but it expects the expression itself as a parameter instead of a string containing the expression. It therefore cannot handle undefined or invalid expressions, but it can handle local and static variables. It also returns the character "0" (zero) when the expression contains the value *null*, to differentiate it from expressions that have no value. For example, if you have a function that does not RETURN a value, that function call has an undefined value.

```
function valType
    parameter x
    return iif( x == null, "0", type( "x" ) )
```

The function works by taking the parameter as a private variable named *x*. Then if it's not *null*, the TYPE() function is used with the string "x" to return the type of the parameter.

See Also EMPTY(), STORE,

WITH

A control statement that causes all the variable and property references within it to first assume that they are properties of the specified object.

Syntax WITH <oRef>
 <statement block>
ENDWITH

<oRef> A reference to the default object.

<statement block> A statement block that assumes that the specified object is the default.

ENDWITH A required keyword that marks the end of the WITH structure.

Description Use WITH when working with multiple properties of the same object. Instead of using the object reference and the dot operator every time you refer to a

property of that object, you specify the object reference once. Then every time a variable or property name is used, it is first checked to see if that name is a property of the specified object. If it is, then that property is used. If not, then the variable or property name is used as-is.

You cannot take advantage of the WITH syntax to create properties. For example:

```
with someObject
  existingProperty = 2
  newProperty = existingProperty
endwith
```

Suppose that *existingProperty* is an existing property of the object, and *newProperty* is not. In the first statement in the WITH block, the value 2 is assigned to the existing property. Then in the second statement, *newProperty* is treated like a variable, because it does not exist in the object. The statement creates a variable named *newProperty*, assigning to it the value of the *existingProperty* property.

Method name conflicts You may encounter naming conflicts when calling methods inside a WITH block in two ways:

The name of the method matches the name of a built-in function. The built-in function takes precedence. For example, you create a class with a method *center()* and try to call it within a WITH block:

```
with x
  center()
  // other code
endwith
```

The *CENTER()* function would be called. It expects parameters, so you'll get a runtime error. You might check your *center()* method, which has no parameters, and wonder what's going on.

It may be possible to call your method by using the explicit object reference, which is normally redundant in a WITH block, and will not work if the object happens to have a property with the same name as the object reference. For example, you could call your *center()* method like this:

```
with x
  x.center()
  // other code
endwith
```

If the object *x* happens to have a property named *x*, then you would have to create a temporary duplicate reference that does not have the same name as the any other property of *x* outside the WITH block first:

```
y = x
with x
  y.center()
  // other code
endwith
```

The name of the method matches the first word of a command. For example, if the object *f* has a method named *open()*, the method call with the dot operator would look like:

```
f.open()
```

Using WITH, it would be:

```
with f
  open()
endwith
```

but that code will not work because the name of the method matches the first word in a *Visual* dBASE command; there are some commands that start with the word OPEN. When the compiler sees the word OPEN, it considers that statement to be a command starting with that keyword, and looks for the rest of the command; for example, OPEN DATABASE. When it doesn't find the rest of the command, it considers the statement to be incorrect and generates a compile-time error.

To call such a method inside a WITH block, you may use an explicit object reference as shown above, or change the statement from a direct method call to an indirect method call—an assignment or through the EMPTY() function. Many methods return values. By assigning the return value of the method call to variable, even a dummy variable, you bypass the naming conflict. For example, with another object that has a *copy()* method (there are several commands that begin with the word COPY):

```
with x
  dummy = copy() // As long as x does not have property named dummy!
endwith
```

For methods that don't return values, you may use the EMPTY() function, which will safely "absorb" the undefined value:

```
with x
  empty( copy() )
endwith
```

Example The following code from a WFM file assigns values to the properties of a newly created Query object inside a WITH block. In this excerpted code, *this* refers to the form:

```
this.messages1 = new Query()
with this.messages1
  left = 55.25
  top = 4.9
  sql = "select * from MESSAGES.DB"
  active = true
endwith
```

Without the WITH block, the code would have looked like this:

```
this.messages1 = new Query()
this.messages1.left = 55.25
this.messages1.top = 4.9
this.messages1.sql = "select * from MESSAGES.DB"
```

```
this.messages1.active = true
```


String objects

Visual dBASE supports two types of strings:

- A *primitive string* that is compatible with earlier versions of dBASE
- A JavaScript-compatible String object.

Visual dBASE will convert one type of string to the other on-the-fly as needed. For example, you may use a String class method on a primitive string value or a literal string:

```
? version().toUpperCase()  
? "peter piper pickles".toProperCase()
```

This creates a temporary String object from which the method or property is called. You may also use a string function on a String object.

Many string object methods have built-in string functions that are practically identical, while others are merely similar with subtle differences.

Note JavaScript is zero-based; dBASE is one-based. For example, to extract a substring starting with the third character, you would use the parameter 2 with the *substring()* method, and the parameter 3 with the SUBSTR() function.

The maximum length of a string in *Visual* dBASE is approximately 1 billion characters, or the amount of virtual memory, whichever is less.

class String

A string of characters.

Syntax [*<oRef>* =] new String([*<expC>*])

<oRef> A variable or property in which you want to store a reference to the newly created String object.

<expC> The string you want to create. If omitted or *null*, the resulting string is empty.

Properties The following tables list the properties and methods of the String class. (No events are associated with this class.)

| Property | Default | Description |
|------------------|---------|--|
| <i>className</i> | STRING | Identifies the object as an instance of the String class (Property discussed in Chapter 5, “Core language.”) |
| <i>length</i> | | The number of characters in the string |
| <i>string</i> | | The value of the String object |

| Method | Parameters | Description |
|----------------------|----------------------------------|--|
| <i>anchor()</i> | <expC> | Tags the string as an anchor <A NAME> |
| <i>asc()</i> | <expC> | Returns the ASCII value of the first character in the designated string |
| <i>big()</i> | | Tags the string as big <BIG> |
| <i>blink()</i> | | Tags the string as blinking <BLINK> |
| <i>bold()</i> | | Tags the string as bold <BOLD> |
| <i>charAt()</i> | <index expN> | Returns the character in the string at the designated position |
| <i>chr()</i> | <expN> | Returns the character equivalent of the specified ASCII value |
| <i>fixed()</i> | | Tags the string as fixed font <TT> |
| <i>fontcolor()</i> | <expC> | Tags the string as the designated color |
| <i>fontsize()</i> | <expN> | Tags the string as the designated font size |
| <i>getBytes()</i> | <index expN> | Returns the value of the byte at the specified index in the string |
| <i>indexOf()</i> | <expC> [, <start index expN>] | Returns the position of the search string inside the string |
| <i>isAlpha()</i> | | Returns <i>true</i> if the first character of the string is alphabetic |
| <i>isLower()</i> | | Returns <i>true</i> if the first character of the string is lowercase |
| <i>isUpper()</i> | | Returns <i>true</i> if the first character of the string is uppercase |
| <i>italics()</i> | | Tags the string as in italics <I> |
| <i>lastIndexOf()</i> | <expC> [, <start index expN>] | Returns the position of the search string inside the string, searching backwards |
| <i>left()</i> | <expN> | Returns the specified number of characters from the beginning of the string |
| <i>leftTrim()</i> | | Returns the string with all leading spaces removed |
| <i>link()</i> | <expC> | Tags the string as a link <A HREF> |
| <i>replicate()</i> | <expC> [, <expN>] | Returns the specified string repeated a number of times |

| Method | Parameters | Description |
|-----------------------|--|---|
| <i>right()</i> | <i><expN></i> | Returns the specified number of characters from the end of the string |
| <i>rightTrim()</i> | | Returns the string with all trailing spaces removed |
| <i>setByte()</i> | <i><index expN></i> , <i><value expN></i> | Assigns a new value to the byte at the specified index in the string |
| <i>small()</i> | | Tags the string as small <SMALL> |
| <i>space()</i> | <i><expN></i> | Returns a string comprising the specified number of spaces |
| <i>strike()</i> | | Tags the string as strikethrough <STRIKE> |
| <i>stuff()</i> | <i><start expN></i> <i>, <quantity expN></i> <i>[, <replacement expC>]</i> | Returns the string with specified characters removed and others inserted in their place |
| <i>sub()</i> | | Tags the string as subscript <SUB> |
| <i>substring()</i> | <i><start index expN></i> <i>, <end index expN></i> | Returns a substring derived from the string |
| <i>sup()</i> | | Tags the string as superscript <SUP> |
| <i>toLowerCase()</i> | | Returns the string in all lowercase |
| <i>toProperCase()</i> | | Returns the string in proper case |
| <i>toUpperCase()</i> | | Returns the string in all uppercase |

Description A String object contains the actual string value, stored in the property *string*, and methods that act upon that value. The methods do not modify the value of *string*; they use it as a base and return another string, number, or *true* or *false*.

The methods are divided into three categories: those that simply wrap the string in HTML tags, those that act upon the contents of the string, and static class methods that do not operate on the string at all.

Because the return values for most string methods are also strings, you can call more than one method for a particular string by chaining the method calls together. For example,

```
cSomething.substring( 4, 7 ).toUpperCase()
```

Example When you concatenate a *null* to a string, the result is *null*. By passing a value that may be *null* through the String class constructor, you can safely concatenate two values without using cumbersome conditional constructs. For example, suppose you are combining, first name, middle initial, and last name. The middle initial field may be *null*. You can safely combine the three like this:

```
fullName = firstName + " " + new String( middleInitial + " " ) + lastName
```

If the middle initial is *null*, adding a space to it results in *null*, and the String object will be an empty string. If the middle initial is not *null*, adding a space will result in a space between the middle initial and the last name.

ASC()

Returns the numeric ASCII value of a specified character.

Syntax ASC(<expC>)

<expC> The character whose ASCII value you want to return. You can specify a string with more than one character, but *Visual* dBASE uses only the first one.

Description ASC() is the inverse of CHR(). ASC() accepts a character and returns its ASCII value—a number from 0 to 255, inclusive. CHR() accepts an ASCII value and returns its character.

See the ASCII table in Appendix A for a listing of ASCII values and their corresponding characters.

Other than the syntactic difference of being a method instead of a function, the *asc*() method behaves identically to the ASC() function.

Example Executing the following statements in the Command window demonstrates the relation between ASC() and CHR():

```
? asc( "A" )           // 65
? chr( asc( "A" ) + 32 ) // "a"
? asc( chr( asc( "A" ) + 32 ) ) // 97
```

In the following example, if the variable cDrive contains a drive letter, ASC() is used to convert the drive letter to a number suitable for the DISKSPACE() function.

```
nDiskSpace = diskSpace( asc( cDrive ) - asc( "A" ) + 1 )
```

See also *asc*(), CHR()

asc()

Returns the numeric ASCII value of a specified character.

Syntax <oRef>.asc(<expC>)

<oRef> A reference to a String object.

<expC> The character whose ASCII value you want to return. You can specify a string with more than one character, but *Visual* dBASE uses only the first one.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the ASC() function.

See also ASC(), chr()

AT()

Returns a number that represents the position of a string within another string.

Syntax AT(<search expC>, <target expC> [, <nth occurrence expN>])

<search expC> The string you want to search for in <target expC>.

<target expC> The string in which to search for <search expC>.

<nth occurrence expN> Which occurrence of the string to find. By default, Visual dBASE searches for the first occurrence. You can search for other occurrences by specifying the number, which must be greater than zero.

Description AT() returns the numeric position where a *search string* begins in a *target string*. AT() searches one character at a time from left to right, beginning to end, from the first character to the last character. The search is case-sensitive. Use UPPER() or LOWER() to make the search case-insensitive.

You can specify which occurrence of the search string to find by specifying a number for <nth occurrence expN>. If you omit this argument, AT() returns the starting position of the first occurrence of the search string.

AT() returns 0 when

- The search string isn't found.
- The search string or target string is empty.
- The search string is longer than the target string.
- The <nth occurrence expN> occurrence doesn't exist.

When AT() counts characters in a memo field, it counts two characters for each carriage-return and linefeed combination (CR/LF) in the memo field.

Use RAT() to find the starting position of <search expC>, searching from *right to left*, end to beginning. Use the substring operator (\$) to learn if one string exists within another.

The *indexOf()* method is similar to the AT() function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the method's optional parameter specifies where to start searching instead of the *n*th occurrence to find.

Example The following function removes characters from a string by looking for the search string with the \$ operator and replacing it with nothing.

```
? strip( "banana", "an" )      // Displays "ba"

function strip( cArg, cStrip )
    local cRet, nLen
    cRet = cArg
    nLen = len( cStrip )
    do while cStrip $ cRet
        cRet := stuff( cRet, at( cStrip, cArg ), nLen, "" )
    enddo
    return cRet
```

All the loop needs know is whether the substring is in the main string, not where it is, so the \$ operator is slightly more convenient than using the AT() function. If the substring is in the main string, then STUFF() is used to remove it, using the position returned by AT(). The length of the substring is stored in a variable before the loop; it never changes so there's no need to get it repeatedly in the loop.

See also *indexOf(), RAT(), STUFF(), SUBSTR()*

CENTER()

Returns a character string that contains a string centered in a line of specified length.

Syntax CENTER(<expC> [, <length expN> [, <pad expC>]])

<expC> The text to center.

<length expN> The length of the resulting line of text. The default is 80 characters.

<pad expC> The single character to pad the string with if <length expN> is greater than the number of characters in <expC>. If <length expN> is equal to or less than the number of characters in <expC>, <pad expC> is ignored.

If <pad expC> is more than one character, CENTER() uses only the first character. If <pad expC> isn't specified, CENTER() pads with spaces.

Description CENTER() returns a character expression with the requisite number of leading and trailing spaces to center it in a line that is a specified number of characters wide.

To create the resulting string, CENTER() performs the following steps.

- Subtracts the length of <expC> or <memo field> from <length expN>
- Divides the result in half and rounds up if necessary
- Pads <expC> on either side with that number of spaces or the first character in <pad expC>

If the length of <expC> or <memo field> is greater than <length expN>, CENTER() does the following:

- Subtracts <length expN> from the length of <expC>
- Divides the result in half and rounds up if necessary
- Truncates both sides of <expC> by that many characters

When the result of subtracting the length of <expC> from <length expN> is an odd number, CENTER() pads one less space on the left if the difference is positive, or truncates one less character on the left if the difference is negative.

See Also LEN(), REPLICATE(), TRANSFORM()

charAt()

Returns the character at the specified position in the string.

Syntax `<expC>.charAt(<expN>)`

<expC> A string.

<expN> Index into the string, which is indexed from left to right. The first character of the string is at index 0 and the last character is at index `<expC>.length - 1`.

Property of String

Description `charAt()` returns the character in a String object at the specified index position. If the index position is after the last character in the string, `charAt()` returns an empty string.

See also `indexOf()`, `SUBSTR()`, `substring()`

CHR()

Returns the character equivalent of a specified ASCII value.

Syntax `CHR(<expN>)`

<expN> The numeric ASCII value, from 0 to 255, inclusive, whose character equivalent you want to return.

Description `CHR()` is the inverse of `ASC()`. `CHR()` accepts an ASCII value and returns its character, while `ASC()` accepts a character and returns its ASCII value.

See the ASCII table in Appendix A for a listing of ASCII values and their corresponding characters.

Other than the syntactic difference of being a method instead of a function, the `chr()` method behaves identically to the `CHR()` function.

Example Executing the following statements in the Command window demonstrates the relation between `ASC()` and `CHR()`:

```
? asc( "A" )           // 65
? chr( asc( "A" ) + 32 ) // "a"
? asc( chr( asc( "A" ) + 32 ) ) // 97
```

See also `ASC()`, `chr()`

chr()

Returns the character equivalent of a specified ASCII value.

Syntax <oRef>.chr(<expN>)

<oRef> A reference to a String object.

<expN> The numeric ASCII value, from 0 to 255, inclusive, whose character equivalent you want to return.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the CHR() function.

See also asc(), CHR()

DIFFERENCE()

Returns a number that represents the phonetic difference between two strings.

Syntax DIFFERENCE(<expC1>, <expC2>)

<expC1> The first character expression to evaluate the SOUNDEX() of and compare to the second value.

<expC2> The second character expression to evaluate the SOUNDEX() of and compare to the first value.

Description SOUNDEX() returns a four-character code that represents the phonetic value of a character expression. DIFFERENCE() compares the SOUNDEX() codes of two character expressions, and returns an integer from 0 to 4 that expresses the difference between the codes.

A returned value of 0 indicates the greatest difference in SOUNDEX() codes—the two expressions have no SOUNDEX() characters in common. A returned value of 4 indicates the least difference—the two expressions have all four SOUNDEX() characters in common. However, using DIFFERENCE() on short strings can produce unexpected results, as shown in the following example.

```
? soundex( "Mom" )                // Displays M500
? soundex( "Dad" )                // Displays D300
? soundex( "Mom", "Dad" )        // Displays 2
```

To compare the character-by-character similarity between two strings rather than the phonetic similarity, use LIKE().

Example The following example sets a filter in the current work area to show those records whose Title field sounds like the word typed in the entryfield *soundsLike*:

```
function showTitlesLike_onClick()
  private cArg
  cArg = form.soundsLike.value
  set filter to difference( TITLE, "&cArg" ) == 4    // Best matches only
```

Macro substitution is used to convert the value in the entryfield into a literal string for the filter expression.

See Also LIKE(), SOUNDEX()

getBytes()

Returns the value of the byte at the specified index in the string.

Syntax <oRef>.getBytes(<index expN>)

<oRef> A reference to the String object that you're using as a structure.

<index expN> The index number of the desired byte. The first byte is at index number zero.

Property of String

Description Strings in *Visual* dBASE are Unicode strings, which use double-byte characters. Use *getBytes()* when using a string as a structure that is passed to a DLL function that you have prototyped with EXTERN, to get the values of the bytes in the structure.

Example The GetClientRect() API function returns the coordinates of a window's client rectangle (the area inside the window borders) in a structure made up of four long integers: left, top, right, and bottom. A long integer is 32 bits, or 4 bytes. Therefore the rectangle structure is 16 bytes long. In a *Visual* dBASE string, each character is two bytes, so a string must be 8 characters long to store the rectangle structure.

The client rectangle coordinates are relative to the window's client area, so the left and top are always zero; the right and bottom coordinates contain the width and height of the client area. The following example displays the width of the client area of a default form in pixels:

```
if type( "GetClientRect" ) # "FP"
  extern CLOGICAL GetClientRect( CHANDLE, CPTR ) USER32
endif
c = space( 8 )           // 16-byte structure
f = new Form()
f.open()                 // Form must be open to have valid hWnd
if GetClientRect( f.hWnd, c )
  ? c.getBytes( 8 ) + bitlshift( c.getBytes( 9 ), 8 )
```

```
endif
f.close()
```

The `SPACE()` function is used to create a string of the desired length. The entire structure will be overwritten by the function, so it doesn't matter that the string initially contains spaces.

After the function call, `getByte()` is used to extract the coordinate, byte-by-byte. The right coordinate starts at offset 8 (the left starts at offset 0, the top at offset 4, and the bottom at offset 12) with the low byte first. The second byte is shifted 8 bits to the left and added to the first byte. Only the first two bytes are used, because the width will always be well under 64 K pixels, the maximum number that can be represented by two bytes (16 bits).

See also `setByte()`, `EXTERN`

indexOf()

Returns a number that represents the position of a string within another string.

Syntax `<target expC>.indexOf(<search expC> [, <from index expN>])`

<target expC> The string in which you want to search for `<search expC>`.

<search expC> The string you want to search for in `<target expC>`.

<from index expN> Where you want to start searching for the string. By default, *Visual dBASE* starts searching at the beginning of the string, index 0.

Property of String

Description This method is similar to the `AT()` function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the optional parameter specifies where to start searching instead of the *n*th occurrence to find.

`indexOf()` returns an index representing where a *search string* begins in a *target string*. The first character of the string is at index 0 and the last character is at index `<target expC>.length - 1`. `indexOf()` searches one character at a time from left to right, beginning to end, from the character at `<from index expN>` to the last character. The search is case-sensitive. Use `toUpperCase()` or `toLowerCase()` to make the search case-insensitive.

`indexOf()` returns -1 when

- The search string isn't found.
- The search string or target string is empty.
- The search string is longer than the target string.

Use `lastIndexOf()` to find the starting position of `<search expC>`, searching from *right to left*, end to beginning.

See also `AT()`, `lastIndexOf()`, `stuff()`, `substring()`

ISALPHA()

Returns *true* if the first character of a string is alphabetic.

Syntax `ISALPHA(<expC>)`

<expC> The string you want to test.

Description `ISALPHA()` tests the first character of the string and returns *true* if it's an alphabetic character. `ISALPHA()` returns *false* if the character isn't alphabetic or if the character expression is empty.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the `isAlpha()` method behaves identically to the `ISAPLHA()` function.

See also `isAlpha()`, `ISLOWER()`, `ISUPPER()`, `LOWER()`, `UPPER()`

isAlpha()

Returns *true* if the first character of a string is alphabetic.

Syntax `<expC>.isAlpha()`

<expC> The string you want to test.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the `ISAPLHA()` function.

See also `ISALPHA()`, `isLower()`, `isUpper()`, `toLowerCase()`, `toUpperCase()`

ISLOWER()

Returns *true* if the first character of a string is alphabetic and lowercase.

Syntax `ISLOWER(<expC>)`

<expC> The string you want to test.

Description ISLOWER() tests the first character of the string and returns *true* if it's a lowercase alphabetic character. ISLOWER() returns *false* if the character isn't lowercase or if the character expression is empty.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the *isLower*() method behaves identically to the ISLOWER() function.

See also ISALPHA(), *isLower*(), ISUPPER(), LDRIVER(), LOWER(), UPPER()

isLower()

Returns *true* if the first character of a string is alphabetic and lowercase.

Syntax <oRef>.isLower()
<expC> The string you want to test.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the ISLOWER() function.

See also *isAlpha*(), ISLOWER(), *isUpper*(), *toLowerCase*(), *toUpperCase*()

ISUPPER()

Returns *true* if the first character of a string is alphabetic and uppercase.

Syntax ISUPPER(<expC>)
<expC> The string you want to test.

Description ISUPPER() tests the first character of the string and returns *true* if it's an uppercase alphabetic character. ISUPPER() returns *false* if the character isn't uppercase or if the character expression is empty.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the *isUpper*() method behaves identically to the ISUPPER() function.

See also ISALPHA(), ISLOWER(), *isUpper*(), LOWER(), UPPER()

isUpper()

Returns *true* if the first character of a string is alphabetic and uppercase.

Syntax <expC>.isUpper()
expC The string you want to test.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the ISUPPER() function.

See also isAlpha(), isLower(), ISUPPER(), toLowerCase(), toUpperCase()

lastIndexOf()

Returns a number that represents the starting position of a string within another string. *lastIndexOf()* searches backward from the right end of the target string, and returns a value counting from the beginning of the target.

Syntax <target expC>.lastIndexOf(<search expC> [, <from index expN>])

<target expC> The string in which you want to search for <search expC>.

<search expC> The string you want to search for in <target expC>.

<from index expN> Where you want to start searching for the string. By default, Visual dBASE starts searching at the end of the string, index <target expC>.length – 1.

Property of String

Description This method is similar to the RAT() function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the optional parameter specifies where to start searching instead of the *n*th occurrence to find.

Use *lastIndexOf()* to search for the <search expC> in a target string, searching right to left, end to beginning, from the character at <from index expN> to the first character. The search is case-sensitive. Use *toUpperCase()* or *toLowerCase()* to make the search case-insensitive.

Even though the search starts from the end of the target string, *lastIndexOf()* returns an index representing where a *search string* begins in a *target string*, counting from the *beginning* of the target. The first character of the string is at index 0 and the last character is at index <target expC>.length – 1. If <search expC> occurs only once in the target, *lastIndexOf()* and *indexOf()* return the same value. For example, "abcdef".lastIndexOf("abc") and "abcdef".indexOf("abc") both return 0.

lastIndexOf() returns -1 when:

- The search string isn't found
- The search string or target string is empty
- The search string is longer than the target string

To find the starting position of *<search expC>*, searching from *left to right*, beginning to end, use *indexOf()*.

See also *indexOf()*, *RAT()*, *stuff()*, *substring()*, *toLowerCase()*, *toUpperCase()*

LEFT()

Returns a specified number of characters from the beginning of a string.

Syntax LEFT(*<expC>*, *<expN>*)

<expC> The string from which you want to extract characters.

<expN> The number of characters to extract from the beginning of the string.

Description Starting with the first character of a character expression, LEFT() returns *<expN>* characters. If *<expN>* is greater than the number of characters in the specified string, LEFT() returns the string as it is, without adding spaces to achieve the specified length. You can use LEN() to determine the actual length of the returned string.

If *<expN>* is less than or equal to zero, LEFT() returns an empty string. If *<expN>* is greater than or equal to zero, LEFT(*<expC>*, *<expN>*) achieves the same results as SUBSTR(*<expC>*, 1, *<expN>*).

When LEFT() returns characters from a memo field, it counts two characters for each carriage-return and linefeed combination (CR/LF).

Other than the syntactic difference of being a method instead of a function, the *left()* method behaves identically to the LEFT() function.

Example See REPLICATE()

See also AT(), LEN(), *left()*, RIGHT(), SUBSTR()

left()

Returns a specified number of characters from the beginning of a character string.

Syntax *<expC>.left(<expN>)*

<expC> The string from which you want to extract characters.

<expN> The number of characters to extract from the beginning of the string.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the LEFT() function.

See also *indexOf(), LEFT(), length, right(), substring()*

leftTrim()

Returns a string with no leading space characters.

Syntax `<expC>.leftTrim()`

<expC> The string from which you want to remove the leading space characters.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the LTRIM() function.

See also *left(), LTRIM(), right(), rightTrim(), substring()*

LEN()

Returns the number of characters in a specified character string.

Syntax `LEN(<expC>)`

<expC> The character string whose length you want to find.

Description LEN() returns the number of characters (the *length*) of a character string or memo field. The length of an empty character string or empty memo field is zero. When LEN() calculates the length of a memo field, it counts two characters for each carriage-return and linefeed combination (CR/LF).

Other than the syntactic difference of reading a property instead of calling a function, *length* contains the same value that LEN() returns.

See also *length, TRIM()*

length

The number of characters in a specified character string.

Syntax `<expC>.length`

<expC> The character string whose length you want to find.

Property of String

Description A string's *length* property reflects the number of characters (the *length*) of a character string. The length of an empty character string is zero.

length is a read-only property.

Other than the syntactic difference of reading a property instead of calling a function, *length* contains the same value that LEN() returns.

See also LEN(), *rightTrim*()

LIKE()

Returns *true* if a specified string matches a specified skeleton string.

Syntax LIKE(<*skeleton expC*>, <*expC*>)

<skeleton expC> A string containing a combination of characters and wildcards. The wildcards are ? and *.

<expC> The string to compare to the skeleton string.

Description Use LIKE() to compare one string to another. The <*skeleton expC*> argument contains wildcard characters and represents a pattern; the <*expC*> argument is compared to this pattern. LIKE() returns *true* if <*expC*> is a string that matches <*skeleton expC*>. To compare the phonetic similarity between two strings rather than the character-by-character similarity, use DIFFERENCE().

Use the wildcard characters ? and * to form the pattern for <*skeleton expC*>. An asterisk (*) stands for any number of characters, including zero characters. The question mark (?) stands for any single character. Both wildcards can appear anywhere and more than once in <*skeleton expC*>. Wildcard characters in <*skeleton expC*> can stand for uppercase or lowercase letters.

If * or ? appears in <*expC*>, they are interpreted as literal, not wildcard, characters, as shown in the following example.

```
? LIKE( "a*d", "abcd" ) // Displays true
? LIKE( "a*d", "aBCd" ) // Displays true
? LIKE( "abcd", "a*d" ) // Displays false
```

LIKE() is case-sensitive. Use UPPER() or LOWER() for case-insensitive comparisons with LIKE(). LIKE() returns *true* if both arguments are empty strings. LIKE() returns *false* if one argument is empty and the other isn't.

See Also AT(), DIFFERENCE()

LOWER()

Converts all uppercase characters in a string to lowercase and returns the resulting string.

Syntax LOWER(<expC>)

<expC> The string you want to convert to lowercase.

Description LOWER() converts the uppercase alphabetic characters in a character expression or memo field to lowercase. LOWER() ignores digits and other characters.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the *toLowerCase()* method behaves identically to the LOWER() function.

See also ISLOWER(), PROPER(), *toLowerCase()*, UPPER()

LTRIM()

Returns a string with no leading space characters.

Syntax LTRIM(<expC>)

<expC> The string from which you want to remove the leading space characters.

Description LTRIM() returns <expC> with no leading space characters.

To remove *trailing* space characters from a string or memo field, use RTRIM() or TRIM().

Other than the syntactic difference of being a method instead of a function, the *leftTrim()* method behaves identically to the LTRIM() function.

See also *leftTrim()*, RTRIM(), STR(), TRIM()

PROPER()

Converts a character string to proper-noun format and returns the resulting string.

Syntax PROPER(<expC>)

<expC> The character string to convert to proper-noun format.

Description PROPER() returns <expC> with the first character in each word capitalized and the remaining letters set to lowercase. PROPER() changes the first character of a word only if it is a lowercase alphabetic character.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the *toProperCase*() method behaves identically to the PROPER() function.

See also LOWER(), *toProperCase*(), UPPER()

RAT()

Returns a number that represents the starting position of a string within another string. RAT() searches backward from the right end of the target string, and returns a value counting from the beginning of the target.

Syntax RAT(<search expC>, <target expC> [, <nth occurrence expN>])

<search expC> The string you want to search for in <target expC>.

<target expC> The string in which to search for <search expC>.

<nth occurrence expN> Which occurrence of the string to find. By default, Visual dBASE searches for the first occurrence from the end. You can search for other occurrences by specifying the number (based on starting from the end), which must be greater than zero.

Description Use RAT() to search for the first or <nth occurrence expN> occurrence of <search expC> in a target string, searching right to left, end to beginning, from the last character to the first character. The search is case-sensitive. Use UPPER() or LOWER() to make the search case-insensitive.

Even though the search starts from the end of the target string or memo field, RAT() returns the numeric position where a *search string* begins in a *target string*, counting from the *beginning* of the target. If <search expC> occurs only once in the target, RAT() and AT() return the same value. For example, RAT("abc", "abcdef") and AT("abc", "abcdef") both return 1.

RAT() returns 0 when:

- The search string isn't found
- The search string or target string is empty
- The search string is longer than the target string
- The *n*th occurrence you specify with <nth occurrence expN> doesn't exist

To find the starting position of <search expC>, searching from *left to right*, beginning to end, use AT(). To learn if one string exists within another, use the substring operator (\$).

The *lastIndexOf()* method is similar to the *RAT()* function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the optional parameter specifies where to start searching instead of the *n*th occurrence to find.

Example Here is a simple file name extraction function that extracts a file name for a path by looking for the last backslash character with *RAT()*. Everything that follows in the string is extracted with *SUBSTR()*. If there is no backslash, the entire string is returned.

```
? getFileName( "C:\\WINDOWS\\SOL.EXE" )

function getFileName( cArg )
    local nPos
    nPos = rat( "\", cArg )
    return iif( n >= 0, substr( cArg, ++nPos ), cArg )
```

Note that the position returned by *RAT()* is incremented before it is passed to *SUBSTR()*. Otherwise, the last backslash would be returned as well.

See also *AT()*, *lastIndexOf()*, *LOWER()*, *STUFF()*, *SUBSTR()*, *UPPER()*

REPLICATE()

Returns a string repeated a specified number of times.

Syntax *REPLICATE*(*<expC>*, *<expN>*)

<expC> The string you want to repeat.

<expN> The number of times to repeat the string.

Description *REPLICATE()* returns a character string composed of a character expression repeated a specified number of times.

If the character expression is an empty string, *REPLICATE()* returns an empty string. If the number of repeats you specify for *<expN>* is 0 or less, *REPLICATE()* returns an empty string.

To repeat space characters, use *SPACE()*.

The *replicate()* method is almost identical to the *REPLICATE()* function, but in addition to the syntactic difference of being a method instead of a function, the repeat count is optional and defaults to 1.

Example The following function pads the left side of a string with a specified character to make the result at least as long as needed.

```
? padl( "Test", 7, "*" ) // Displays ***Test

function padl( cArg, nLen, cPad )
    if argcount() < 3
        cPad = ""
    endif
```

```
replicate ( )
```

```
return replicate( left( cPad + " ", 1 ), nLen - len( cArg ) ) + cArg
```

To make sure only one character is repeated, a space is added to the parameter (in case the parameter is an empty string or was omitted), and the LEFT() function is used to extract the first character (in case the parameter was more than one character).

See also *replicate()*, *SPACE()*

replicate()

Returns a string repeated a specified number of times.

Syntax *<oRef>.replicate(<expC> [, <expN>])*

<oRef> A reference to a String object.

<expC> The string you want to repeat.

<expN> The number of times to repeat the string; by default, 1.

Property of String

Description This method is almost identical to the REPLICATE() function, but in addition to the syntactic difference of being a method instead of a function, the repeat count is optional and defaults to 1.

See also REPLICATE(), *space()*

RIGHT()

Returns characters from the end of a character string.

Syntax RIGHT(<expC>, <expN>)

<expC> The string from which you want to extract characters.

<expN> The number of characters to extract from the string.

Description Starting with the last character of a character expression, RIGHT() returns a specified number of characters. If the number of characters you specify for <expN> is greater than the number of characters in the specified string or memo field, RIGHT() returns the string as is, without adding spaces to achieve the specified length. If <expN> is less than or equal to zero, RIGHT() returns an empty string.

Strings often have trailing blanks. You may want to remove them with TRIM() before using RIGHT().

When RIGHT() returns characters from a memo field, it counts two characters for each carriage-return and linefeed combination (CR/LF).

Other than the syntactic difference of being a method instead of a function, the *right()* method behaves identically to the RIGHT() function.

See also AT(), LEFT(), RAT(), *right()*, SUBSTR(), TRIM()

right()

Returns characters from the end of a character string.

Syntax *<expC>.right(<expN>)*

<expC> The string from which you want to extract characters.

<expN> The number of characters to extract from the string.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the RIGHT() function.

See also *indexOf()*, *left()*, *lastIndexOf()*, RIGHT(), *substring()*

rightTrim()

Returns a string with no trailing space characters.

Syntax *<expC>.rightTrim()*

<expC> The string from which you want to remove the trailing space characters.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the TRIM() function.

See also *left()*, *leftTrim()*, *right()*, *substring()*, TRIM()

RTRIM()

Returns a string with no trailing space characters.

Syntax RTRIM(<expC>)

setByte ()

<expC> The string from which you want to remove the trailing space characters.

Description RTRIM() is identical to TRIM(). See TRIM() for details.

See also LTRIM(), *rightTrim*(), TRIM()

setByte()

Assigns a new value to the byte at the specified index in the string.

Syntax <oRef>.setByte(<index expN>, <value expN>)

<oRef> A reference to the String object that you're using as a structure.

<index expN> The index number of the byte to set. The first byte is at index number zero.

<value expN> The new byte value, from 0 to 255.

Property of String

Description Strings in *Visual* dBASE are Unicode strings, which use double-byte characters. Use *setByte*() when using a string as a structure that is passed to a DLL function that you have prototyped with EXTERN, to set the values of the bytes in the structure.

The *length* of the structure string should be one-half the number of bytes in the structure, rounded up. Setting the individual bytes of a Unicode string will most likely cause the string to become unprintable.

Example Suppose you need to copy a string into a structure that is used by a function. The function expects the string to be composed of single-byte characters. *Visual* dBASE strings are double-byte, so you will need to use *setByte*() to copy each character of the string into the structure string, byte-by-byte.

The following function copies a string into a structure string at the specified offset, and pads the rest of the length in the structure with null characters.

```
function setStructString( cStruct, nIndex, cValue, nChars )
    if argcount() < 4
        nChars = len( cValue ) // Default length is length of string
    endif
    local n
    for n = 0 to min( len( cValue ), nChars ) - 1
        cStruct.setByte( nIndex + n, asc( cValue.charAt( n ) ) )
    endfor
    do while n < nChars // Pad length with null characters
        cStruct.setByte( nIndex + n++, 0 )
    enddo
```

In the FOR loop, the MIN() function is used to copy all the characters in the string, or the specified number of characters, whichever is less. This means that you can safely pass a string of any length to the function, and as long as you specify the correct length, you don't have to worry about the string being too long. Each character is extracted with the *charAt()* method and converted to its ASCII value (a number from 0 to 255) with the *ASC()* function.

See also *getBytes()*, *EXTERN*

SOUNDEX()

Returns a four-character string that represents the SOUNDEX (sound-alike) code of another string.

Syntax SOUNDEX(<expC>)

<expC> The string for which to calculate the soundex code.

Description SOUNDEX() returns a four-character code that represents the phonetic value of a character expression. The code is in the form "letter digit digit digit," where "letter" is the first alphabetic character in the expression being evaluated. The more phonetically similar two strings are, the more similar their SOUNDEX codes.

Use SOUNDEX() to find words that sound similar, or are spelled similarly, such as names like "Smith," "Smyth," and "Smythe." Using the U.S. language driver, these all evaluate to S531.

SOUNDEX() returns "0000" if the character expression is an empty string or if the first nonblank character isn't a letter. SOUNDEX() returns 0's for the first digit encountered and for all following characters, regardless of whether they're digits or alphabetic characters.

To compare the SOUNDEX values of two character expressions or memo fields, use DIFFERENCE(). If you want to compare the character-by-character similarity between two strings rather than the phonetic similarity, use LIKE().

SOUNDEX() is language driver-specific. If the current language driver is U.S., SOUNDEX() does the following to calculate the phonetic value of a string:

- Ignores leading spaces.
- Ignores the letters A, E, I, O, U, Y, H, and W.
- Ignores case.
- Converts the first nonblank character to uppercase and makes it the first character in the SOUNDEX code.
- Converts B, F, P, and V to 1.
- Converts C, G, J, K, Q, S, X, and Z to 2.
- Converts D and T to 3.

SPACE ()

- Converts L to 4.
- Converts M and N to 5.
- Converts R to 6.
- Removes the second occurrence of any adjacent letters that receive the same digits as phonetic values.
- Pads the end of the resulting string with zeros if fewer than three digits remain.
- Truncates the resulting string to three digits if more than three digits remain.
- Concatenates the first character of the code to the remaining three digits to create the "letter digit digit digit" soundex code.

Example To perform a sound-alike match for the Last_name field of a table, first create an index using SOUNDEX():

```
index on soundex( LAST_NAME ) tag LAST_SNDX
```

Then to perform the search, use SOUNDEX() on the search value entered, like this:

```
function searchButton_onClick()  
    set order to LAST_SNDX  
    if not seek( soundex( form.soundsLike.value ) )  
        msgbox( "No names similar", "Search failed" )  
    endif
```

See Also DIFFERENCE(), LIKE()

SPACE()

Returns a specified number of space characters.

Syntax SPACE(<expN>)

<expN> The number of spaces you want to return.

Description SPACE() returns a character string composed of a specified number of space characters. The space character is ASCII code 32.

If <expN> is 0 or less, SPACE() returns an empty string.

To create a string using a character other than the space character, use REPLICATE().

Other than the syntactic difference of being a method instead of a function, the *space()* method behaves identically to the SPACE() function.

See also REPLICATE(), *space()*

space()

Returns a specified number of space characters.

Syntax <oRef>.space(<expN>)

<oRef> A reference to a String object.

<expN> The number of spaces you want to return.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the SPACE() function.

See also replicate(), SPACE()

STR()

Returns the character string equivalent of a specified numeric expression.

Syntax STR(<expN> [, <length expN> [, <decimals expN> [, <expC>]]])

<expN> The numeric expression to return as a character string.

<length expN> The length of the character string to return. The valid range is 1 to 20, inclusive, and includes a decimal point, decimal digits, and minus sign characters. The default is 10. If <length expN> is smaller than the number of integer digits in <expN>, STR() returns asterisks (*).

<decimals expN> The number of characters to reserve for decimal digits. The default and lowest allowable value is 0. If you do not specify a value for <decimals expN>, STR() rounds <expN> to the nearest whole number. If you want to specify a value for <decimals expN>, you must also specify a value for <length expN>.

<expC> The character to pad the beginning of the returned character string with when the length of the returned string is less than <length expN> digits long. The default pad character is a space. If you want to specify a value for <expC>, you must also specify values for <length expN> and <decimals expN>. You can specify more than one character for <expC>, but STR() uses only the first one.

Description Use STR() to convert a number to a string, so you can manipulate it as characters. For example, you can index on a numeric field in combination with a character field by converting the numeric field to character with STR().

Visual dBASE rounds and pads numbers to fit within parameters you set with <length expN> and <decimals expN>, following these rules:

- If *<decimals expN>* is smaller than the number of decimals in *<expN>*, STR() rounds to the most accurate number that will fit in *<length expN>*. For example, STR(10.765,5,1) returns " 10.8" (with a single leading space), and STR(10.765,5,2) returns "10.77".
- If *<length expN>* isn't large enough for *<decimals expN>* number of decimal places, STR() rounds *<expN>* to the most accurate number that will fit in *<length expN>*. For example, STR(10.765,4,3) returns "10.8".
- If *<decimals expN>* is larger than the number of decimals in *<expN>*, and *<length expN>* is larger than the returned string, STR() adds zeros (0) to the end of the returned string. Visual dBASE only adds enough zero to bring the number of decimal digits to a maximum of *<decimals expN>*.
- If the returned string is still shorter than *<length expN>*, dBASE pads the left to fill to the length of *<length expN>*. For example, STR(10.765,8,6) returns "10.76500" for a returned length of 8; STR(10.765,7,6) returns "10.7650" for a returned length of 7; and STR(10.765,12,6) returns " 10.765000" (with three leading spaces) for a returned length of 12.

To remove the leading spaces created by STR(), use LTRIM(). If you concatenate a number to a string with the + or - operators, Visual dBASE automatically converts the number to a string, using the number of decimal places specified by SET DECIMALS, and removes the leading spaces.

See Also LTRIM(), VAL()

STUFF()

Returns a string with specified characters removed and others inserted in their place.

Syntax STUFF(*<target expC>*, *<start expN>*, *<quantity expN>*, *<replacement expC>*)

<target expC> The string you want to remove characters from and replace with new characters.

<start expN> The character position in the string at which you want to start removing characters.

<quantity expN> The number of characters you want to remove from the string.

<replacement expC> The characters you want to insert in the string.

Description STUFF() returns a target character expression with a replacement character string inserted at a specified position. Starting at the position you specify, *<start expN>*, STUFF() removes a specified number, *<quantity expN>*, of characters from the original string.

If the target character expression is an empty string, STUFF() returns the replacement string.

If *<start expN>* is less than or equal to 0, STUFF() treats *<start expN>* as 1. If *<quantity expN>* is less than or equal to 0, STUFF() inserts the replacement string at position *<start expN>* without removing any characters from the target.

If *<start expN>* is greater than the length of the target, STUFF() doesn't remove any characters and appends the replacement string to the end of the target.

If the replacement string is empty, STUFF() removes the characters specified by *<quantity expN>* from the target, starting at *<start expN>*, without adding characters.

The *stuff()* method is almost identical to the STUFF() function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the replacement string is optional, and defaults to an empty string.

See also AT(), LEFT(), RAT(), REPLICATE(), RIGHT(), SPACE(), STUFF(), SUBSTR()

stuff()

Returns a string with specified characters removed and others inserted in their place.

Syntax *<expC>.stuff(<start expN>, <quantity expN> [, <replacement expC>])*

<expC> The string in which you want to remove and replace characters.

<start expN> The character position in the string at which you want to start removing characters.

<quantity expN> The number of characters you want to remove from the string.

<replacement expC> The characters you want to insert in the string. By default, this is an empty string.

Property of String

Description This method is almost identical to the STUFF() function. However, the *stuff()* method is zero-based (the function is one-based), a replacement string is optional, and the method defaults to an empty string.

See also *indexOf(), left(), lastIndexOf(), replicate(), right(), space(), STUFF(), substring()*

SUBSTR()

Returns a substring derived from a specified character string.

Syntax SUBSTR(*<expC>*, *<start expN>* [, *<length expN>*])

<expC> The string you want to extract characters from.

<start expN> The character position in the string to start extracting characters.

<length expN> The number of characters to extract from the string.

Description Starting in a character expression at the position you specify for <start expN>, SUBSTR() returns the number of characters you specify for <length expN>. If <start expN> is greater than the length of <expC>, or <length expN> is zero or a negative number, SUBSTR() returns an empty string.

If you don't specify <length expN>, SUBSTR() returns all characters starting from position <start expN> to the end of the string. If <length expN> is greater than the number of characters from <start expN> to the end of the string, SUBSTR() returns only as many characters as are left in the string, without adding space characters to achieve the specified length. You can use LEN() to determine the actual length of the returned string.

When SUBSTR() returns characters from a memo field, it counts two characters for each carriage-return and linefeed combination (CR/LF) in the memo field.

The *substring()* method is similar to the SUBSTR() function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the method takes a starting and ending position, while the function takes a start position and the number of character to extract.

Example See RAT()

See also AT(), LEFT(), LEN(), RAT(), RIGHT(), STUFF(), *substring()*.

substring()

Returns a substring derived from a specified character string.

Syntax <expC>.substring(<index1 expN>, <index2 expN>)

<expC> The string you want to extract characters from.

<index1 expN>, <index2 expN> Indexes into the string, which is indexed from left to right. The first character of the string is at index 0 and the last character is at index <expC>.length - 1.

Property of String

Description This method is similar to the SUBSTR() function, but in addition to the syntactic difference of being a method instead of a function and the fact that the position is zero-based, the method takes a starting and ending position, while the function takes a start position and the number of character to extract.

<index1 expN> and <index2 expN> determine the position of the substring to extract. *substring()* begins at the lesser of the two indexes and extracts up to the

character before the other index. If the two indexes are the same, *substring()* returns an empty string.

If the starting index is after the last character in the string, *substring()* returns an empty string.

See also *indexOf()*, *left()*, *length*, *toProperCase()*, *lastIndexOf()*, *right()*, *stuff()*, *SUBSTR()*

toLowerCase()

Converts all uppercase characters in a string to lowercase and returns the resulting string.

Syntax *<expC>.toLowerCase()*

<expC> The string you want to convert to lowercase.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the LOWER() function.

See also *isLower()*, *isUpper()*, LOWER(), *toProperCase()*, *toUpperCase()*

toProperCase()

Converts a character string to proper-noun format and returns the resulting string.

Syntax *<expC>.toProperCase()*

<expC> The string you want to convert to proper-noun format.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the PROPER() function.

See also PROPER(), *toLowerCase()*, *toUpperCase()*

toUpperCase()

Converts all lowercase characters in a string to uppercase and returns the resulting string.

Syntax *<expC>.toUpperCase()*

TRANSFORM()

<expC> The string you want to convert to uppercase.

Property of String

Description Other than the syntactic difference of being a method instead of a function, this method behaves identically to the UPPER() function.

See also *isLower(), isUpper(), toLowerCase(), toProperCase(), UPPER()*

TRANSFORM()

Applies a formatting template to an expression, returning a formatted string.

Syntax TRANSFORM(<exp>, <picture expC>)

<exp> The expression to be formatted.

<picture expC> The string containing the template characters necessary to format <exp>. The template characters are the same characters used in the *picture* property of an entryfield.

Description TRANSFORM() returns an expression in the template format you indicate with <picture expC>.

Example Suppose you store phone numbers as ten digits only to save storage space. You can display the number with the usual characters using a formatting template:

```
? transform( "8005551234", "@R (999) 999-9999" ) // Displays "(800) 555-1234"
```

Negative numbers can be displayed enclosed in parentheses:

```
? transform( -45, "@( 9999" ) // Displays "( -45)"  
? transform( 123, "@( 9999" ) // Displays " 123 "
```

See Also *picture* (page 15-148), STR()

TRIM()

Returns a string with no trailing space characters.

Syntax TRIM(<expC>)

<expC> The string from which you want to remove the trailing space characters.

Description TRIM() returns a character expression with no trailing space characters. TRIM() is identical to RTRIM().

To remove trailing blanks before concatenating a string to another string, use the - operator instead of the + operator.

Warning Do not create index expression with TRIM() that result in key values that vary in length from record to record. This results in unbalanced indexes that may become corrupted. Use the - operator, which relocates trailing blanks without changing the resulting length of the concatenated string.

To remove *leading* space characters from a string, use LTRIM().

Other than the syntactic difference of being a method instead of a function, the *rightTrim*() method behaves identically to the TRIM() function.

See also LEFT(), LTRIM(), RIGHT(), *rightTrim*()

UPPER()

Converts all lowercase characters in a string to uppercase and returns the resulting string.

Syntax UPPER(<expC>)

<expC> The character string you want to convert to uppercase.

Description UPPER() converts the lowercase alphabetic characters in a character expression or memo field to uppercase. UPPER() ignores digits and other characters.

The current language driver defines the character values that are lowercase and uppercase alphabetic. In a U.S. language driver, a lowercase alphabetic character is from a to z, and an uppercase alphabetic character is from A to Z.

Other than the syntactic difference of being a method instead of a function, the *toUpperCase*() method behaves identically to the UPPER() function.

Example UPPER() is frequently used to make searches case-insensitive. First you create an index using UPPER(), for example:

```
index on upper( LAST_NAME - " " - FIRST_NAME ) tag FULL_NAME
```

Then when searching, convert the search value to uppercase to match:

```
seek upper( form.search.value )
```

See also ISLOWER(), ISUPPER(), LOWER(), PROPER(), *toUpperCase*()

VAL()

Returns the number at the beginning of a character string.

Syntax VAL(<expC>)

<expC> The character expression that contains the number.

VAL ()

Description Use VAL() to convert a string that contains a number into an actual number. Once you convert a string to a number, you can perform arithmetic operations with it.

If the character string you specify contains both letters and numbers, VAL() returns the value of the entire number to the left of the first nonnumeric character. If the string contains a nonnumeric character other than a blank space in the first position, VAL() returns 0. For example, VAL("ABC123ABC456") returns 0, VAL("123ABC456ABC") returns 123, and VAL(" 123") also returns 123.

See Also STR()

Math / Money

Visual dBASE supports a wide range of mathematic, trigonometric, and financial functions.

ABS()

Returns the absolute value of a specified number.

Syntax `ABS(<expN>)`

<expN> The number whose absolute value you want to return.

Description `ABS()` returns the absolute value of a number. The absolute value of a number represents its magnitude. Magnitude is always expressed as a positive value, so the absolute value of a negative number is its positive equivalent.

See also `CEILING()`, `FLOOR()`, `INT()`, `ROUND()`

ACOS()

Returns the inverse cosine (arccosine) of a number.

Syntax `ACOS(<expN>)`

<expN> The cosine of an angle, from -1 to +1.

Description `ACOS()` returns the radian value of the angle whose cosine is <expN>. `ACOS()` returns a number from 0 to pi radians. `ACOS()` returns zero when <expN> is 1. For values of x from 0 to pi, `ACOS(y)` returns x if `COS(x)` returns y .

To convert the returned radian value to degrees, use RTOD(). For example, if the default number of decimal places is 2, ACOS(.5) returns 1.05 radians while RTOD(ACOS(.5)) returns 60.00 degrees.

Use SET DECIMALS to set the number of decimal places ACOS() displays.

To find the arcsecant of a value, use the arccosine of 1 divided by the value. For example, the arcsecant of 2 is ACOS(1/2), or 1.05 radians.

See also ASIN(), ATAN(), ATN2(), COS(), DTOR(), RTOD(), SET DECIMALS

ASIN()

Returns the inverse sine (arcsine) of a number.

Syntax ASIN(<expN>)

<expN> The sine of an angle, from -1 to +1.

Description ASIN() returns the radian value of the angle whose sine is <expN>. ASIN() returns a number from $-\pi/2$ to $\pi/2$ radians. ASIN() returns zero when <expN> is 0. For values of x from $-\pi/2$ to $\pi/2$, ASIN(y) returns x if SIN(x) returns y .

To convert the returned radian value to degrees, use RTOD(). For example, if the default number of decimal places is 2, ASIN(.5) returns .52 radians while RTOD(ASIN(.5)) returns 30.00 degrees.

Use SET DECIMALS to set the number of decimal places ASIN() displays.

To find the arccosecant of a value, use the arcsine of 1 divided by the value. For example, the arccosecant of 1.54 is ASIN(1/1.54), or .71 radians.

See also ACOS(), ATAN(), ATN2(), DTOR(), RTOD(), SET DECIMALS, SIN()

ATAN()

Returns the inverse tangent (arctangent) of a number.

Syntax ATAN(<expN>)

<expN> Any positive or negative number representing the tangent of an angle.

Description ATAN() returns the radian value of the angle whose tangent is <expN>. ATAN() returns a number from $-\pi/2$ to $\pi/2$ radians. ATAN() returns 0 when <expN> is 0. For values of x from $-\pi/2$ to $\pi/2$, ATAN(y) returns x if TAN(x) returns y .

To convert the returned radian value to degrees, use RTOD(). For example, if the default number of decimal places is 2, ATAN(1) returns 0.79 radians, while RTOD(ATAN(1)) returns 45.00 degrees.

Use SET DECIMALS to set the number of decimal places ATAN() displays.

ATAN() differs from ATN2() in that ATAN() takes the tangent as the argument, but ATN2() takes the sine and cosine as the arguments.

To find the arccotangent of a value, subtract the arctangent of the value from $\pi/2$. For example, the arccotangent of 1.73 is $\pi/2 - \text{ATAN}(1.73)$, or .52.

See also ACOS(), ASIN(), ATN2(), RTOD(), SET DECIMALS, TAN()

ATN2()

Returns the inverse tangent (arctangent) of a given point.

Syntax ATN2(<sine expN>, <cosine expN>)

<sine expN> The sine of an angle. If <sine expN> is 0, <cosine expN> can't also be 0.

<cosine expN> The cosine of an angle. If <cosine expN> is 0, <sine expN> can't also be 0. When <cosine expN> is 0 and <sine expN> is a positive or negative (nonzero) number, ATN2() returns $+\pi/2$ or $-\pi/2$, respectively.

Description ATN2() returns the angle size in radians when you specify the sine and cosine of the angle. ATN2() returns a number from $-\pi$ to $+\pi$ radians. ATN2() returns 0 when <sine expN> is 0. When you specify 0 for both arguments, Visual dBASE returns an error.

To convert the returned radian value to degrees, use RTOD(). For example, if the default number of decimal places is 2, ATN2(1,0) returns 1.57 radians while RTOD(ATN2(1,0)) returns 90.00 degrees.

Use SET DECIMALS to set the number of decimal places ATN2() displays.

ATN2() differs from ATAN() in that ATN2() takes the sine and cosine as the arguments, but ATAN() takes the tangent as the argument. See ATAN() for instructions on finding the arccotangent.

See also ACOS(), ASIN(), ATAN(), COS(), RTOD(), SET DECIMALS, SIN(), TAN()

CEILING()

Returns the nearest integer that is greater than or equal to a specified number.

Syntax CEILING(<expN>)

<expN> A number, from which to determine and return the integer that is greater than or equal to it.

Description CEILING() returns the nearest integer that is greater than or equal to <expN>; in effect, rounding positive numbers up and negative numbers down towards zero. If you pass a number with any digits other than 0 as decimal digits, CEILING() returns the nearest integer that is greater than the number. If you pass an integer to CEILING(), or a number with only 0s for decimal digits, it returns that number.

For example, if the default number of decimal places is 2,

- CEILING(2.10) returns 3.00
- CEILING(-2.10) returns -2.00
- CEILING(2.00) returns 2.00
- CEILING(2) returns 2
- CEILING(-2.00) returns -2.00

Use SET DECIMALS to set the number of decimal places CEILING() displays.

See the table in the description of INT() that compares INT(), FLOOR(), CEILING(), and ROUND().

See also FLOOR(), INT(), ROUND(), SET DECIMALS

COS()

Returns the trigonometric cosine of an angle.

Syntax COS(<expN>)

<expN> The size of the angle in radians. To convert an angle's degree value to radians, use DTOR(). For example, to find the cosine of a 30-degree angle, use COS(DTOR(30)).

Description COS() calculates the ratio between the side adjacent to an angle and the hypotenuse in a right triangle. COS() returns a number from -1 to +1. COS() returns 0 when <expN> is $\pi/2$ or $3\pi/2$ radians.

Use SET DECIMALS to set the number of decimal places COS() displays.

The secant of an angle is the reciprocal of the cosine of the angle. To return the secant of an angle, use $1/\text{COS}()$.

See also ACOS(), DTOR(), PI(), RTOD(), SET DECIMALS, SIN(), TAN()

DTOR()

Returns the radian value of an angle whose measurement is given in degrees.

Syntax DTOR(<expN>)

<expN> A negative or positive number that is the size of the angle in degrees.

Description DTOR() converts the measurement of an angle from degrees to radians. To convert degrees to radians, *Visual dBASE*

- Multiplies the number of degrees by pi
- Divides the result by 180
- Returns the quotient

A 180-degree angle is equivalent to pi radians.

Use DTOR() in the trigonometric functions SIN(), COS(), and TAN() because these functions require the angle value in radians. For example, to find the sine of a 45-degree angle, use SIN(DTOR(45)), which returns .71 if the default number of decimal places is 2.

Use SET DECIMALS to set the number of decimal places DTOR() displays.

See also ACOS(), ASIN(), ATAN(), ATN2(), COS(), PI(), RTOD(), SET DECIMALS, SIN(), TAN()

EXP()

Returns e raised to a specified power.

Syntax EXP(<expN>)

<expN> The positive, negative, or zero power (exponent) to raise the number e to.

Description EXP() returns a number equal to e (the base of the natural logarithm) raised to the <expN> power. For example, EXP(2) returns 7.39 because $e^2 = 7.39$.

EXP() is the inverse of LOG(). In other words, if $y = \text{EXP}(x)$, then $\text{LOG}(y) = x$.

Use SET DECIMALS to set the number of decimal places EXP() displays.

See also LOG(), LOG10(), SET DECIMALS

FLOOR()

Returns the nearest integer that is less than or equal to a specified number.

Syntax FLOOR(<expN>)

<expN> A number from which to determine and return the integer that is less than or equal to it.

Description FLOOR() returns the nearest integer that is less than or equal to *<expN>*; in effect, rounding positive numbers down and negative numbers up away from zero. If you pass a number with any digits other than zero (0) as decimal digits, FLOOR() returns the nearest integer that is less than the number. If you pass an integer to FLOOR(), or a number with only zeros for decimal digits, it returns that number.

For example, if the default number of decimal places is 2,

- FLOOR(2.10) returns 2.00
- FLOOR(-2.10) returns -3.00
- FLOOR(2.00) returns 2.00
- FLOOR(2) returns 2
- FLOOR(-2.00) returns -2.00

Use SET DECIMALS to set the number of decimal places FLOOR() displays.

When you pass a positive number to it, FLOOR() operates exactly like INT(). See the table in the description of INT() that compares INT(), FLOOR(), CEILING(), and ROUND().

See also CEILING(), INT(), ROUND(), SET DECIMALS

FV()

Returns the future value of an investment.

Syntax FV(*<payment expN>*, *<interest expN>*, *<term expN>*)

<payment expN> The amount of the periodic payment. Specify the payment in the same time increment as the interest and term. The payment can be negative or positive.

<interest expN> The interest rate per period expressed as a positive decimal number. Specify the interest rate in the same time increment as the payment and term.

<term expN> The number of payments. Specify the term in the same time increment as the payment and interest.

Description Use FV() to calculate the amount realized (future value) after equal periodic payments (deposits) at a fixed interest rate. FV() returns a float representing the total of the payments plus the interest generated and compounded.

Express the interest rate as a decimal. For example, if the annual interest rate is 9.5%, *<interest expN>* is .095 (9.5/100) for payments made annually.

Express *<payment expN>*, *<interest expN>*, and *<term expN>* in the same time increment. For example, if the payment is monthly, express the interest rate per month, and the number of payments in months. You would express an annual interest rate of 9.5%, for example, as .095/12, which is 9.5/100 divided by 12 months.

The formula *Visual dBASE* uses to calculate FV() is as follows:

$$fv = pmt * \frac{1 + int^{term} - 1}{int}$$

where int = rate / 100

For the future value an investment of \$350 made monthly for five years, earning 9% interest, the formula expressed as a *Visual dBASE* expression looks like this:

```
? FV(350, .09/12, 60)           // Returns 26398.45
? 350*((1+.09/12)^60-1)/(.09/12) // Returns 26398.45
```

In other words, if you invest \$350/month for the next five years into an account that pays an annual interest rate of 9%, at the end of five years you will have \$26398.45.

Use SET DECIMALS to set the number of decimal places FV() displays.

See Also PAYMENT(), PV(), SET DECIMALS

INT()

Returns the integer portion of a specified number.

Syntax INT(<expN>)
<expN> A number whose integer value you want to determine and return.

Description Use INT() to remove the decimal digits of a number and retain only the integer portion, the whole number.
If you pass a number with decimal places to a function, command, or method that uses an integer as an argument, such as SET EPOCH, *Visual dBASE* automatically truncates that number, in which case you don't need to use INT().

The following table compares INT(), FLOOR(), CEILING(), and ROUND(). (In these examples, the value of the second ROUND() argument is 0.)

| <expN> | INT() | FLOOR() | CEILING() | ROUND() |
|--------|--------|----------|------------|----------|
| 2.56 | 2 | 2 | 3 | 3 |
| -2.56 | -2 | -3 | -2 | -3 |
| 2.45 | 2 | 2 | 3 | 2 |
| -2.45 | -2 | -3 | -2 | -2 |

See also ABS(), CEILING(), FLOOR(), ROUND()

LOG()

Returns the logarithm to the base *e* (natural logarithm) of a specified number.

Syntax LOG(<expN>)

<expN> A positive nonzero number that equals e raised to the log. If you specify 0 or a negative number for <expN>, Visual dBASE generates an error.

Description LOG() returns the natural logarithm of <expN>. The natural logarithm is the power (exponent) to which you raise the mathematical constant e to get <expN>. For example, LOG(5) returns 1.61 because $e^{1.61} = 5$.

LOG() is the inverse of EXP(). In other words, if $\text{LOG}(y) = x$, then $y = \text{EXP}(x)$.

Use SET DECIMALS to set the number of decimal places LOG() displays.

See also EXP(), LOG10(), SET DECIMALS

LOG10()

Returns the logarithm to the base 10 of a specified number.

Syntax LOG10(<expN>)

<expN> A positive nonzero number which equals 10 raised to the log. If you specify 0 or a negative number for <expN>, Visual dBASE returns an error.

Description LOG10() returns the common logarithm of <expN>. The common logarithm is the power (exponent) to which you raise 10 to get <expN>. For example, LOG10(100) returns 2 because $10^2 = 100$.

Use SET DECIMALS to set the number of decimal places LOG10() displays.

See also EXP(), LOG(), SET DECIMALS

MAX()

Compares two numbers (or two date, character, or logical expressions) and returns the greater value.

Syntax MAX(<exp1>, <exp2>)

<exp1> A numeric, date, character, or logical expression to compare to a second expression of the same type.

<exp2> The second expression to compare to <exp1>.

Description Use MAX() to compare two numbers to determine the greater of the two values. You can use MAX() to ensure that a number is not less than a particular limit.

MAX() may also be used to compare two dates, character strings, or logical values, in which case MAX() returns:

- The later of the two dates. In *Visual* dBASE, a blank date is considered later than a non-blank date.
- The character string with the higher *collation value*. Collation values are determined by the language driver in use, and are case-sensitive. For example, with the DB437US driver, the letter “B” is higher than the letter “A”, but “a” is higher than “B” (all lowercase letters are collated higher than uppercase letters).
- *true* if one or both logical expressions evaluate to *true*. (The logical OR operator has the same effect.)

If <exp1> and <exp2> are equal, MAX() returns their value.

See also CALCULATE, IIF(), MIN()

MIN()

Compares two numbers (or two date, character, or logical expressions) and returns the lesser value.

Syntax MIN(<exp1>, <exp2>)

<exp1> A numeric, date, character, or logical expression to compare to a second expression of the same type.

<exp2> The second expression to compare to <exp1>.

Description Use MIN() to compare two numbers to determine the lesser of the two values. You can use MIN() to ensure that a number is not greater than a particular limit.

MIN() may also be used to compare two dates, character strings, or logical values, in which case MIN() returns:

- The earlier of the two dates. In *Visual* dBASE, a non-blank date is considered earlier than a blank date.
- The character string with the lower *collation value*. Collation values are determined by the language driver in use, and are case-sensitive. For example, with the DB437US driver, the letter “a” is lower than the letter “b”, but “B” is lower than “a” (all uppercase letters are collated lower than lowercase letters).
- *false* if one or both logical expressions evaluate to *false*. (The logical AND operator has the same effect.)

If <exp1> and <exp2> are equal, MIN() returns their value.

See also CALCULATE, IIF(), MAX()

MOD()

Returns the modulus (remainder) of one number divided by another.

Syntax MOD(<dividend expN>, <divisor expN>)

<dividend expN> The number to be divided.

<divisor expN> The number to divide by.

Description MOD() divides <dividend expN> by <divisor expN> and returns the remainder. In other words, MOD(X,Y) returns the remainder of x/y .

The modulus formula is

$$\text{<dividend>} - \text{INT}(\text{<dividend>/<divisor>}) * \text{<divisor>}$$

where INT() truncates a number to its integer portion.

Note Earlier versions of dBASE used FLOOR() instead of INT() in the modulus calculation. This change only affects the result if <dividend expN> and <divisor expN> are not the same sign, which in itself is an ambiguous case.

The % symbol is also used as the modulus operator. It performs the same function as MOD(). For example, the following two expressions are identical:

```
mod( x, 2 )
x % 2
```

See Also CEILING(), FLOOR(), INT()

PAYMENT()

Returns the periodic amount required to repay a debt.

Syntax PAYMENT(<principal expN>, <interest expN>, <term expN>)

<principal expN> The original amount to be repaid over time.

<interest expN> The interest rate per period expressed as a positive decimal number. Specify the interest rate in the same time increment as the term.

<term expN> The number of payments. Specify the term in the same time increment as the interest.

Description Use PAYMENT() to calculate the periodic amount (payment) required to repay a loan or investment of <principal expN> amount in <term expN> payments. PAYMENT() returns a number based on a fixed interest rate compounding over a fixed length of time.

If <principal expN> is positive, PAYMENT() returns a positive number.

If <principal expN> is negative, PAYMENT() returns a negative number.

Express the interest rate as a decimal. For example, if the annual interest rate is 9.5%, *<interest expN>* is .095 (9.5/100) for payments made annually.

Express *<interest expN>* and *<term expN>* in the same time increment. For example, if the payments are monthly, express the interest rate per month, and the number of payments in months. You would express an annual interest rate of 9.5%, for example, as .095/12, which is 9.5/100 divided by 12 months.

The formula *Visual* dBASE uses to calculate PAYMENT() is as follows:

$$\text{pmt} = \text{princ} * \frac{\text{int} * (1 + \text{int})^{\text{term}}}{(1 + \text{int})^{\text{term}} - 1}$$

where int = rate/100

For the monthly payment required to repay a principal amount of \$16860.68 in five years, at 9% interest, the formula expressed as a *Visual* dBASE expression looks like this:

```
? PAYMENT(16860.68,.09/12,60)           // Returns 350.00
nTemp = (1 + .09/12)^60
? 16860.68*(.09/12*nTemp)/(nTemp-1) // Returns 350.00
```

Use SET DECIMALS to set the number of decimal places PAYMENT() displays.

See Also FV(), PV(), SET DECIMALS

PI()

Returns the approximate value of pi, the ratio of a circle's circumference to its diameter.

Syntax PI()

Description PI() returns a number that is approximately 3.141592653589793. pi is a constant that can be used in mathematical calculations. For example, use it to calculate the area and circumference of a circle or the volume of a cone or cylinder.

Use SET DECIMALS to set the number of decimal places PI() displays.

See also COS(), DTOR(), RTOD(), SET DECIMALS, SIN(), TAN()

PV()

Returns the present value of an investment.

Syntax PV(<payment expN>, <interest expN>, <term expN>)

<payment expN> The amount of the periodic payment. Specify the payment in the same time increment as the interest and term. The payment can be negative or positive.

<interest expN> The interest rate per period expressed as a positive decimal number. Specify the interest rate in the same time increment as the payment and term.

<term expN> The number of payments. Specify the term in the same time increment as the payment and interest.

Description PV() is a financial function that calculates the original principal balance (present value) of an investment. PV() returns a float that is the amount to be repaid with equal periodic payments at a fixed interest rate compounding over a fixed length of time. For example, use PV() if you want to know how much you need to invest now to receive regular payments for a specified length of time.

Express the interest rate as a decimal. For example, if the annual interest rate is 9.5%, <interest expN> is .095 (9.5 / 100) for payments made annually.

Express <payment expN>, <interest expN>, and <term expN> in the same time increment. For example, if the payment is monthly, express the interest rate per month, and the number of payments in months. Express an annual interest rate of 9.5%, for example, as .095/12, which is 9.5/100 divided by 12 months.

The formula *Visual* dBASE uses to calculate PV() is as follows:

$$pv = pmt * \frac{(1 + int)^{term} - 1}{int * (1 + int)^{term}}$$

where int = rate 100

For the present value of an investment earning 9% interest, to be paid at \$350 monthly for five years, the formula expressed as a *Visual* dBASE expression looks like this:

```
? PV(350,.09/12,60)           // Returns 16860.68
nTemp = (1 + .09/12)^60
? 350*(nTemp-1)/(.09/12*nTemp) // Returns 16860.68
```

In other words, you have to invest \$16,860.68 now into an account paying an interest rate of 9% annually to receive \$350/month for the next five years.

Use SET DECIMALS to set the number of decimal places PV() displays.

See Also FV(), PAYMENT(), SET DECIMALS

RANDOM()

Returns a pseudo-random number between 0 and 1 exclusive (never 0 and never 1).

Syntax RANDOM([<expN>])

<expN> The number with which you want to seed RANDOM().

Description Computers cannot generate truly random numbers, but you can use RANDOM() to generate a series of numbers that appear to have a random distribution. A series of pseudo-random numbers relies on a seed value, which determines the exact numbers that appear in the series. If you use the same seed value, you get the same series of numbers.

Pseudo-random numbers, when considered as a whole series, appear to be random; that is, you cannot tell from one number what the next will be. But the first number in the series is related to the seed value. Therefore, you should seed RANDOM() only once at the beginning of each series, like before simulating a card shuffle or randomly assigning work shifts. Seeding during a series defeats the design of the random number generator.

If you specify a positive <expN> value, RANDOM() uses that <expN> as the seed value, so a positive value should be used for testing, since the numbers will be the same each time. If <expN> is negative, RANDOM() uses a seed value based on the number of seconds past midnight on your computer system clock. As a result, a negative <expN> value most likely will give you a different series of random numbers each time.

If you don't specify <expN>, or use zero, RANDOM() returns the next number in the series.

When *Visual* dBASE first starts up, the random number generator is seeded with a fixed internal seed value of 179757.

Use SET DECIMALS to set the number of decimal places RANDOM() displays.

See also GENERATE, SET DECIMALS

ROUND()

Returns a specified number rounded to the nearest integer or a specified number of decimal places.

Syntax ROUND(<expN 1> [, <expN 2>])

<expN 1> The number you want to round.

<expN 2> If <expN 2> is positive, the number of decimal places to round <expN 1> to. If <expN 2> is negative, whether to round <expN 1> to the nearest tens, hundreds, thousands, and so on.

Description Use ROUND() to round a number to a specified number of decimal places or to a specified tens, hundreds, thousands value, and so forth. Use ROUND() with SET DECIMALS to round a number *and* remove trailing zeros.

If the digit in position `<expN 2> + 1` is between 0 and 4 inclusive, `<expN 1>` (with `<expN 2>` decimal places) remains the same; if the digit in position `<expN 2> + 1` is between 5 and 9 inclusive, the digit in position `<expN 2>` is increased by 1.

Use 0 as `<expN 2>` to round a number to the nearest whole number. Using `-1` rounds a number to the nearest multiple of ten; rounding to a `-2` rounds a number to the nearest multiple of one hundred; and so on. For example, `ROUND(14932,-2)` returns 14900 and `ROUND(14932,-3)` returns 15000.

For example, if the default number of decimal places is 2,

- `ROUND(2.50)` returns 3.00
- `ROUND(-2.50)` returns -2.00
- `ROUND(2.00)` returns 2.00

See the table in the description of `INT()` that compares `INT()`, `FLOOR()`, `CEILING()`, and `ROUND()`.

See also `ABS()`, `CEILING()`, `FLOOR()`, `INT()`

RTOD()

Returns the degree value of an angle measured in radians.

Syntax `RTOD(<expN>)`

<expN> A negative or positive number that is the size of the angle in radians.

Description `RTOD()` converts the measurement of an angle from radians to degrees.

To convert radians to degrees, *Visual dBASE*

- Multiplies the number of radians by 180
- Divides the result by pi
- Returns the quotient

An angle of pi radians is equivalent to 180 degrees.

Use `RTOD()` with the trigonometric functions `ACOS()`, `ASIN()`, `ATAN()`, and `ATN2()` to convert the radian return values of these functions to degrees. For example, if the default number of decimal places is 2, `ATAN(1)` returns the value of the angle in radians, 0.79, while `RTOD(ATAN(1))` returns the value of the angle in degrees, 45.00.

Use `SET DECIMALS` to set the number of decimal places `RTOD()` displays.

See also `ACOS()`, `ASIN()`, `ATAN()`, `ATN2()`, `COS()`, `DTOR()`, `PI()`, `SET DECIMALS`, `SIN()`, `TAN()`

SET CURRENCY

SET CURRENCY determines the character(s) used as the currency symbol, and the position of that symbol when displaying monetary values

Syntax SET CURRENCY left | right

SET CURRENCY TO [<expC>]

LEFT Places currency symbol(s) to the left of currency numbers.

RIGHT Places currency symbol(s) to the right of currency numbers.

<expC> The characters that appear as a currency symbol. Although *Visual* dBASE imposes no limit to the length of <expC>, it recognizes only the first nine characters. You can't include numbers in <expC>.

Description Currency symbols are displayed for numbers when you use the "\$" template symbol in a formatting template or the TRANSFORM() function. The defaults for SET CURRENCY are set by the Regional settings of the Windows Control Panel.

Use SET CURRENCY left | right to specify the position of currency symbol(s) in monetary numeric values. Use SET CURRENCY TO to establish a currency symbol other than the default.

When SET CURRENCY is LEFT, dBASE displays only as many currency symbols as fit, together with the digits to the left of any decimal point, within ten character spaces.

SET CURRENCY TO without the <expC> option resets the currency symbol to the default set with the Regional settings of the Windows Control Panel.

See Also SET POINT, SET SEPARATOR, TRANSFORM()

SET DECIMALS

Determines the number of decimal places of numbers to display.

Syntax SET DECIMALS TO [<expN>]

<expN> The number of decimals places, from 0 to 18. The default is 2.

Description Use SET DECIMALS to specify the number of decimal places of numbers you want *Visual* dBASE to display. SET DECIMALS affects the *display* of most mathematical calculations, but not the way numbers are stored on disk or maintained internally.

Excess digits are rounded when a number is displayed. For example, with the default setting of two decimal places, the number 1.995 is displayed as 2.00.

Use SET PRECISION to set the number of decimal places used in comparisons. SET DECIMALS and SET PRECISION are independent settings.

SET DECIMALS TO without *<expN>* resets the number of decimal places back to the default of 2.

See Also INT(), RANDOM(), ROUND(), SET PRECISION, VAL()

SET POINT

Specifies the character that separates decimal digits from integer digits in numeric display.

Syntax SET POINT TO [*<expC>*]

<expC> The character representing the decimal point. You can specify more than one character, but *Visual* dBASE uses only the first one. If you specify a number as a character for *<expC>* (for example, "3"), dBASE returns an error.

The default is set by the Regional Settings of the Windows Control Panel.

Description SET POINT affects both numeric input and display with commands such as EDIT. SET POINT also affects numeric display with commands such as DISPLAY MEMORY, STORE, =, and the PICTURE "." template character. You must use the period in the PICTURE option, regardless of the setting of SET POINT.

SET POINT has no effect on the representation of numbers in *Visual* dBASE expressions and statements. Only a period is valid as a decimal point. For example, if you SET POINT TO "," (comma) and issue the following command:

```
? MAX(123,4, 123,5)
```

Visual dBASE returns an error. The correct syntax is:

```
? MAX(123.4, 123.5)
```

SET POINT TO without the *<expC>* option resets the decimal character to the default set with the Regional settings of the Windows Control Panel.

See Also SET DECIMALS, SET SEPARATOR, STORE

SET PRECISION

Determines the number of digits used when comparing numbers.

Syntax SET PRECISION TO [*<expN>*]

<expN> The number of digits, from 10 to 16. The default is 10.

Description Use SET PRECISION to change the accuracy, or precision, of numeric comparisons. You can set precision from 10 to 16 digits.

SET PRECISION affects data comparisons, but not mathematical computations or data display. Math computations always use full precision internally. To change the number of decimal places dBASE displays, use SET DECIMALS.

In general, you should use as *little* precision as possible for comparisons. Like many programs, *Visual* dBASE handles numbers as base-2 floating point numbers. This format precisely represents fractional values such as 0.5 (1/2) or 0.375 (3/8), but only approximates other values such as 0.4 and 1/9. In addition, precision is also used to represent the integer portion of a number; the larger the integer portion, the less precision is available for the fractional portion. Therefore, comparing values with too much precision results in erroneous mismatches.

Example The following examples demonstrate how numbers are represented and how the precision setting affects data comparisons:

```
set decimals to 18      // to see as many digits as possible
set precision to 16     // maximum
? 0.5                  //      0.500000000000000000 exact
? 0.375                //      0.375000000000000000 exact
? 0.4                  //      0.400000000000000022 16 digits precision
? 1/9                  //      0.111111111111111105 16 digits precision
? 12345.4              //      12345.399999999999640000 11 digits precision
? 123456789.4          //      123456789.400000006000000000 7 digits precision
? 12345.4 - 12345      //      0.399999999999636202 11 digits precision
? 12345.4 - 12345 == 0.4 // False, too much precision attempted
set precision to 10
? 12345.4 - 12345 == 0.4 // True
set decimals to 0      // Has no effect on comparisons
? 12345.4 - 12345 == 0.4 // Still True
set precision to 16
? 12345.4 - 12345 == 0.4 // Still False
set precision to 11
? 12345.4 - 12345 == 0.4 // True
set precision to 12
? 12345.4 - 12345 == 0.4 // True
```

Note that the final comparison to 12 digits returns *true* because the first 12 digits just happen to be the same for both the calculated and literal value of 0.4. In fact, there are only 11 digits of precision in the calculated value. The 12th digit is the first rounding digit.

See Also SET DECIMALS

SET SEPARATOR

Specifies the character that separates each group of three digits (whole numbers) to the left of the decimal point in the display of numbers greater than or equal to 1000.

Syntax SET SEPARATOR TO [<expC>]

<expC> The *whole-number separator*, which is the character that separates each group of three digits to the left of the decimal point in the display of numbers greater than or equal to 1000. You can specify more than one character, but *Visual* dBASE uses only the first one. If you specify a number as a character for <expC> (for example, "3"), dBASE returns an error.

The default is set by the Regional Settings of the Windows Control Panel.

Description SET SEPARATOR affects only the PICTURE "," template character and the numeric display of byte totals for the commands such as DIR, DISPLAY FILES, and LIST FILES. For example, if you SET SEPARATOR TO "." (period) and issue the following, *Visual* dBASE returns 123456 displayed as 123.456:

```
? 123456 PICTURE "999,999"
```

You must use the comma in the PICTURE function, regardless of the setting of SET SEPARATOR.

SET SEPARATOR TO without the <expC> option resets the separator to the default set with the Regional Settings of the Windows Control Panel.

Setting a whole-number separator with SET SEPARATOR doesn't affect the values of numbers, only their display.

See Also SET POINT

SIGN()

Returns an integer that indicates if a specified number is positive, negative, or zero (0).

Syntax SIGN(<expN>)

<expN> The number whose sign (positive, negative, or zero) to determine.

Description Use SIGN() to reduce an arbitrary numeric value into one three numbers: 1, -1, or zero. SIGN() returns 1 if a specified number is positive, -1 if that number is negative, and 0 if that number is 0.

SIGN() is used when the numbers 1, -1, and/or 0 are appropriate for an action, based on the sign—but not the magnitude—of another number. When interested in the sign alone, it's more straightforward to compare the number with zero using a comparison operator.

SIGN() always returns an integer, regardless of the value of SET DECIMALS.

Example The following example is a custom *next()* method for a detail rowset that automatically navigates in the master rowset:

```
function next( nArg )
    if not rowset::next( nArg )                // Navigate as far as specified, but
```



```

// if end of detail rowset
this.masterRowset.next( sign( nArg ) ) // Move forward or backward in master
if nArg < 0 // If navigating backwards
    this.last() // Go to last matching detail row
endif
endif
endif

```

No matter how many records are skipped in the detail rowset, the master rowset is navigated forward one or backward one row only, by using the SIGN() function to convert the row count to 1 or -1 (or zero). Without the SIGN() function, you would have to use a more cumbersome IIF() function or IF block.

When checking to see if the navigation was backwards, it would be redundant to use the SIGN() function again, since you would have to compare the result to zero or -1 anyway. Simply using the less than logical operator is all that is needed.

See Also ABS(), MAX(), MIN(), SET DECIMALS

SIN()

Returns the trigonometric sine of an angle.

Syntax SIN(<expN>)

<expN> The size of the angle in radians. To convert an angle's degree value to radians, use DTOR(). For example, to find the sine of a 30-degree angle, use SIN(DTOR(30)).

Description SIN() calculates the ratio between the side opposite an angle and the hypotenuse in a right triangle. SIN() returns a number from -1 to +1. SIN() returns zero when <expN> is zero, pi, or 2pi radians.

Use SET DECIMALS to set the number of decimal places SIN() displays.

The cosecant of an angle is the reciprocal of the sine of the angle. To return the cosecant of an angle, use 1/SIN().

See also ASIN(), COS(), DTOR(), PI(), RTOD(), SET DECIMALS, TAN()

SQRT()

Returns the square root of a number.

Syntax SQRT(<expN>)

<expN> A positive number whose square root you want to return. If <expN> is a negative number, Visual dBASE generates an error.

Description SQRT() returns the positive square root of a non-negative number. For example SQRT(36) returns 6 because $6^2 = 36$. The square root of 0 is 0.

An alternate way to find the square root is to raise the value to the power of 0.5. For example, the following two statements display the same value:

```
? sqrt(36) )    // displays 6.00
? 36^.5         // displays 6.00
```

Use SET DECIMALS to set the number of decimal places SQRT() displays.

See also EXP(), LOG(), LOG10(), SET DECIMALS

TAN()

Returns the trigonometric tangent of an angle.

Syntax TAN(<expN>)

<expN> The size of the angle in radians. To convert an angle's degree value to radians, use DTOR(). For example, to find the tangent of a 30-degree angle, use TAN(DTOR(30)).

Description TAN() calculates the ratio between the side opposite an angle and the side adjacent to the angle in a right triangle. TAN() returns a number that increases from zero to plus or minus infinity. TAN() returns zero when <expN> is 0, pi, or 2π radians. TAN() is undefined (returns infinity) when <expN> is $\pi/2$ or $3\pi/2$ radians.

Use SET DECIMALS to set the number of decimal places TAN() displays.

The cotangent of an angle is the reciprocal of the tangent of the angle. To return the cotangent of an angle, use $1/\text{TAN}()$.

See also ATAN(), ATN2(), COS(), DTOR(), PI(), RTOD(), SET DECIMALS, SIN()

Date and time objects

Visual dBASE supports two types of dates:

- A *primitive date* that is compatible with earlier versions of dBASE
- A JavaScript-compatible Date object.

A Date object represents a moment in time. It is stored as the number of milliseconds since January 1, 1970 00:00:00 GMT (Greenwich Mean Time). Although GMT and UTC (a compromise between the English and French acronyms for Universal Coordinated Time) are derived differently, they are considered to represent the same time.

Modern operating systems have their own current time zone setting, which is used when handling Date objects. For example, two computers with different time zone settings—whether or not they are physically in different time zones—will display the same time differently.

Primitive dates represent the date only, not the time. (They are considered to be the first millisecond—midnight—of that date.) Literal dates are delimited by curly braces and are evaluated according to the rules used by the CTOD() function. An invalid literal date is always converted to the next valid one; for example, if the current date format is month/day/year, {02/29/1997} is considered March 1, 1997. An empty date is valid and is represented by empty braces: {}.

Visual dBASE will convert one type of date to the other on-the-fly as needed. For example, you may use a Date class method on a primitive date variable or a literal date:

```
? date().toGMTString()  
? {8/21/97}.toString()
```

This creates a temporary Date object from which the method or property is called. Because the object is a temporary copy, calling the *set* methods or assigning to the properties is allowed, but has no apparent effect. You may also

use a date function on a Date object, in which case the time portion of the Date object will be truncated.

Note While the JavaScript-compatible methods are zero-based, *Visual* dBASE functions are one-based. For example, the *getMonth()* method returns 0 for January, while *MONTH()* returns 1.

Visual dBASE also features a Timer object that can cause actions to occur at timed intervals.

class Date

An object that represents a moment in time.

Syntax [*<oRef>* =] new Date()

or

[*<oRef>* =] new Date(*<date expC>*)

or

[*<oRef>* =] new Date(*<msec expN>*)

or

[*<oRef>* =] new Date(*<year expN>*, *<month expN>*, *<day expN>*
[, *<hours expN>* , *<minutes expN>* , *<seconds expN>*])

<oRef> A variable or property in which you want to store a reference to the newly created Date object.

<date expC> A string representing a date and time.

<msec expN> The number of milliseconds since January 1, 1970 00:00:00 GMT. Negative values can be used for dates before 1970.

<year expN> The year.

<month expN> A number representing the month, between 0 and 11: zero for January, one for February, and so on, up to 11 for December.

<day expN> The day of the month, from 1 to 31.

<hours expN> The hours portion of the time, from 0 to 23.

<minutes expN> The minutes portion of the time, from 0 to 59.

<seconds expN> The seconds portion of the time, from 0 to 59.

Properties The following tables list the properties and methods of the Date class. (No events are associated with this class.)

| Property | Default | Description |
|------------------|---------|--|
| <i>className</i> | DATE | Identifies the object as an instance of the Date class (Property discussed in Chapter 5, “Core language.”) |
| <i>date</i> | | The day of the month |
| <i>day</i> | | The day of the week, from 0 to 6: 0 is Sunday, 1 is Monday, and so on |
| <i>hour</i> | | The hour of the time |
| <i>minute</i> | | The minute of the time |
| <i>month</i> | | The month of the year, from 0 to 11: 0 is January, 1 is February, and so on |
| <i>second</i> | | The second of the time |
| <i>year</i> | | The year of the date |

| Method | Parameters | Description |
|----------------------------|---|--|
| <i>getDate()</i> | | Returns day of month |
| <i>getDay()</i> | | Returns day of week |
| <i>getHours()</i> | | Returns hours portion of time |
| <i>getMinutes()</i> | | Returns minutes portion of time |
| <i>getMonth()</i> | | Returns month of year |
| <i>getSeconds()</i> | | Returns seconds portion of time |
| <i>getTime()</i> | | Returns date/time equivalent |
| <i>getTimezoneOffset()</i> | | Returns time zone offset for current locale |
| <i>getYear()</i> | | Returns year of date |
| <i>parse()</i> | <date expC> | Calculates time equivalent for date string |
| <i>setDate()</i> | <expN> | Sets day of month |
| <i>setHours()</i> | <expN> | Sets hours portion of time |
| <i>setMinutes()</i> | <expN> | Sets minutes portion of time |
| <i>setMonth()</i> | <expN> | Sets month of year |
| <i>setSeconds()</i> | <expN> | Sets seconds portion of time |
| <i>setTime()</i> | <expN> | Sets date/time |
| <i>setYear()</i> | <expN> | Sets year of date |
| <i>toGMTString()</i> | | Converts date to string, using Internet (GMT) conventions |
| <i>toLocaleString()</i> | | Converts date to string, using locale conventions |
| <i>toString()</i> | | Converts date to string, using standard JavaScript conventions |
| <i>UTC()</i> | <year expN> , <month expN> , <day expN> [, <hours expN> , <minutes expN> , <seconds expN>] | Calculates time equivalent of date parameters |

Description A Date object represents both a date and time.

There are four ways to create a new Date object:

- When called with no parameters, the Date object contains the current system date and time.
- You can pass a string containing a date and optionally a time.
- You can pass a number representing the number of milliseconds since January 1, 1970, 00:00:00 GMT. Use a negative number for dates before 1970.
- You can pass numeric parameters for each component of the date, and optionally each component of the time.

When you specify a date/time in a string or with the component numbers, the time zone defaults to the current locale.

If you specify a date but don't specify hours, minutes, or seconds, they are set to zero. When passing a string, the *<date expC>* can be in a variety of formats, with or without the time, as shown in the following examples:

```
d1 = new Date( "Jan 5 1996" )           // month, day, year
d2 = new Date( "18 Dec 1994 15:34" )    // day, month, year, and time
d3 = new Date( "1987 Nov 4 9:18:34" )   // year, month, day, and time with seconds
```

You may spell out the month or abbreviate it, down to the first three letters; for example, "April", "Apri", or "Apr". For consistency and because of the three-letter month of May, you should either always spell it out completely or use the first three letters.

Date objects have an inherent value. The format of the date is platform-dependent; in Visual dBASE, the format is same as using the *toLocaleString()* method. Use the *toGMTString()*, *toLocaleString()*, and *toString()* methods to format the Date objects, or create your own. Date objects will automatically type-convert into strings, using the inherent format.

In Visual dBASE, every Date object has a separate property for each date and time component. You may read or write to these properties directly (except for the *day* property, which is read-only), or use the equivalent method. For example, assigning a value to the *minute* property has the same effect as calling the *setMinutes()* method with the value as the parameter. Standard JavaScript does not have these properties, so if you want your code to be portable, avoid direct access to the properties and use the methods.

See also DATE()

class Timer

An object that initiates a recurring action at preset intervals.

Syntax [*<Ref>* =] new Timer()

<oRef> A variable or property in which you want to store a reference to the newly created Timer object.

Properties The following tables list the properties and events of the Timer class. (No methods are associated with this class.)

| Property | Default | Description |
|------------------|---------|---|
| <i>className</i> | Timer | Identifies the object as an instance of the Timer class (Property discussed in Chapter 5, "Core language.") |
| <i>enabled</i> | false | Whether the Timer is active |
| <i>interval</i> | 10 | The interval between actions, in seconds |

| Event | Parameters | Description |
|----------------|------------|---|
| <i>onTimer</i> | | Action to take when <i>interval</i> expires |

Description To use a Timer object:

- 1 Assign an event handler to the *onTimer* event.
- 2 Set the *interval* property to the desired number of seconds.
- 3 Set the *enabled* property to *true* when you want to activate the timer.

The Timer object will start counting down time whenever *Visual* dBASE is idle. When the number of seconds assigned to *interval* has passed, the Timer object's *onTimer* event fires. After the event fires, the Timer object's internal timer is reset back to the *interval*, and the countdown repeats.

To disable the timer, set the *enabled* property to *false*.

A Timer object counts idle time; that is when *Visual* dBASE is not doing anything. This includes waiting for input in the Command window or Navigator. If a process, such as an event handler or program, is running, the counter in all active Timer objects is suspended. When the process is complete and *Visual* dBASE is idle again, the count resumes.

Example Suppose you want to display the date and time in a form. The following is an *onOpen* event handler that creates a Timer object and attaches it to the form. A reference to the form is added to the Timer object so that the timer's *onTimer* event handler can update the form. Another method in the form is assigned as the Timer object's *onTimer* event handler. The time is updated every two seconds instead of every second, so that *Visual* dBASE is not too bogged down constantly updating the time.

```
function Form_onOpen()
    this.timer = new Timer()           // Make timer a property of the form
    this.timer.parent = this           // Assign form as timer's parent
    this.timer.onTimer = this.updateClock // Assign method in form to timer
    this.timer.interval = 2            // Fire timer every 2 seconds
    this.timer.enabled = true          // Activate timer
```

The following is the *updateClock()* method of the form, assigned as the *onTimer* event handler. Because the Timer object calls this method, the *this* reference

refers to the Timer object, not the form, even though the method is a method of the form. A reference to the form has been stored in the *parent* property of the timer; an Text component of the form named *clock* is updated through that reference.

```
function updateClock()
    this.parent.clock.text = new Date()
```

The timer should be deactivated when the form is closed. Use the form's *onClose* event:

```
function Form_onClose()
    this.timer.enabled = false
```

CDOW()

Returns the name of the day of the week of a specified date.

Syntax CDOW(<expD>)

<expD> The date whose corresponding weekday name to return.

Description CDOW() returns a character string containing the name of the day of the week on which a date falls. To return the day of the week as a number from 1 to 7, use DOW().

If you pass an blank or invalid date to CDOW(), it returns "Unknown".

Example The following is a *beforeGetValue* event handler for a date field. It displays recent posting dates as days of the week. Anything older than a week it displays as the date.

```
function postdate_beforeGetValue
    local nDays
    nDays = date() - this.value
    do case
        case this.value == {}           // Blank date
            return "Not posted"
        case nDays < 0                  // Date should never be after current date
            return "Error"
        case nDays == 0                 // Same date as today
            return "Today"
        case nDays < 7                  // Date within the past week
            return cdow( this.value )
        otherwise                       // Older date
            return dtoc( this.value )
    endcase
```

See Also CMONTH(), DATE(), DAY(), DOW(), YEAR()

CMONTH()

Returns the name of the month of a specified date.

Syntax CMONTH(<expD>)

<expD> The date whose corresponding month name to return.

Description CMONTH() returns a character string containing the name of the month in which a date falls. To return the month as a number from 1 to 12, use MONTH().

If you pass an blank or invalid date to CMONTH(), it returns "Unknown".

Example The following funtion uses CMONTH(), DAY(), and YEAR() to return the month, day, and year in a character string—like the MDY() function, but with no leading zero in the day and always with the full year.

```
function mdcy( dArg )
    return cmonth( dArg ) + " " + day( dArg ) + ", " + year( dArg )
```

See Also CDOW(), DAY(), MDY(), MONTH(), YEAR()

CTOD()

Interprets a specified character expression as a literal date.

Syntax CTOD(<expC>)

<expC> The character expression, in the current date format, to return as a date.

Description Use CTOD() to convert a character expression containing a literal date to a date value. Once you convert the string to a date, you can manipulate it with date functions and date arithmetic.

A literal date must be in format:

```
<number><separator><number><separator><number>[BC]
```

where <separator> should be a slash (/), hyphen (-), or period (.). The two <separator> characters should match. You may specify a BC date by including the letters "BC" (not case-sensitive) at the end of the literal date.

To specify a literal date in code, use curly braces ({ }) as literal date delimiters; there is no need to use CTOD(). For example, there two are equivalent:

```
{04/05/06}
ctod( "04/05/06" )
```

The interpretation of the literal date—that is, which numbers are the day, month, and year, and how two-digit years are handled—is controlled by the

DATE()

current settings for SET DATE and SET EPOCH. For example, if SET DATE is MDY and SET EPOCH is 1930, the literal date above is April 5, 2006.

SET DATE also controls the display of dates, while SET EPOCH does not. SET CENTURY controls the display of dates, but has no effect on how dates are interpreted. Two-digit years are always treated as years in the current epoch.

If you pass an invalid date to CTOD(), it attempts to convert the date to a valid one. For example, it interprets June 31 (June only has 30 days) as July 1. If you pass an empty or non-literal-date string to CTOD(), it returns a blank date, which is a valid date value.

Example Suppose a form allows the input of the month and year only. You want to store this as the first day of that month. First create a literal date string from the month and year numbers, then use CTOD() to convert that string into a date, as follows:

```
function saveButton_onClick
  local cDate
  cDate = "" + form.month.value + "/01/" + form.year.value // Create string
  form.rowset.fields[ "Start date" ].value = ctod( cDate ) // Store in date field
  form.rowset.save()
```

This function assumes that the current SET DATE format is MDY, or something similar, like AMERICAN.

See Also DTOC(), DTOS(), SET DATE, SET CENTURY, SET EPOCH

DATE()

Returns the system date.

Syntax DATE()

Description DATE() returns your computer system's current date.

To change the system date, use SET DATE TO.

Example The following statement counts how many records in a table of payments are more than 30 days overdue.

```
count for date() - LAST_PAY > 30 to nOver30
```

See Also SET DATE TO, TIME()

DAY()

Returns the numeric value of the day of the month for a specified date.

Syntax DAY(<expD>)

<expD> The date whose corresponding day-of-the-month number you want to return.

Description DAY() returns a date's day of the month number—a value from 1 to 31.
DAY() returns zero for a blank date.

Example The following is an *onOpen* event handler for a form that makes the “Ship” button invisible on the first day of the month, when inventory is being reconciled:

```
function Form_onOpen()
  if day( date() ) == 1           // Get today's day of month, if first of month
    this.shipButton.visible = false // Prevent shipping
  endif
```

See also DOW(), getDate(), MONTH(), YEAR()

DMY()

Returns a specified date as a character string in DD MONTH YY or DD MONTH YYYY format.

Syntax DMY(<expD>)

<expD> The date to format.

Description DMY() returns a date in DD MONTH YY or DD MONTH YYYY format, where DD is the day number, MONTH is the full month name, and YY is the year number. If SET CENTURY is OFF (the default), DMY() returns the year as 2 digits. If SET CENTURY is ON, DMY() returns the year as 4 digits. If the day is only one digit, it is preceded by a space.

If you pass an blank date to DMY(), it returns "0 Unknown 00" or "0 Unknown 0000".

See Also MDY(), SET CENTURY

DOW()

Returns the day of the week corresponding to a specified date as a number from 1 to 7.

Syntax DOW(<expD>)

<expD> The date whose corresponding weekday number you want to return.

Description DOW() returns the number of the day of the week on which a date falls:

| Day | Number |
|-----------|--------|
| Sunday | 1 |
| Monday | 2 |
| Tuesday | 3 |
| Wednesday | 4 |
| Thursday | 5 |
| Friday | 6 |
| Saturday | 7 |

To return the *name* of the day of the week instead of the number, use CDOW().
DOW() returns zero for a blank date.

Example The following function calculates the date for the Monday that follows the specified date:

```
function nextMonday( dArg )
  if dow( dArg ) == 1           // If it's Sunday
    return dArg + 1             // Monday is the next day
  else                           // Otherwise, subtract DOW()
    return dArg - dow( dArg ) + 9 // to get last week Saturday
  endif                         // then add 9 for next week Monday
```

See also CDOW(), DAY(), MONTH(), YEAR()

D T O C ()

Converts a date into a literal date string.

Syntax D T O C (<expD>)
<expD> The date to return as a string.

Description There are many different ways to represent a date as a string. Use D T O C () to convert a date into a literal date string, one that is suitable for conversion back into a date by C T O D ().

The order of the day, month, and year is controlled by the current SET DATE setting. Whether the year is expressed as two or four digits is controlled by SET CENTURY. The separator character is controlled by SET MARK.

Note To convert a date expression to a character string suitable for indexing or sorting, always use D T O S (), which converts the date into a consistent and sortable format.

If you pass a blank date to DTOC(), it returns a string with spaces instead of digits. For example, if the SET DATE format is AMERICAN and SET CENTURY is OFF, DTOC({ }) returns " / / ".

When concatenating a date to a string, *Visual* dBASE automatically converts the date using DTOC() for you.

Example The following statement writes the current date to the text file opened in the File object fLog:

```
fLog.puts( dtoc( date() ) )
```

The *puts*() method expects a string.

See Also CTOD(), DTOS(), SET CENTURY, SET DATE, SET MARK

DTOS()

Returns a specified date as a character string in YYYYMMDD format.

Syntax DTOS(<expD>)

<expD> The date expression to return as a character string in YYYYMMDD format.

Description Use DTOS() to convert a date expression to a character string suitable for indexing or sorting. For example, you can use DTOS() when indexing on a date field in combination with another field of a different type. DTOS() always returns a character string in YYYYMMDD format, even if SET CENTURY is OFF.

If you pass a blank date to DTOS(), it returns a string with eight spaces, which matches the length of the normal result.

Example The following statement indexes a table of orders by customer ID and order date. The customer ID field is a character field.

```
index on CUST_ID + dtos( ORDER_DATE ) tag CUST_DATE
```

See Also DTOC(), INDEX

ELAPSED()

Returns the number of seconds elapsed between two specified times.

Syntax ELAPSED(<stop time expC>, <start time expC> [, <exp>])

<stop time expC> The time expression, in the format HH:MM:SS, at which to stop timing seconds elapsed. The <stop time expC> argument should be a later time than <start time expC>; if it is not, *Visual* dBASE returns a negative value.

<start time expC> The time expression, in the format HH:MM:SS, at which to start timing seconds elapsed. The <start time expC> argument should be an earlier time than <stop time expC>; if it is not, Visual dBASE returns a negative value.

<exp> Any expression, which causes ELAPSED() to calculate hundredths of a second. The format of both <start time expC> and <stop time expC> can be HH:MM:SS.hh.

Description Use ELAPSED() with TIME() to time a process. Call TIME() at the start of the process and store the resulting time string to a variable. Then call TIME() again at the end of the process. Call ELAPSED() with the start and stop times to calculate the number of seconds between.

ELAPSED() subtracts the value of <start time expC> from <stop time expC>. If <start time expC> is the later time, ELAPSED() returns a negative value. Both <stop time expC> and <start time expC> must be in HH:MM:SS or HH:MM:SS.hh format, where HH is the hour, MM the minutes, SS the seconds, and hh is hundredths of a second.

Without <exp>, any hundredths of a second are truncated and ignored; ELAPSED() does not round hundredths of a second when <exp> is omitted.

Example The following example shows a top-level routine that calls processes records. A subroutine does the processing, and returns the number of records processed. The elapsed time is used to calculate the throughput of a process.

```
local cTimeStart, nRecs, nRecSec, cMsg
cTimeStart = time(1)
nRecs = processRecords()
nRecSec = nRecs / elapsed( time(1), cTimeStart, 1 )
cMsg = ltrim( str( nRecs ) ) + " records processed, " + ;
      ltrim( str( nRecSec ) ) + " records/sec"
msgbox( cMsg, "Process complete" )
```

Note that both the TIME() and ELAPSED() functions use the optional dummy parameter to return and calculate the time to the hundredth of a second.

See Also SECONDS(), TIME()

enabled

Specifies whether a Timer object is active and counting down time.

Property of Timer

Description Set the *enabled* property to *true* to activate the Timer object. When the number of seconds of idle time specified in the *interval* property has passed, the timer's *onTimer* event fires.

When the *enabled* property is set to *false*, the Timer stops counting time and the internal counter is reset. For example, suppose that

- 1 The *interval* is 10, and *enabled* is set to *true*.
- 2 Then 9 seconds of idle time go by, and
- 3 *enabled* is set to *false*.

If *enabled* is set to *true* again, the *onTimer* will fire after another 10 seconds has gone by, even though there was only 1 second left before the timer was disabled.

If a Timer is intended to go off only once instead of repeatedly, set the *enabled* property to *false* in the *onTimer* event handler.

Example Running the following statements in the Command window will cause a message to be displayed once, 5 seconds after timer the is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

See also *interval*, *onTimer*

enabled is also a property of many form components (page 15-94)

getDate()

Returns the numeric value of the day of the month.

Syntax <oRef>.getDate()

<oRef> The Date object whose corresponding day-of-the-month number you want to return.

Property of Date

Description *getDate()* returns a date's day of the month number—a value from 1 to 31.

If the Date object contains a blank date, *getDate()* returns 0.

Example The following is an *onOpen* event handler for a form that makes the “Ship” button invisible on the first day of the month, when inventory is being reconciled:

```
function Form_onOpen()
  if day( date() ) == 1           // Get today's day of month, if first of month
    this.shipButton.visible = false // Prevent shipping
  endif
```

See also *getDay()*, *getMonth()*, *getYear()*, *setDate()*

getDay()

Returns the day of the week corresponding to a specified date as a number from 0 to 6.

Syntax <oRef>.getDay()

<oRef> The Date object whose corresponding weekday number you want to return.

Property of Date

Description *getDay()* returns the number of the day of the week on which a date falls. The number is zero-based:

| Day | Number |
|-----------|--------|
| Sunday | 0 |
| Monday | 1 |
| Tuesday | 2 |
| Wednesday | 3 |
| Thursday | 4 |
| Friday | 5 |
| Saturday | 6 |

Note The equivalent date function *DAY()* is one-based, not zero-based.

The day of the week is the only date/time component you cannot set directly; there is no corresponding *set-* method. It is always based on the date itself.

Example The following is an *onOpen* event handler for a form that makes the “Game center” button visible on the weekends:

```
function Form_onOpen()
    if new Date().getDay() % 6 == 0           // If today is a weekend day
        this.gameCenterButton.visible = true // Enable access to game center page
    endif
```

The day number modulo 6 is zero for both day numbers 0 and 6, the days on the weekend.

See also *DAY()*, *getDate()*, *getMonth()*, *getYear()*

getHours()

Returns the hours portion of a date object.

Syntax <oRef>.getHours()

<oRef> The date object whose hours you want to return.

Property of Date

Description *getHours()* returns the hours portion of the time (using a 24-hour clock) in a Date object: an integer from 0 to 23.

Example The following function returns *true* if the date/time passed to it is during the graveyard shift, between 10 p.m. and 6 a.m.:

```
function isGraveyard( dArg )
  return ( dArg.getHours() >= 22 or dArg.getHours() < 6 )
```

See also *getMinutes()*, *getSeconds()*, *getYear()*, *setHours()*

getMinutes()

Returns the minutes portion of a date object.

Syntax *<oRef>.getMinutes()*

<oRef> The date object whose minutes you want to return.

Property of Date

Description *getMinutes()* returns the minutes portion of the time in a Date object: an integer from 0 to 59.

See also *getHours()*, *getSeconds()*, *getYear()*, *setMinutes()*

getMonth()

Returns the number of the month for a specified date.

Syntax *<oRef>.getMonth()*

<oRef> The Date object whose corresponding month number you want to return.

Property of Date

Description *getMonth()* returns a date's month number. The number is zero-based:

| Month | Number |
|----------|--------|
| January | 0 |
| February | 1 |

`getSeconds ()`

| Month | Number |
|-----------|--------|
| March | 2 |
| April | 3 |
| May | 4 |
| June | 5 |
| July | 6 |
| August | 7 |
| September | 8 |
| October | 9 |
| November | 10 |
| December | 11 |

Note The equivalent date function `MONTH()` is one-based, not zero-based.

See also `getDate()`, `getDay()`, `getYear()`, `MONTH()`, `setMonth()`

getSeconds()

Returns the seconds portion of a date object.

Syntax `<oRef>.getSeconds()`

<oRef> The date object whose seconds you want to return.

Property of Date

Description `getSeconds()` returns the seconds portion of the time in a Date object: an integer from 0 to 59.

See also `getHours()`, `getMinutes()`, `setSeconds()`

getTime()

Returns time equivalent of date/time, in milliseconds.

Syntax `<oRef>.getTime()`

<oRef> The Date object whose time equivalent you want to return.

Property of Date

Description `getTime()` returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the date/time stored in the Date object. All date/times are represented internally by this millisecond number.

Example The following is a stopwatch function that returns the number of seconds since the last time it was called.

```
function stopwatch()
  local thisTime, nSecs
  thisTime = new Date().getTime()
  static lastTime = thisTime
  nSecs = ( thisTime - lastTime ) / 1000
  lastTime := thisTime
  return nSecs
```

The function uses a Date object's *getTime()* method, which keeps time in milliseconds. Whenever the function is called, the variable *thisTime* is set to the current time in milliseconds. The first time through the function, the *lastTime* variable is set to that same time. The difference is calculated, and then the value of *thisTime* is saved in the static variable *lastTime* for the next function call.

To reset the timer, call the function; you may ignore the return value. Then the next time you call the function, you will get the elapsed time. If you're measuring a series of intervals, call the function once between intervals. For example:

```
stopwatch()           // Reset timer
// Process 1
time1 = stopwatch()  // Time for first process
// Process 2
time2 = stopwatch()  // Time for second process
// etc.
```

See also *parse()*, *setTime()*, *UTC()*

getTimezoneOffset()

Returns the time zone offset for a date object in the current locale, in minutes.

Syntax *<oRef>.getTimezoneOffset()*

<oRef> A date object created in the locale in question.

Property of Date

Description All time zones have an offset from GMT (Greenwich Mean Time), from twelve hours behind to twelve hours ahead. *getTimezoneOffset()* returns this offset, in minutes, for the locale in which the Date object was created, taking Daylight Savings Time into account.

For example, the United States and Canada Pacific time zone is eight hours behind GMT. A date in January, when Daylight Savings Time is not in effect, created in the Pacific time zone would have a time zone offset of -480. A date in July, when Daylight Savings Time is in effect, would have a time zone offset of -420, or seven hours, since Daylight Savings Time moves clocks one hour forward, closer to GMT.

In Windows, the locale is determined by the Time Zone setting in each system's Date/Time properties, which is found in the Control Panel, or by double-clicking the clock in the Taskbar.

All Date objects default to the time zone setting of the current locale.

See also `toGMTString()`, `UTC()`

getYear()

Returns the year of a specified date.

Syntax `<obj>.getYear()`

<obj> *<expD>* The Date object whose corresponding year number you want to return.

Property of Date

Description `getYear()` returns a date's year number. For dates in the 1900s, the 2-digit year is returned. The SET CENTURY setting has no effect on `getYear()`.

See also `getDate()`, `getDay()`, `getMonth()`, `YEAR()`

interval

The amount of idle time, in seconds, between the firings of the timer.

Property of Timer

Description Set the *enabled* property to *true* to activate the Timer object. When the number of seconds of idle time specified in the *interval* property has passed, the timer's *onTimer* event fires.

When the *enabled* property is set to *false*, the Timer stops counting time and the internal counter is reset. For example, suppose that

- 1 The *interval* is 10, and *enabled* is set to *true*.
- 2 Then 9 seconds of idle time go by, and
- 3 *enabled* is set to *false*.

If *enabled* is set to *true* again, the *onTimer* will fire after another 10 seconds has gone by, even though there was only 1 second left before the timer was disabled.

interval must be zero or greater. The *interval* may be a fraction of a second; the resolution of the timer is one system clock tick, approximately 0.055 seconds. When *interval* is zero, the timer fires once per clock tick.

Setting the *interval* always resets the internal counter to the newly specified time.

Example Running the following statements in the Command window will cause a message to be displayed once, 5 seconds after timer is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

See also *enabled, onTimer*

MDY()

Returns a specified date as a character string in MONTH DD, YY format.

Syntax MDY(<expD>)

<expD> The date to return as a character string in MONTH DD, YY format.

Description MDY() returns a date in MONTH DD, YY or MONTH DD, YYYY format, where MONTH is the full month name, DD is the day number, and YY is the year number. If SET CENTURY is OFF (the default), MDY() returns the year as 2 digits. If SET CENTURY is ON, MDY() returns the year as 4 digits. MDY() always returns the day portion as 2 digits, with a leading zero for the first nine days of the month.

If you pass an invalid date to MDY(), it returns "Unknown 00, 00" or "Unknown 00, 0000".

See Also DMY(), SET CENTURY

MONTH()

Returns the number of the month for a specified date.

Syntax MONTH(<expD>)

<expD> The date whose corresponding month number you want to return.

Description MONTH() returns a date's month number:

| Month | Number |
|----------|--------|
| January | 1 |
| February | 2 |
| March | 3 |

| Month | Number |
|-----------|--------|
| April | 4 |
| May | 5 |
| June | 6 |
| July | 7 |
| August | 8 |
| September | 9 |
| October | 10 |
| November | 11 |
| December | 12 |

To return the *name* of the month instead of the number, use CMONTH().

MONTH() returns zero for a blank date.

Example The following function returns the date of the last day of the year of the specified date, using date math only. This makes the calculation independent of the current SET DATE setting. The function relies on another function that returns the last day of the month of a specified date.

```
function LDoY( dArg )
  local dDec
  dDec = dArg - day( dArg ) + 28 * ( 13 - month( dArg ) )
  return LDoM( dDec )

function LDoM( dArg )
  local dNxtMonth
  dNxtMonth = dArg - day( dArg ) + 45
  return dNxtMonth - day( dNxtMonth )
```

See also DAY(), DOW(), *getMonth*(), YEAR()

onTimer

When the timer's interval has elapsed.

Parameters none

Property of Timer

Description A Timer object's *onTimer* event is fired every time the amount of idle time specified by the timer's *interval* property has elapsed.

Like all event handlers, inside the *onTimer* event handler, the reference *this* refers to the Timer object itself. To refer to other objects, add references to those objects as properties to the Timer object before activating the timer.

While processing the *onTimer* event, all active timers are suspended, since *Visual* dBASE is busy processing code. Once the *onTimer* event handler has

completed, its internal counter is reset to the *interval*, and all active timers resume counting.

If a Timer is intended to go off only once instead of repeatedly, set the *enabled* property to *false* in the *onTimer* event handler.

Example Running the following statements in the Command window will cause a message to be displayed once, 5 seconds after timer is enabled:

```
t = new Timer()
t.onTimer = {; ? "Ding!"; this.enabled = false}
t.interval = 5
t.enabled = true
```

See also *enabled*, *interval*

parse()

Returns time equivalent of a date/time string, in milliseconds.

Syntax `Date.parse(<date expC>)`

<date expC> The date/time string you want to convert.

Property of Date

Description `parse()` returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the specified date/time string, defaulting to the operating system's current time zone setting. For example, if the time zone is currently set to United States Eastern Standard Time, which is five hours behind GMT, then `Date.parse("Sep 14 1995 11:20")` yields a time which is equivalent to 16:20 GMT.

The string may be in any of the forms acceptable to the Date class constructor, as described under class Date at the beginning of this chapter. In contrast, the `UTC()` method uses numeric parameters for each of the date and time components and assumes GMT as the time zone.

Because `parse()` is a static class method, you call it via the Date class, not a Date object.

Example The following code fragment resets an existing date object *d1* to a date typed into a text control:

```
d1.setTime( Date.parse( this.form.dateText.value ) )
```

See also `getTime()`, `setTime()`, `UTC()`

SECONDS()

Returns the number of seconds that have elapsed on your computer's system clock since midnight.

Syntax SECONDS()

Description SECONDS() returns the number of seconds to the hundredth of a second that have elapsed on your system clock since midnight (12 am). There are 86,400 seconds in a day, so the maximum value SECONDS() can return is 86,399.99, just before midnight.

Use SECONDS() to calculate the amount of time that portions of your program take to run. SECONDS() is more convenient for this purpose than TIME() because SECONDS() returns a number rather than a character string.

You can also use SECONDS() instead of ELAPSED() to determine elapsed time for the current day to within hundredths of a second.

See also ELAPSED(), *getTime()*, TIME()

SET CENTURY

Controls the format in which *Visual* dBASE displays the year portion of dates.

Syntax SET CENTURY on | off

Description When SET CENTURY is ON, *Visual* dBASE displays dates in the current format with 4-digit years; when SET CENTURY is OFF, *Visual* dBASE displays dates in the current format with 2-digit years.

You can enter a date with a 2-, 3-, or 4-digit year whether SET CENTURY is ON or OFF. *Visual* dBASE assumes that 2-digit years are in the epoch designated by SET EPOCH, by default the 1900s. If SET CENTURY is OFF, *Visual* dBASE truncates any digits to the left of the last two when displaying the date. However, *Visual* dBASE stores the correct value of the date internally.

The following table shows the how *Visual* dBASE displays and stores dates depending on the setting of SET CENTURY. (The table assumes SET DATE is AMERICAN and SET EPOCH is 1900.)

| You enter date as | <i>Visual</i> dBASE stores date as | With SET CENTURY ON, <i>Visual</i> dBASE displays | With SET CENTURY OFF, <i>Visual</i> dBASE displays |
|-------------------|------------------------------------|---|--|
| {10/13/94} | 10/13/1994 | 10/13/1994 | 10/13/94 |
| {10/13/994} | 10/13/0994 | 10/13/0994 | 10/13/94 |
| {10/13/1994} | 10/13/1994 | 10/13/1994 | 10/13/94 |
| {10/13/2094} | 10/13/2094 | 10/13/2094 | 10/13/94 |

As the table shows, SET CENTURY doesn't affect the relationship between how you enter a date and how *Visual* dBASE evaluates and stores it. SET CENTURY affects only how *Visual* dBASE *displays* the year portion of the date.

See Also SET DATE, SET EPOCH

SET DATE

Specifies the format *Visual* dBASE uses for the display and entry of dates.

Syntax SET DATE [TO]
AMERICAN | ANSI | BRITISH | FRENCH | GERMAN | ITALIAN | JAPAN | USA | MDY | DMY | YMD

TO Include for readability only; TO has no affect on the operation of the command.

AMERICAN | ANSI | BRITISH | FRENCH | GERMAN | ITALIAN | JAPAN | USA | MDY | DMY | YMD The options correspond to the following formats:

| Option | Format |
|----------|----------|
| AMERICAN | MM/DD/YY |
| ANSI | YY.MM.DD |
| BRITISH | DD/MM/YY |
| FRENCH | DD/MM/YY |
| GERMAN | DD.MM.YY |
| ITALIAN | DD-MM-YY |
| JAPAN | YY/MM/DD |
| USA | MM-DD-YY |
| MDY | MM/DD/YY |
| DMY | DD/MM/YY |
| YMD | YY/MM/DD |

Description SET DATE determines how *Visual* dBASE displays dates; and how literal date strings, like those in curly braces ({ }), are interpreted. If SET CENTURY is ON, *Visual* dBASE displays all formats with a 4-digit year.

The default for SET DATE is set by the Regional Settings in the Windows Control Panel. To change the default, set the DATE parameter in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the DATE parameter directly in VDB.INI.

SET DATE overrides any prior SET MARK setting. However, you can use SET MARK after SET DATE to change the date separator character.

See Also CTOD(), SET CENTURY, SET EPOCH, SET MARK

SET DATE TO

Sets the system date.

Syntax SET DATE TO <expC>

<expC> The character expression, in the current date format, to set as the current system date.

Description Use SET DATE TO to reset the date on your system clock. The date string in <expC> must match the current setting of SET DATE.

The date must be in the range from January 1, 1980, to December 31, 2099.

See Also DATE(), SET DATE, SET TIME

SET EPOCH

Sets the base year for interpreting two-digit years in dates.

Syntax SET EPOCH TO <expN>

Default The default base year is 1900, yielding years from 1900 to 1999.

Description Use SET EPOCH to change how two-digit years are interpreted. This allows you to keep SET CENTURY OFF, while enabling entry of dates that cross a century boundary. The following table shows how dates are interpreted using three different SET EPOCH settings:

| Date | 1900 | 1930 | 2000 |
|----------|------------|------------|------------|
| {5/5/00} | 05/05/1900 | 05/05/2000 | 05/05/2000 |
| {5/5/30} | 05/05/1930 | 05/05/1930 | 05/05/2030 |
| {5/5/99} | 05/05/1999 | 05/05/1999 | 05/05/2099 |

For example, if you SET EPOCH TO 1930, you can continue to use most applications with two-digit years unchanged well into the 21st century, (although you would no longer be able to enter dates before 1930, which would not be a problem with many applications). If your applications use dates that span more than one hundred years, then SET EPOCH alone will not help; you must SET CENTURY ON.

The base year setting takes effect whenever dates are interpreted. In programs, two-digit years in literal dates are evaluated at compile-time. If you use SET EPOCH, be sure it is set correctly when you compile code or run new or changed programs.

SET EPOCH is session-based. You may get the value of SET EPOCH with the SET() and SETTO() functions.

See Also SET CENTURY, SET DATE

SET MARK

Determines the character *Visual* dBASE uses to separate the month, day, and year when it displays dates.

Syntax SET MARK TO [*<expC>*]

<expC> The single *date separator character*. You can specify more than one character for *<expC>*, but *Visual* dBASE uses only the first one.

Description Use SET MARK to change the date separator from the default character. For example, if you issue SET DATE AMERICAN, the date separator character is a forward slash (/), and *Visual* dBASE displays dates in MM/DD/YY format. However, if you specify SET MARK TO "." after issuing SET DATE AMERICAN, *Visual* dBASE displays dates in the format MM.DD.YY. If you issue SET DATE AMERICAN again, the format returns to MM/DD/YY.

Issuing SET MARK TO without *<expC>* resets the date separator character to that of the current date format.

SET MARK controls the separator used for display only. You may use any valid separator character when designating a literal date.

See Also SET CENTURY, SET DATE

SET TIME

Sets the system time.

Syntax SET TIME TO *<expC>*

<expC> The time, which you must specify in one of the following formats:

- HH
- HH:MM or HH.MM
- HH:MM:SS or HH.MM.SS

Description Use SET TIME to reset your system's clock.

See Also SET DATE TO, TIME()

setDate()

Sets day of month.

`setHours ()`

Syntax `<oRef>.setDate(<expN>)`

<oRef> The Date object whose day you want to change.

<expN> The day of month number, normally between 1 and 31.

Property of Date

Description `setDate()` sets the day of month for the Date object.

See also `getDate()`, `setMonth()`, `setYear()`

setHours()

Sets hours portion of time.

Syntax `<oRef>.setHours(<expN>)`

<oRef> The Date object whose hours you want to change.

<expN> The hour number, normally between 0 and 23.

Property of Date

Description `setHours()` sets the hours portion of the time for the Date object.

See also `getHours()`, `setMinutes()`, `setSeconds()`

setMinutes()

Sets minutes portion of time.

Syntax `<oRef>.setMinutes(<expN>)`

<oRef> The Date object whose minutes you want to change.

<expN> The minute number, normally between 0 and 59.

Property of Date

Description `setMinutes()` sets the minutes portion of the time for the Date object.

See also `getMinutes()`, `setHours()`, `setSeconds()`

setMonth()

Sets month of year.

Syntax `<oRef>.setMonth(<expN>)`

<oRef> The Date object whose month you want to change.

<expN> The month number, normally between 0 and 11: 0 for January, 1 for February, and so on, up to 11 for December.

Property of Date

Description `setMonth()` sets the month of year for the Date object.

See also `getMonth()`, `setDate()`, `setYear()`

setSeconds()

Sets seconds portion of time.

Syntax `<oRef>.setSeconds(<expN>)`

<oRef> The Date object whose seconds you want to change.

<expN> The number of seconds, normally between 0 and 59.

Property of Date

Description `setSeconds()` sets the seconds portion of the time for the Date object.

See also `getSeconds()`, `setHours()`, `setMinutes()`

setTime()

Sets date/time of Date object.

Syntax `<oRef>.setTime(<expN>)`

<oRef> The Date object whose time you want to set.

<expN> The number of milliseconds since January 1, 1970 00:00:00 GMT for the desired date/time.

Property of Date

Description While you may use standard date/time nomenclature when creating a new Date object, `setTime()` requires a number of milliseconds. Therefore `setTime()` is used primarily to copy the date/time from one Date object to another. If you tried copying dates like this:

```
d1 = new Date( "Aug 24 1996" )
d2 = new Date()
```

```
setYear ( )
```

```
d2 = d1 // Copy date
```

what you're actually doing is copying an object reference for the first `Date` object into another variable. Both variables now point to the same object, so changing the date/time in one would appear to change the date/time in the other.

To actually copy the date/time, use `setTime()` and `getTime()`:

```
d1 = new Date( "Aug 24 1996" )
d2 = new Date()
d2.setTime( d1.getTime() ) // Copy date
```

If you're copying the date/time when you're creating the second `Date` object, you can use the millisecond value in the `Date` class constructor:

```
d1 = new Date( "Aug 24 1996" )
d2 = new Date( d1.getTime() ) // Create copy of date
```

You may also perform *date math* by adding or subtracting milliseconds from the value.

See also `getTime()`

setYear()

Sets year of date.

Syntax `<oRef>.setYear(<expN>)`

<oRef> The `Date` object whose year you want to change.

<expN> The year. For years in the 1900s, you can specify the year as either a 2-digit or 4-digit year.

Property of `Date`

Description `setYear()` sets the year for the `Date` object.

See also `getYear()`, `setDate()`, `setMonth()`

TIME()

Returns the system time as a character string in HH:MM:SS or HH:MM:SS.hh format.

Syntax `TIME([<exp>])`

<exp> Any expression, which causes `TIME()` to return the current time to the hundredth of a second.

Description TIME() returns a character expression that is your computer system's current time. If you do not pass TIME() an expression, it returns the current system time in HH:MM:SS format, where HH is the hour, MM the minutes, and SS the seconds.

If you pass TIME() an expression, it returns the current system time in HH:MM:SS.hh format, where .hh is hundredths of a second. The type and value of the expression you pass to TIME() has no effect other than to make it include hundredths of a second.

To change the system time, use SET TIME.

See also DATE(), ELAPSED(), SET TIME

toGMTString()

Converts the date into a string, using Internet (GMT) conventions.

Syntax <oRef>.toGMTString()
<oRef> The Date object you want to convert.

Property of Date

Description toGMTString() converts the date, which was created using the operating system's time zone setting, to GMT and returns a string in a format like, "Tue, 07 May 1996 02:55:27 GMT".

Example When the following statement is executed in the Command window, the current date and time is displayed in the results pane in GMT format:

```
? new Date().toGMTString()
```

See also toLocaleString(), toString()

toLocaleString()

Converts the date into a string, using locale conventions.

Syntax <oRef>.toLocaleString()
<oRef> The Date object you want to convert.

Property of Date

Description toLocaleString() converts the date to a string, using the standards for the current locale, like "05/06/96 19:55:27".

Visual dBASE uses Windows' Regional settings from the Control Panel.

toString ()

Example When the following statement is executed in the Command window, the current date and time is displayed in the results pane in locale format:

```
? new Date().toLocaleString()
```

See also *toGMTString(), toString()*

toString()

Converts the date into a string, using standard JavaScript conventions.

Syntax *<oRef>.toString()*

<oRef> The Date object you want to convert.

Property of Date

Description *toString()* converts the date to a string, in standard JavaScript format, which includes the complete time zone description, for example,

“Mon May 06 19:55:27 Pacific Daylight Time 1996”

Example When the following statement is executed in the Command window, the current date and time is displayed in the results pane in standard format:

```
? new Date().toString()
```

See also *toGMTString(), toLocaleString()*

UTC()

Returns time equivalent of the specified date/time parameters using GMT, in milliseconds.

Syntax Date.UTC(*<year expN>*, *<month expN>*, *<day expN>*
[, *<hours expN>* [, *<minutes expN>* [, *<seconds expN>*]]])

<year expN> The year.

<month expN> A number representing the month, between 0 and 11: zero for January, one for February, and so on, up to 11 for December.

<day expN> The day of the month, from 1 to 31.

<hours expN> The hours portion of the time, from 0 to 23.

<minutes expN> The minutes portion of the time, from 0 to 59.

<seconds expN> The seconds portion of the time, from 0 to 59.

Property of Date

Description *UTC()* returns the number of milliseconds since January 1, 1970 00:00:00 GMT for the date/time parameters specified, using GMT as the time zone. In contrast, the *parse()* method takes a string as a parameter, and uses the operating system's current time zone setting as the default.

Because *UTC()* is a static class method, you call it via the *Date* class, not a *Date* object.

Example You cannot specify a time zone when creating a *Date* object with separate date and time components, but you can use *UTC()* for GMT:

```
dLocale = new Date( nYear, nMonth, nDay )           // Time zone of locale
dGMT    = new Date( Date.UTC( nYear, nMonth, nDay ) ) // GMT
```

See also *getTime()*, *setTime()*, *parse()*

YEAR()

Returns the year of a specified date.

Syntax YEAR(<expD>)
<expD> The date whose corresponding year number you want to return.

Property of Date

Description YEAR() returns a date's 4-digit year number. The SET CENTURY setting has no effect on YEAR().

YEAR() returns zero for a blank date.

See also DAY(), MONTH(), *getYear()*

Bitwise

The functions in this chapter are used for bit manipulation and base conversion for unsigned 32-bit values. These values are often passed to and returned by Windows API and other DLL functions. Interpreting such values often requires analysis and manipulation of individual bits.

For all parameters designated as 32-bit integers, non-integers will be truncated to integers. For integers larger than 32 bits, only the least significant (right-most) 32 bits are used.

BITAND()

Performs a bitwise AND.

Syntax BITAND(<expN1>, <expN2>)

<expN1>

<expN2> Two 32-bit integers

Description BITAND() compares bits in the numeric value <expN1> with corresponding bits in the numeric value <expN2>. When both bits in the same position are on (set to 1), the corresponding bit in the returned value is on. In any other case, the bit is off (set to 0).

| AND | 0 | 1 |
|-----|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

Use BITAND() to force individual bits to zero. Create a bit mask: a 32-bit integer with zeroes in the bits you want to force to zero and ones in the bits you want to

BITLESHIFT()

leave alone. Use this bit mask as either one of the parameters to BITAND(), and the other parameter as the number that is modified.

Example The following program displays Windows version information extracted from the return value of the Windows API function GetVersion(), which returns a 32-bit integer. The major version number is in the low byte of the low word, and the minor version number is in the high byte of the low word. For example, if the version is 4.10, the major version number is 4 and the minor version number is 10.

As is common practice, macro-functions are created with the #define preprocessor directive to simplify common bit manipulations. There are functions to extract the high word and low word of 32-bit value, and the high byte and low byte of a 16-bit value. The HIBYTE() macro-function has some defensive programming in case the parameter is larger than 16 bits. The functions BITAND() and BITZRSHIFT() are used to extract the values.

```
#define HIWORD(n) (bitzrshift((n),16))
#define LOWORD(n) (bitand((n),0xFFFF))
#define HIBYTE(n) (bitand(bitzrshift((n),8),0xFF))
#define LOBYTE(n) (bitand((n),0xFF))

if type( "GetVersion" ) # "FP"
    extern clong GetVersion() kernel32
endif

local v, vMajor, vMinor, vBuild, isNT
v = GetVersion()
vMajor = LOBYTE( LOWORD( v ) )
vMinor = HIBYTE( LOWORD( v ) )
isNT = not bitset( v, 31 ) // High bit clear if NT
vBuild = iif( isNT, HIWORD( v ), 0 ) // Ignores Win32s

? iif( isNT, "Windows NT", "Windows 9x" ), ;
  "version " + ltrim( str( vMajor ) ) + "." + str( vMinor, 2, 0, "0" )
if isNT
    ?? " build", ltrim( str( vBuild ) )
endif
```

To get the low word of a 32-bit integer, a bit mask is created with ones in all 16 low bits. The hexadecimal value of this number is FFFF, as shown in the LOWORD() macro-function. Similarly, to get the low byte of a 16-bit integer, the bit mask has ones in the low 8 bits: FF. All the other bits are set to zero when the bitwise AND is performed.

The major version number uses both LOBYTE() and LOWORD(). While this is redundant—LOBYTE() alone would work—it's left in to make the code more symmetrical and self-documenting.

See Also BITOR(), BITSET(), BITXOR()

BITLESHIFT()

Shifts a number's bits to the left.

Syntax BITLSHIFT(<int expN>, <shift expN>)

<int expN> A 32-bit integer.

<shift expN> The number of places to shift, from 0 to 32.

Description BITLSHIFT() moves each bit in the numeric value <int expN> to the left the number of times you specify in <shift expN>. Each time the bits are shifted, the least significant bit (bit 0, the bit farthest to the right) is set to 0, and the most significant bit (bit 31, the bit farthest to the left) is lost.

Shifting a number's bits to the left once has the effect of multiplying the number by two, except that if the number gets too large—equal to or greater than 2^{32} (roughly 4 billion)—the high bit is lost.

Example The following macro-function takes three separate values for red, green, and blue and combines them into a single 24-bit value.

```
#define RGB(r,g,b) ;
    (bitlshift(bitand((r),0xff),16)+bitlshift(bitand((g),0xff),8)+bitand((b),0xff))
```

Each value is 8 bits—BITAND() makes sure of that. The red value is shifted 16 bits to the left to make room for the green and blue values. The green value is shifted 8 bits to the left to make room for the blue value. All three numbers are added together to form a single 24-bit number. For example, suppose you pass the following values, shown here in binary to the macro-function:

```
Red    11000011
Green  10101010
Blue   11111111
```

Shifting the red and green results in the following values:

```
Red    11000011 00000000 00000000
Green  00000000 10101010 00000000
Blue   00000000 00000000 11111111
```

The 8-bit values are shifted so their bits do not overlap. Now, adding the values together combines them into a single 24-bit value:

```
RGB    11000011 10101010 11111111
```

See Also BITRSHIFT(), BITZRSHIFT()

BITNOT()

Inverts the bits in a number

Syntax BITNOT(<expN>)

<expN> A 32-bit integer.

Description BITNOT() inverts all 32 bits in <expN>. All zeroes become ones, and all ones become zeroes.

To invert specific bits, use BITXOR().

See also BITXOR()

BITOR()

Performs a bitwise OR.

Syntax BITOR(<expN1>, <expN2>)

<expN1>
<expN2> Two 32-bit integers

Description BITOR() compares bits in the numeric value <expN1> with corresponding bits in the numeric value <expN2>. When either or both bits in the same position are on (set to 1), the corresponding bit in the returned value is on. When neither element is on, the bit is off (set to 0).

| OR | 0 | 1 |
|----|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

Use BITOR() to force individual bits to one. Create a bit mask: a 32-bit integer with ones in the bits you want to force to one and zeroes in the bits you want to leave alone. Use this bit mask as either one of the parameters to BITOR(), and the other parameter as the number that is modified.

See Also BITAND(), BITSET(), BITXOR()

BITRSHIFT()

Shifts a number's bits to the right, maintaining sign.

Syntax BITRSHIFT(<int expN>, <shift expN>)

<int expN> A signed 32-bit integer.
<shift expN> The number of places to shift, from 0 to 32.

Description Unlike the other bitwise functions, BITRSHIFT() treats its 32-bit integer as a signed 32-bit integer. The sign of a 32-bit integer is stored in the most significant bit (bit 31), which is also referred to as the high bit. If the high bit is 1, the number is negative if it is treated as a signed integer. Otherwise, it is simply a very large unsigned integer.

BITRSHIFT() moves each bit in the numeric value <int expN> to the right the number of times you specify in <shift expN>. Each time the bits are shifted, the

previous value of the high bit is restored, and the least significant bit (bit 0, the bit farthest to the right) is lost. This is called a sign-extended shift, because the sign is maintained.

A similar function, BITZRSHIFT(), performs a zero-fill right shift, which always sets the high bit to zero. If *<int expN>* is a positive integer less than 2³¹, BITZRSHIFT() and BITRSHIFT() have the same effect, because the high bit for such an integer is zero.

Use BITRSHIFT() when you're treating the integer as a signed integer. Use BITZRSHIFT() when the integer is unsigned.

Shifting a number's bits to the right once has the effect of dividing the number by two, dropping any fractions.

See Also BITLSHIFT(), BITZRSHIFT()

BITSET()

Checks if a specified bit in a numeric value is on.

Syntax BITSET(*<int expN>*, *<bit expN>*)

<int expN> A 32-bit integer.

<bit expN> The bit number, from 0 (the least significant bit) to 31 (the most significant bit).

Description BITSET() evaluates the number *<int expN>* and returns *true* if the bit in position *<bit expN>* is on (set to 1), or *false* if it is off (set to 0). For example, the binary representation of 3 is

```
00000000 00000000 00000000 00000011
```

bit number 0 is on, bit number 2 is off.

Example The following statement from the example for BITAND()

```
isNT = not bitset( v, 31 )           // High bit clear if NT
```

uses BITSET() to check the high bit of the value returned by the GetVersion() Windows API function. If the bit is not set, the operating system is Windows NT.

See Also BITAND(), BITLSHIFT(), BITOR(), BITRSHIFT(), BITXOR(), BITZRSHIFT()

BITXOR()

Performs a bitwise exclusive OR.

Syntax BITXOR(*<expN1>*, *<expN2>*)

<expN1>
<expN2> Two 32-bit integers

Description BITXOR() compares bits in a numeric value *<expN1>* with corresponding bits in the numeric value *<expN2>*. When one (and only one) of two bits in the same position are on (set to 1), the corresponding bit in the returned value is on. In any other case, the bit is off (set to 0).

| XOR | 0 | 1 |
|-----|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

This operation is known as exclusive OR, since one bit (and only one bit) must be set on for the corresponding bit in the returned value to be set on.

Use BITXOR() to flip individual bits. Create a bit mask: a 32-bit integer with ones in the bits you want to flip and zeroes in the bits you want to leave alone. Use this bit mask as either one of the parameters to BITXOR(), and the other parameter as the number that is modified.

See Also BITAND(), BITNOT(), BITOR(), BITSET(),

BITZRSHIFT()

Shifts a number's bits to the right.

Syntax BITZRSHIFT(*<int expN>*, *<shift expN>*)

<int expN> A 32-bit integer.

<shift expN> The number of places to shift, from 0 to 32.

Description BITZRSHIFT() moves each bit in the numeric value *<int expN>* to the right the number of times you specify in *<shift expN>*. Each time the bits are shifted, the most significant bit (bit 31, the bit farthest to the left) is set to 0, and the least significant bit (bit 0, the bit farthest to the right) is lost.

Shifting a number's bits to the right once has the effect of dividing the number by two, dropping any fractions.

Like most other bitwise functions, BITZRSHIFT() treats *<int expN>* as an unsigned integer. To shift a signed integer, use BITRSHIFT() instead.

Example The following macro-function, defined with the #define preprocessor directive:

```
#define HIWORD(n) (bitzrshift((n),16))
```

extracts the high word (16 bits) of a 32-bit integer. Shifting the bits 16 places to the right with BITZRSHIFT() moves the high word into the low word, filling the now-

vacated high bits with zeros, resulting in a 32-bit integer with the same value as the original high word.

See also BITLSHIFT(), BITRSHIFT()

HTOI()

Returns the numeric value of a specified hexadecimal number.

Syntax HTOI(<expC>)

<expC> The hexadecimal number whose numeric value to return.

Description Use HTOI() to convert a string containing a hexadecimal number to its numeric value (in decimal). For example, you might allow the input of a hexadecimal number. This input would have to go into a string because the hexadecimal digits A through F are considered characters. To use the hexadecimal number, you would have to convert the hexadecimal string into its numeric value.

HTOI() will attempt to convert a hexadecimal number of any magnitude; it is not limited to 32 bits (8 hexadecimal digits).

You may specify literal hexadecimal numbers by preceding them with 0x; HTOI() is not necessary. For example, 0x64 and HTOI("64") result in the same number: 100 decimal.

Example The following example converts a hexadecimal string typed into an Entryfield into the corresponding numeric value and stores it in a custom property called *numValue*.

```
function address_onChange
    this.numValue = htoi( this.value )
```

See Also ITOH()

ITOH()

Returns the hexadecimal equivalent of a specified number, as a character string.

Syntax ITOH(<int expN> [, <chars expN>])

<int expN> The 32-bit integer whose hexadecimal equivalent to return.

<chars expN> The minimum number of characters to include in the returned hexadecimal character string.

Description Use ITOH() to convert a number to a character string representing its hexadecimal equivalent. The hexadecimal number may be used for display and

ITOH()

editing/input purposes. To use the hexadecimal number as a number, it must be converted back into a numeric value with HTOI().

By default, ITOH() uses only as many characters as necessary to represent *<int expN>* in hexadecimal. If *<chars expN>* is greater than the number of characters required, ITOH() pads the returned string with leading 0's to make it *<chars expN>* characters long. If *<chars expN>* is less than the number of characters required, it is ignored. For example, ITOH(21) returns the string "15", while ITOH(21,4) returns "0015".

Because ITOH() treats the integer as a 32-bit integer, negative integers are always converted into 8 hexadecimal digits. For example, ITOH(-1) returns "FFFFFFFF".

See Also HTOI()

Array objects

Visual dBASE supports a wide variety of array types:

- Arrays of contiguously numbered elements, in one or more dimensions. Elements are numbered from one. There are methods specifically for one- and two-dimensional arrays, which mimic a row of fields and a table of rows.
- Associative arrays, in which the elements are addressed by a key string instead of a number.
- Sparse arrays, which use non-contiguous numbers to refer to elements.

All arrays are objects, and use square brackets ([]) as indexing operators.

Array elements may contain any data type, including object references to other arrays. Therefore you can create nested arrays (multi-dimensional arrays of arrays with fixed length in each dimension), ragged arrays (nested arrays with variable lengths), arrays of associative arrays, and so on.

There are two array classes: `Array` and `AssocArray`. Sparse arrays can be created with any other object. In addition to creating properties by name, you can create numeric properties using the indexing operators. For example,

```
o = new Object()
o.title = "Summer"
o[ 2000 ] = "Sydney"
o[ 1996 ] = "Atlanta"
? o[ 1996 + 4 ] // Displays "Sydney"
```

Array functions

Visual dBASE supports a number of array functions, most of which have equivalent methods in the Array class. These functions are:

| Function | Array class method |
|--------------|--|
| ACOPY() | No equivalent |
| ADEL() | <i>delete()</i> |
| ADIR() | <i>dir()</i> |
| ADIREXT() | <i>dirExt()</i> |
| AELEMENT() | <i>element()</i> |
| AFIELDS() | <i>fields()</i> |
| AFILL() | <i>fill()</i> |
| AGROW() | <i>grow()</i> |
| AINS() | <i>insert()</i> |
| ALEN() | For number of elements, check array's <i>size</i> property |
| ARESIZE() | <i>resize()</i> |
| ASCAN() | <i>scan()</i> |
| ASORT() | <i>sort()</i> |
| ASUBSCRIPT() | <i>subscript()</i> |

Like the equivalent methods, these functions operate on one- and two-dimensional arrays only. ACOPY() and ALEN(), which have no direct equivalents, are described separately. The other functions are not described in detail.

The use of those functions, although not recommended, is similar to the equivalent method. For a given method like:

```
aExample.fill( 0 ) // Fill array with zeros
```

the equivalent function uses the reference to the array as its first parameter and all other parameters (if any) following it:

```
afill( aExample, 0 )
```

The parameters following the array name in the function are identical to the parameters to the equivalent method, and the functions return the same values as the methods.

class Array

An array of elements, in one or more dimensions.

Syntax [*<oRef>* =] new Array([*<dim1 expN>* [,*<dim2 expN>*...]])

<oRef> A variable or property in which to store a reference to the newly created Array object.

<dim1 expN> [, <dim2 expN> ...] The size of the array in each specified dimension. If no dimensions are specified, the array is a one-dimensional array with zero elements.

Properties The following tables list the properties and methods of the Array class. (No events are associated with this class.)

| Property | Default | Description |
|-------------------|---------|---|
| <i>className</i> | ARRAY | Identifies the object as an instance of the Array class (Property discussed in Chapter 5, “Core language.”) |
| <i>dimensions</i> | | The number of dimensions in the array |
| <i>size</i> | 0 | The number of elements in the array |

| Method | Parameters | Description |
|------------------|---|---|
| <i>add()</i> | <exp> | Increases the size of a one-dimensional array by one and assigns the passed value to the new element. |
| <i>delete()</i> | <position expN> [1 2] | Deletes an element from a one-dimensional array, or deletes a row (1) or column (2) of elements from a two-dimensional array, without changing the size of the array. |
| <i>dir()</i> | [<filespec expC>] | Stores in the array five characteristics of specified files: name, size, modified date, modified time, and file attribute(s). Returns the number of files whose characteristics are stored. |
| <i>dirExt()</i> | [<filespec expC>] | Same as <i>dir()</i> method, but adds short (8.3) file name, create date, create time, and access date. |
| <i>element()</i> | <row expN> [, <col expN>] | Returns the element number for the element at the specified row and column. |
| <i>fields()</i> | | Stores table structure information for the current table in the array |
| <i>fill()</i> | <exp> , <start expN> [, <count expN>] | Stores a specified value into one or more elements of the array. |
| <i>grow()</i> | 1 2 | When passed 1, adds a single element to a one-dimensional array or a row to a two-dimensional array; when passed 2, adds a column to the array. |
| <i>insert()</i> | <element expN> [1 2] | Inserts an element, row (1), or column (2) into an array without changing the size of the array (the last element, row, or column is lost). |
| <i>resize()</i> | <rows expN> [, <cols expN> [, <retain values>]] | Increases or decreases the size of an array. First passed parameter indicates the new number of rows, the second parameter indicates the new number of columns. If the third parameter is zero, current values are relocated; if nonzero, they are retained in their old positions. |
| <i>scan()</i> | <exp> , <start expN> [, <count expN>] | Searches an array for the specified expression; returns the element number of the first element that matches the expression, or zero if the search is unsuccessful. |

| Method | Parameters | Description |
|--------------------------|--|--|
| <code>sort()</code> | <code><start expN></code> <code>[, <count expN></code> <code>[0 1]]</code> | Sorts the elements in a one-dimensional array or the rows in a two-dimensional array in ascending (0) or descending (1) order. |
| <code>subscript()</code> | <code><element expN></code> <code>1 2</code> | Returns the row (1) or column (2) subscript for the specified element number. |

Description

An Array object is a standard array of elements, addressed by a contiguous range of numbers in one or more dimensions. The array can hold as many elements as memory allows. You can create arrays that contain more than two dimensions, but most *Visual* dBASE Array methods work only on one- or two-dimensional arrays. For a two-dimensional array, the first dimension is considered the row and the second dimension is the column. For example, the following statement creates an array with 3 rows and 4 columns:

```
a = new Array( 3, 4 )
```

There are two ways to refer to individual elements in an array; you can use either element *subscripts* or the element *number*. Element subscripts, one for each dimension, are values that represent the element's position in that dimension. For a two-dimensional array, they indicate the row and column in which an element is located. Element numbers indicate the sequential position of the element in the array, starting with the first element in the array and increasing in each dimension, with the last dimension first. For a two-dimensional array, the first element is in the first column of the first row, the second element is in the second column of the first row, and so on.

To determine the number of dimensions in an array, check its *dimensions* property (it's read-only). The array's *size* property reflects the number of elements in the array. To determine the number of rows or columns in a two-dimensional array, use the `ALen()` function. There is no built-in way to determine the size of dimensions above two.

In an Array object, element numbering starts with one. You cannot create elements outside the defined range of elements or subscripts (although you could change the dimensions of the array if desired). For example, a 3-row, 4-column array has 12 elements, numbered 1 to 12. The first element's subscripts are [1,1] and the last element is [3,4].

Certain *Visual* dBASE methods require the element number, and others require the subscripts. If you are using one- or two-dimensional arrays, you can use `element()` to determine the element number if you know the subscripts, and `subscript()` to determine the subscripts if you know the element number.

Array elements may contain any data type, including object references to other arrays. Therefore you can create nested arrays (multi-dimensional arrays of arrays with fixed length in each dimension), ragged arrays (nested arrays with variable lengths), arrays of associative arrays, and so on.

With both nested and multi-dimensional arrays, you end up with multiple dimensions or levels of elements, but when you nest arrays, you create separate array objects, and the methods that are designed to work on the multiple

dimensions of a single Array object will not work on the separate dimensions of the nested arrays.

In addition to creating an array with the NEW operator, you can create a populated one-dimensional array using the literal array syntax. For example, this statement

```
a1 = {"A", "B", "C"}
```

creates an Array object with three elements: "A", "B", and "C". You can nest literal arrays. For example, if this statement:

```
a2 = { {1, 2, 3}, a1 }
```

followed the first, you would then have a nested array.

To access a value in a nested array, use the index operators in series. Continuing the example, the third element in the first array would be accessed with:

```
x = a2[1][3] // 3
```

One-dimensional arrays are the only Array objects that are allowed to have zero elements. This is particularly useful for building arrays dynamically. To create a zero-element array, create a NEW Array with no parameters:

```
a0 = new Array()
```

Then use the *add()* method to add elements to the array.

Example The following statements create a 3 row, 4 column array with the letters "A" through "L" with two different techniques and use a function to display each array.

```
aAlpha = new Array( 3, 4 )
aAlpha[0,0] = "A"; aAlpha[0,1] = "B"; aAlpha[0,2] = "C"; aAlpha[0,3] = "D"
aAlpha[1,0] = "E"; aAlpha[1,1] = "F"; aAlpha[1,2] = "G"; aAlpha[1,3] = "H"
aAlpha[2,0] = "I"; aAlpha[2,1] = "J"; aAlpha[2,2] = "K"; aAlpha[2,3] = "L"
displayArray( aAlpha )

aAlpha = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L" }
aAlpha.resize( 3, 4 )
displayArray( aAlpha )
```

The second array takes advantage of the literal array syntax, but *resize()* only creates a one- or two-dimensional array.

The *displayArray()* function is used to display the contents of the array in the results pane of the Command window. It is shown in the example for *dimensions* on page 10-13.

See also class AssocArray

class AssocArray

A one-dimensional associative array, in which the elements can be referenced by string.

Syntax [*<ObjRef>* =] new AssocArray()

<ObjRef> A variable or property in which to store a reference to the newly created AssocArray object.

Properties The following tables list the properties and methods of the AssocArray class. (No events are associated with this class.)

| Property | Default | Description |
|------------------|------------|--|
| <i>className</i> | ASSOCARRAY | Identifies the object as an instance of the AssocArray class (Property discussed in Chapter 5, “Core language.”) |
| <i>firstKey</i> | | Character string assigned as the subscript of the first element of an associative array |

| Method | Parameters | Description |
|----------------------|-------------------------|--|
| <i>count</i> () | | Returns the number of elements in the associative array |
| <i>isKey</i> () | <i><key expC></i> | Returns <i>true</i> or <i>false</i> to indicate whether the character string is a key of the associative array |
| <i>nextKey</i> () | <i><key expC></i> | Returns the associative array key following the passed key |
| <i>removeAll</i> () | | Deletes all elements from the associative array |
| <i>removeKey</i> () | <i><key expC></i> | Deletes the specified element from the associative array |

Description In an associative array, elements are associated with arbitrary character strings, which act as key values. The keys may be of any length, and are case-sensitive. An AssocArray is a one-dimensional array.

New elements are created simply by assigning a value to a key. If the key does not exist, a new element is created. If the key already exists, then the old value is replaced. For example,

```
aTest = new AssocArray()
aTest[ "alpha" ] = 1 // Create element with key "alpha" value 1
aTest[ "beta" ] = 2 // Create element with key "beta" value 2
aTest[ "alpha" ] = 3 // Change value of element "alpha" to 3
aTest[ "Beta" ] = 4 // Create element with key "Beta" value 4
```

The *isKey*() method will check if a given string is a key value in the associative array, and *removeKey*() will remove the element for a given key value from the array. *removeAll*() removes all the elements from the array.

The order of elements in an associative array is undefined. They are not necessarily sorted in the order they were added or sorted by their key values. You can think of an associative array as a bag of elements, and depending on what's in the bag, the order is different. But no matter what's in the associative array, you can use its *firstKey* property to get a key value, and use the *nextKey*() method to get all the other key values. The *count*() method will return the number of elements in the array so that you can call *nextKey*() as many times as needed.

Example Suppose you want to create an associative array that associates country codes with the name of the country. You could use a table for the lookup, but because the lookups don't change, reading the table into an array once at the beginning of the application makes the application run faster.

```
use COUNTRY order CODE
aCountry = new AssocArray()
scan
  aCountry[ CODE ] = NAME
endscan
```

If you had to create the array manually, the code would look like this:

```
aCountry = new AssocArray()
aCountry[ "AFG" ] = "Afghanistan"
aCountry[ "ALB" ] = "Albania"
aCountry[ "ALG" ] = "Algeria"
aCountry[ "ASA" ] = "American Samoa"
:
aCountry[ "ZAM" ] = "Zambia"
aCountry[ "ZIM" ] = "Zimbabwe"
```

See also class Array

ACOPY()

Copies elements from one array to another. Returns the number of elements copied.

Syntax ACOPY(<source array>, <target array>
[, <starting element expN> [, <elements expN> [, <target element expN>]]])

<source array> A reference to the array from which to copy elements.

<target array> A reference to the array that elements are copied to.

<starting element expN> The position of the element in <source array> from which ACOPY() starts copying. Without <starting element expN>, ACOPY() copies all the elements in <source array> to <target array>.

<elements expN> The number of elements in <source array> to copy. Without <elements expN>, ACOPY() copies all the elements in <source array> from <starting element expN> to the end of the array. If you want to specify a value for <elements expN>, you must also specify a value for <starting element expN>.

<target element expN> The position in <target array> to which ACOPY() starts copying. Without <target element expN>, ACOPY() copies elements to <target array> starting at the first position. If you want to specify a value for <target element expN>, you must also specify values for <starting element expN> and <elements expN>.

add ()

Description ACOPY() copies elements from one array to another. The dimensions of the two array do not have to match; the elements are handled according to element number.

The target array must be big enough to contain all the elements being copied from the source array; otherwise no elements are copied and an error occurs.

See Also *element()*

add()

Adds an element to a one-dimensional array.

Syntax `<oRef>.add(<exp>)`

<oRef> A reference to the one-dimensional array to which you want to add the element.

<exp> An expression of any type you want to assign to the new element.

Property of Array

Description Use *add()* to dynamically build a one-dimensional array.

add() adds a new element to a one-dimensional array and assigns *<exp>* to the new element.

You can create an empty one-dimensional array in a statement like:

```
a = new Array() // No parameters to Array class creates empty 1-D array
```

and add elements as needed.

Example The following function is an *onOpen* event handler for a ComboBox component. It creates a one-dimensional array from values in a field in a table and assigns that array as the *dataSource* property of the Select component. The table is already opened in the query *sections1*.

```
function sectionCombo_onOpen()  
  this.aSections = new Array()  
  if form.sections1.rowset.first()  
    do  
      this.aSections.add( form.sections1.rowset.fields[ "Name" ].value )  
      until not form.sections1.rowset.next()  
    endif  
    this.dataSource = "array this.aSections"
```

See also *grow()*

ALLEN()

Returns the number of elements, rows, or columns of an array.

Syntax ALLEN(<array> [, <expN>])

<array> A reference to a one- or two-dimensional array.

<expN> The number 0, 1, or 2, indicating which array information to return: elements, rows, or columns. The following table describes what ALLEN() returns for different <expN> values:

| If <expN> is... | ALLEN() returns... |
|------------------------|---|
| <i>not supplied</i> | Number of elements in the array |
| 0 | Number of elements in the array |
| 1 | For a one-dimensional array, the number of elements For a two-dimensional array, the number of rows (the first subscript of the array) |
| 2 | For a one-dimensional array, 0 (zero) For a two-dimensional array, the number of columns (the second subscript of the array) |
| <i>any other value</i> | 0 (zero) |

Property of Array

Description Use ALLEN() to determine the dimensions of an array—either the number of elements it contains, or the number of rows or columns it contains.

The number of elements in an array (with any number of dimensions) is also reflected in the array's *size* property.

If you need to determine both the number of rows and the number of columns a two-dimensional array contains, call ALLEN() twice, once with a value of 1 for <expN> and once with a value of 2 for <expN>. For example, the following determines the number of rows and columns contained in aExample:

```
nRows = alen( aExample, 1 )
nCols = alen( aExample, 2 )
```

Example ALLEN() is used in the *displayArray()* function, shown in the example for *dimensions* on page 10-13, to determine the number of rows and columns in a two-dimensional array.

See Also *size, subscript()*

count()

Returns the number of elements in an associative array.

delete ()

Property of AssocArray

Description Use *count()* to determine the number of elements in an associative array.
Because associative arrays use arbitrary strings as keys and change size dynamically, you need to get the number of elements in an associative array if you want to loop through its elements.

Example The following statements loop through an associative array and display all its elements:

```
aTest = new AssocArray()  
aTest[ "USA" ] = "United States of America"  
aTest[ "RUS" ] = "Russian Federation"  
aTest[ "GER" ] = "Germany"  
aTest[ "CHN" ] = "People's Republic of China"  
  
cKey = aTest.firstKey           // Get first key in AssocArray  
// Get number of elements in AssocArray and loop through them  
for nElements = 1 to aTest.count()  
    ? cKey                     // Display key value  
    ? aTest[ cKey ]            // Display element value  
    cKey := aTest.nextKey( cKey ) // Get next key value  
endfor
```

See also *firstKey*, *nextKey()*

count() is also a method of the Rowset class (page 14-62).

delete()

Deletes an element from a one-dimensional array, or deletes a row or column of elements from a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful. The remaining elements move forward to replace the deleted element(s); the dimensions of the array do not change.

Syntax <oRef>.delete(<position expN> [, <row/column expN>])

<oRef> A reference to the one- or two-dimensional array from which you want to delete data.

<position expN> When the array is a one-dimensional array, <position expN> specifies the number of the element to delete.

When the array is a two-dimensional array, <position expN> specifies the number of the row or column whose elements you want to delete. The second argument (discussed in the next paragraph) specifies whether <position expN> is a row or a column.

<row/column expN> Either 1 or 2. If you omit this argument or specify 1, a row is deleted from a two-dimensional array. If you specify 2, a column is deleted. Visual dBASE generates an error if you use <row/column expN> with a one-dimensional array.

Property of Array

Description Use *delete*() to delete selected elements from an array without changing the size of the array. *delete*() does the following:

- Deletes an element from a one-dimensional array, or deletes a row or column from a two-dimensional array
- Moves all remaining elements toward the beginning of the array (up if a row is deleted, to the left if an element or column is deleted)
- Inserts *false* values in the last position(s)

Adjust the array's *size* property or use *resize*() to make the array smaller after you *delete*() if you want the net effect of removing elements.

One-dimensional arrays When you issue *delete*() for a one-dimensional array, the element in the specified position is deleted, and the remaining elements move one position toward the beginning of the array. The logical value *false* is stored to the element in the last position.

For example, if you define a one-dimensional array with

```
aAlpha = {"A", "B", "C"}
```

the resulting array has one row and can be illustrated as follows:

```
A    B    C
```

Issuing *aAlpha.delete*(2) deletes element number 2 whose value is "B," moves the value in *aAlpha*[3] to *aAlpha*[2], and stores *false* to *aAlpha*[3] so that the array now contains these values:

```
A    C    false
```

Two-dimensional arrays When you issue *delete*() for a two-dimensional array, the elements in the specified row or column are deleted, and the elements in the remaining rows or columns move one position toward the beginning of the array. The logical value *false* is stored to the elements in the last row or column.

For example, suppose you define a two-dimensional array and store letters to the array. The following illustration shows how the array is changed by *aAlpha.delete*(2,2).

`delete ()`

Figure 10.1 Using `delete ()` with a two-dimensional array

aAlpha.delete(2,2)

- 1 Original array created as:
`aAlpha = new Array(3,4)`
`aAlpha[1,1] = "A"`
`aAlpha[1,2] = "B"`
`...`
`aAlpha[3,4] = "L"`

| | | | | | | | |
|---|----------|----|----------|----|----------|----|----------|
| 1 | A 1,1 | 2 | B 1,2 | 3 | C 1,3 | 4 | D 1,4 |
| 5 | E 2,1 | 6 | F 2,2 | 7 | G 2,3 | 8 | H 2,4 |
| 9 | I 3,1 | 10 | J 3,2 | 11 | K 3,3 | 12 | L 3,4 |

Initial contents of the array aAlpha

- 2 `aAlpha.delete(2,2)`
deletes the elements in the
second column...

| | | | | | | |
|---|----------|--|----|----------|----|----------|
| 1 | A 1,1 | | 3 | C 1,3 | 4 | D 1,4 |
| 5 | E 2,1 | | 7 | G 2,3 | 8 | H 2,4 |
| 9 | I 3,1 | | 11 | K 3,3 | 12 | L 3,4 |

- 3 Shifts the elements in the
remaining columns towards the
beginning of the array...

| | | | | | | | |
|---|----------|----|----------|----|----------|----|--|
| 1 | A 1,1 | 2 | C 1,2 | 3 | D 1,3 | 4 | |
| 5 | E 2,1 | 6 | G 2,2 | 7 | H 2,3 | 8 | |
| 9 | I 3,1 | 10 | K 3,2 | 11 | L 3,3 | 12 | |

- 4 And inserts logical *false* values as
elements in the last column,
resulting in this array:

| | | | | | | | |
|---|----------|----|----------|----|----------|----|--------------|
| 1 | A 1,1 | 2 | C 1,2 | 3 | D 1,3 | 4 | false 1,4 |
| 5 | E 2,1 | 6 | G 2,2 | 7 | H 2,3 | 8 | false 2,4 |
| 9 | I 3,1 | 10 | K 3,2 | 11 | L 3,3 | 12 | false 3,4 |

*Contents of the array after issuing
aAlpha.delete(2,2)*

Example The following code removes elements from the array `aTest` that have the letter "e" in them.

```

aTest = {"alpha", "beta", "gamma", "delta"}
nDeleted = 0 // Count deleted elements
// Loop through array backwards
for nElement = aTest.size to 1 step -1
    if "e" $ aTest[ nElement ] // If element contains "e"
        aTest.delete( nElement ) // Delete element
        nDeleted++ // Increment delete count
    endif
endfor

```

```

if nDeleted > 0
    aTest.size := aTest.size - nDeleted    // Discard false elements
endif

// Display elements (looping forward)
for nElement = 1 to aTest.size
    ? aTest[ nElement ]
endfor

```

The loop to delete the elements runs through the array backwards because *delete()* moves all remaining elements forward. You would then have to recheck the same element number and juggle the element counter. It's simpler to just loop through the array backwards.

See also *fill()*, *grow()*, *insert()*, *resize()*, *size*

delete() is also a method of the File (page 11-13), Rowset (page 14-65), and UpdateSet (page 14-65) classes.

dimensions

The number of dimensions in an Array object.

Property of Array

Description *dimensions* indicates the number of dimensions in an Array object. It is a read-only property.

You can use the *resize()* method to change the number of dimensions to one or two, but for more than two you would have to create a new array.

If the array has one or two dimensions, you can use the *ALen()* function to determine the size of each dimension. There is no built-in way to determine dimension sizes for arrays with more than two dimensions.

Example The following function displays the contents of an array, but only if the array has one or two dimensions:

```

function displayArray( aArg, nColWidth )
local nElement, nCols, nRows, nCol, nRow
#define DEFAULT_WIDTH 2
if argcount() < 2
    nColWidth = DEFAULT_WIDTH
endif
do case
case aArg.dimensions == 1
    ? replicate( "-", nColWidth * aArg.size )
    ?
    for nElement = 1 to aArg.size          // Display elements
        ?? aArg[ nElement ] at nColWidth * ( nElement - 1 )
    endfor                                // in a single line
case aArg.dimensions == 2
    nRows = alen( aArg, 1 )                // Determine # of rows

```

`dir ()`

```
nCols = alen( aArg, 2 )           // Determine # of columns
? replicate( "-", nColWidth * nCols )
for nRow = 1 to nRows
    ?                               // Each row on its own line
    for nCol = 1 to nCols           // Display each row as before
        ?? aArg[ nRow, nCol ] at nColWidth * ( nCol - 1 )
    endfor
endfor
otherwise
    msgbox( "Error: only 1 or 2 dimensions allowed", "Alert" )
endcase
```

See also `resize()`, `size`, `subscript()`

`dir()`

Fills the array with five characteristics of specified files: name, size, modified date, modified time, and file attribute(s). Returns the number of files whose characteristics are stored.

Syntax `<oRef>.dir([<filename skeleton expC> [, <DOS file attribute list expC>]])`

<oRef> A reference to the array in which you want to store the file information. `dir()` will automatically redimension or increase the size of the array to accommodate the file information, if necessary.

<filename skeleton expC> The file-name pattern (using wildcards) describing the files whose information you want to store to **<oRef>**.

<file attribute list expC> The letter or letters D, H, S, and/or V representing one or more file file attributes.

If you want to specify a value for **<file attribute expC>**, you must also specify a value or `"*.*"` for **<filename skeleton expC>**.

The meaning of each attribute is as follows:

| Character | Meaning |
|-----------|--------------|
| D | Directories |
| H | Hidden files |
| S | System files |
| V | Volume label |

If you supply more than one letter for **<file attribute expC>**, include all the letters between one set of quotation marks, for example, `aFiles.dir("*.*", "HS")`.

Property of Array

Description Use `dir()` to store information about files to an array, which is dynamically resized so all returned information fits in the array. The resulting array is

always a two-dimensional array, unless there are no files, in which case the array is not modified.

Without *<filename skeleton expC>*, *dir()* stores information about all files in the current directory, unless they are hidden or system files. For example, if you want to return information only on DBF tables, use *"*.DBF"* as *<filename skeleton expC>*.

If you want to include directories, hidden files, or system files in the array, use *<file attribute expC>*. When D, H, or S is included in *<file attribute expC>*, all directories, hidden files, and/or system files (respectively) that match *<filename skeleton expC>* are added to the array.

When V is included in *<file attribute expC>*, *dir()* ignores *<filename skeleton expC>* as well as other characters in the attribute list, and stores the volume label to the first element of the array.

dir() stores the following information for each file in each row of the array. The data type for each is shown in parentheses:

| Column 1 | Column 2 | Column 3 | Column 4 | Column 5 |
|--------------------------|-------------------|-------------------------|------------------------------|----------------------------------|
| File name (character) | Size (numeric) | Modified date (date) | Modified time (character) | File attribute(s) (character) |

The last column (file attribute) can contain one or more of the following file attributes, in the order shown:

| Attribute | Meaning |
|-----------|---|
| R | Read-only file |
| A | Archive file (modified since it was last backed up) |
| S | System file |
| H | Hidden file |
| D | Directory |

If the file has the attribute, the letter code is in the column. Otherwise, there is a period. For example, a file with none of the attributes would have the following string in column 5:

.....

A read-only, hidden file would have the following string in column 5:

R..H.

Use *dirExt()* to get extended Windows 95/NT file information.

Example The following example uses *dir()* to store the file information for all the files in the root directory of the current drive to the array *aFiles*. The file name and attributes string is displayed for all the files in the results pane of the Command window. Manifest constants to represent the columns are created with the *#define* preprocessor directive to make the code more readable.

dirExt ()

```
#define ARRAY_DIR_NAME 1 // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 2
#define ARRAY_DIR_DATE 3
#define ARRAY_DIR_TIME 4
#define ARRAY_DIR_ATTR 5

aFiles = new Array() // Array will be resized as needed
nFiles = aFiles.dir( "\ *.*", "HS" ) // Include Hidden and System files
for nFile = 1 to nFiles
    ? aFiles[ nFile, ARRAY_DIR_NAME ]
    ?? aFiles[ nFile, ARRAY_DIR_ATTR ] at 25
endfor
```

See also `dirExt()`, `sort()`

dirExt()

`dirExt()` is an extended version of the `dir()` method. It fills the array with nine characteristics of specified files: name, size, modified date, modified time, file attribute(s), short (8.3) file name, create date, create time, and access date. Returns the number of files whose characteristics are stored.

Syntax `<oRef>.dirExt([<filename skeleton expC> [, <file attribute list expC>]])`

<oRef> A reference to the array in which you want to store the file information. `dirExt()` will automatically redimension or increase the size of the array to accommodate the file information, if necessary.

<filename skeleton expC> The file-name pattern (using wildcards) describing the files whose information you want to store to `<oRef>`.

<file attribute list expC> The letter or letters D, H, S, and/or V representing one or more file attributes.

If you want to specify a value for `<file attribute expC>`, you must also specify a value or `"*.*"` for `<filename skeleton expC>`.

The meaning of each attribute is as follows:

| Character | Meaning |
|-----------|--------------|
| D | Directories |
| H | Hidden files |
| S | System files |
| V | Volume label |

If you supply more than one letter for `<file attribute expC>`, include all the letters between one set of quotation marks, for example, `aFiles.dirExt("*.*", "HS")`.

Property of Array

Description Use `dirExt()` to store information about files to an array, which is dynamically resized so all returned information fits in the array. The resulting array is always a two-dimensional array, unless there are no files, in which case the array is not modified.

Without `<filename skeleton expC>`, `dirExt()` stores information about all files in the current directory, unless they are hidden or system files. For example, if you want to return information only on DBF tables, use `"*.DBF"` as `<filename skeleton expC>`.

If you want to include directories, hidden files, or system files in the array, use `<file attribute expC>`. When D, H, or S is included in `<file attribute expC>`, all directories, hidden files, and/or system files (respectively) that match `<filename skeleton expC>` are added to the array.

When V is included in `<file attribute expC>`, `dirExt()` ignores `<filename skeleton expC>` as well as other characters in the attribute list, and stores the volume label to the first element of the array.

`dirExt()` stores the following information for each file in each row of the array. The data type for each is shown in parentheses:

| Column 1 | Column 2 | Column 3 | Column 4 | Column 5 |
|--------------------------|-------------------|-------------------------|------------------------------|----------------------------------|
| File name (character) | Size (numeric) | Modified date (date) | Modified time (character) | File attribute(s) (character) |

| Column 6 | Column 7 | Column 8 | Column 9 |
|--------------------------------------|-----------------------|----------------------------|-----------------------|
| Short (8.3) file name (character) | Create date (date) | Create time (character) | Access date (date) |

Column 5 (file attribute) can contain one or more of the following file attributes, in the order shown:

| Attribute | Meaning |
|-----------|---|
| R | Read-only file |
| A | Archive file (modified since it was last backed up) |
| S | System file |
| H | Hidden file |
| D | Directory |

If the file has the attribute, the letter code is in the column. Otherwise, there is a period. For example, a file with none of the attributes would have the following string in column 5:

.....

A read-only, hidden file would have the following string in column 5:

R..H.

Use `dir()` to get basic file information only.

Example The following example uses *dirExt*() to store the file information for all the files in the root directory of the current drive to the array *aFiles*. The file name and access date is displayed for all the files in the results pane of the Command window. Manifest constants to represent the columns are created with the *#define* preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME      1 // Manifest constants for columns returned by dirExt()
#define ARRAY_DIR_SIZE      2
#define ARRAY_DIR_DATE      3
#define ARRAY_DIR_TIME      4
#define ARRAY_DIR_ATTR      5
#define ARRAY_DIR_SHORT_NAME 6
#define ARRAY_DIR_CREATE_DATE 7
#define ARRAY_DIR_CREATE_TIME 8
#define ARRAY_DIR_ACCESS_DATE 9

aFiles = new Array() // Array will be resized as needed
nFiles = aFiles.dirExt( "\ *.*", "HS" ) // Include Hidden and System files
for nFile = 1 to nFiles
    ? aFiles[ nFile, ARRAY_DIR_NAME ]
    ?? aFiles[ nFile, ARRAY_DIR_ACCESS_DATE ] at 25
endfor
```

See also *dir*(), *sort*()

element()

Returns the number of a specified element in a one- or two-dimensional array.

Syntax <oRef>.element(<subscript1 expN> [, <subscript2 expN>])

<oRef> A reference to a one- or two-dimensional array.

<subscript1 expN> The first subscript of the element. In a one-dimensional array, this is the same as the element number. In a two-dimensional array, this is the row.

<subscript2 expN> When <oRef> is a two-dimensional array, <subscript2 expN> specifies the second subscript, or column, of the element.

If <oRef> is a two-dimensional array and you do not specify a value for <subscript2 expN>, Visual dBASE assumes the value 1, the first column in the row. Visual dBASE generates an error if you use <subscript2 expN> with a one-dimensional array.

Property of Array

Description Use *element*() when you know the subscripts of an element in a two-dimensional array and need the element number for use with another method, such as *fill*() or *scan*().

In one-dimensional arrays, the number of an element is the same as its subscript, so there is no need to use *element()*. For example, if *aOne* is a one-dimensional array, *aOne.element(3)* returns 3, *aOne.element(5)* returns 5, and so on.

element() is the inverse of *subscript()*, which returns an element's row or column subscript number when you specify the element number.

Example The following statement returns the element number of the third column of the fourth row of the array *aTwo*. The result depends on the number of columns in *aTwo*.

```
nElement = aTwo.element( 3, 2 ) // Fourth row, third column
```

See also *fill()*, *insert()*, *scan()*, *size*, *sort()*, *subscript()*

fields()

Fills the array with the current table's structural information. Returns the number of fields whose characteristics are stored.

Syntax *<oRef>.fields()*

<oRef> A reference to the array in which you want to store the field information. *fields()* will automatically redimension or increase the size of the array to accommodate the field information, if necessary.

Property of Array

Description Use *fields()* to store information about the structure of the current table to an array, which is dynamically resized so all returned information fits in the array. The resulting array is always a two-dimensional array, unless there are no table open in the current work area, in which case the array is not modified.

fields() stores the following information for each field in each row of the array. The data type for each is shown in parentheses:

| Column 1 | Column 2 | Column 3 | Column 4 |
|---------------------------|---------------------------|---------------------------|-----------------------------|
| Field name (character) | Field type (character) | Field length (numeric) | Decimal places (numeric) |

Visual dBASE uses the following codes for field types (some codes are used for more than one field type):

| Code | Field type |
|------|-------------------------|
| B | Binary |
| C | Character, Alphanumeric |
| D | Date, Timestamp |

| Code | Field type |
|------|------------------|
| F | Float, Double |
| G | General, OLE |
| L | Logical, Boolean |
| M | Memo |
| N | Numeric |

fields() stores the same information into an array that COPY STRUCTURE EXTENDED stores into a table, except *fields()* doesn't create a column+ containing FIELD_IDX information.

See Also COPY STRUCTURE EXTENDED

fill()

Stores a specified value into one or more locations in an array, and returns the number of elements stored.

Syntax <oRef>.fill(<exp> [, <start expN> [, <count expN>]])

<oRef> A reference to a one- or two-dimensional array you want to fill with the specified value <exp>.

<exp> An expression you want to store in the specified array.

<start expN> The element number at which you want to begin storing <exp>. If you do not specify <start expN>, Visual dBASE begins at the first element in the array.

<count expN> The number of elements in which you want to store <exp>, starting at element <start expN>. If you do not specify <count expN>, Visual dBASE stores <exp> from <start expN> to the last element in the array. If you want to specify a value for <count expN>, you must also specify a value for <start expN>.

If you do not specify <start expN> or <count expN>, Visual dBASE fills all elements in the array with <exp>.

Property of Array

Description Use *fill()* to store a value into all or some elements of an array. For example, if you are going to use elements of an array to calculate totals, you can use *fill()* to initialize all values in the array to 0.

fill() stores values into the array sequentially. Starting at the first element in the array or at the element specified by <start expN>, *fill()* stores the value in each element in a row, then moves to the first element in the next row, continuing to store values until the array is filled or until it has inserted <count expN> elements. *fill()* overwrites any existing data in the array.

If you know an element's subscripts, you can use *element()* to determine its element number for use as *<start expN>*.

Example Suppose you're measuring the performance of a process, keeping track of six different variables, some of which may not be used for any given request. In addition to keeping an average, you want to always display the last three measurements. You can use an array with 3 rows and 6 columns, and *insert()* a new row at the beginning of the array for each request. You *fill()* the new row with zeros to initialize the variables in case they're not used. The code, with simulated input, would look like this:

```
#define SHOW_LAST      3 // Manifest constants for number of measurements
#define NUM_MEASUREMENTS 6 // to maintain
aMeasure = new Array( SHOW_LAST, NUM_MEASUREMENTS )
aMeasure.fill( "" ) // Start with all blanks

// Simulated input
newRequest()
aMeasure[ 1, 1 ] = 34
aMeasure[ 1, 4 ] = 16
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 3 ] = 67
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 1 ] = 27
aMeasure[ 1, 2 ] = 29
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 2 ] = 31
aMeasure[ 1, 6 ] = 40
displayArray( aMeasure, 10 )
// End simulated input

function newRequest()
    aMeasure.insert( 1 ) // Insert row at top, losing last row
    aMeasure.fill( 0, 1, NUM_MEASUREMENTS ) // Fill first row with zeros
```

The *displayArray()* function is used to display the contents of the array in the results pane of the Command window. It is shown in the example for *dimensions* on page 10-13.

See also *element()*

firstKey

Returns the character string key for the first element of an associative array.

Property of AssocArray

Description Use *firstKey* when you want to loop through the elements in an associative array. Once you have gotten the key value for the first element with *firstKey*, use *nextKey*() to get the key values for the rest of the elements.

Note The order of elements in an associative array is undefined. They are not necessarily stored in the order in which you add them, or sorted by their key values. You can't assume that the value returned by *firstKey* will be consistent, or that it will return the first item you added.

For an empty associative array, *firstKey* is the logical value *false*. Because *false* is a different data type than valid key values (which are character strings), it requires extra code to look for *false* to see if the array is empty. It's easier to get the number of elements in the array with *count*() and see if it's greater than zero.

Example The following statements loop through an associative array and display all its elements:

```
aTest = new AssocArray()
aTest[ "USA" ] = "United States of America"
aTest[ "RUS" ] = "Russian Federation"
aTest[ "GER" ] = "Germany"
aTest[ "CHN" ] = "People's Republic of China"

cKey = aTest.firstKey           // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for nElements = 1 to aTest.count()
    ? cKey                      // Display key value
    ? aTest[ cKey ]             // Display element value
    cKey := aTest.nextKey( cKey ) // Get next key value
endfor
```

See also *isKey*(), *nextKey*()

grow()

Adds an element, row, or column to an array and returns the number of added elements.

Syntax <oRef>.grow(<expN>)

<oRef> A reference to a one- or two-dimensional array you want to add elements to.

<expN> Either 1 or 2. When you specify 1, *grow*() adds a single element to a one-dimensional array or a row to a two-dimensional array. When you specify 2, *grow*() adds a column to the array.

Property of Array

Description Use *grow*() to insert an element, row, or column into an array and change the size of the array to reflect the added elements. *grow*() can make a one-

dimensional array two-dimensional. All added elements are initialized to *false* values.

One-dimensional arrays When you specify 1 for *<expN>*, *grow()* adds a single element to the array. When you specify 2, *grow()* makes the array two-dimensional, and existing elements are moved into the first column. This is shown in the following figure:

Figure 10.2 Adding a column to a one-dimensional array using *aAlpha.grow(2)*

aAlpha.grow(2)

- 1 Original array created as:
aAlpha = {"A", "B", "C", "D"}

| | | | |
|-------------|-------------|-------------|-------------|
| 1 A 1 | 2 B 2 | 3 C 3 | 4 D 4 |
|-------------|-------------|-------------|-------------|

Initial contents of the array aAlpha.

- 2 aAlpha.grow(2) adds a new column to the array, makes it a two dimensional array with dimensions [4,2], and copies the old values into the first column.

| | |
|---------------|-------------------|
| 1 A 1,1 | 2 false 1,2 |
| 3 B 2,1 | 4 false 2,2 |
| 5 C 3,1 | 6 false 3,2 |
| 7 D 4,1 | 8 false 4,2 |

*Contents of the array after issuing
aAlpha.grow(2)*

Use *add()* to add a new element to a one-dimensional array and assign its value in one step.

Note You may also assign a new value to the array's *size* property to make a one-dimensional array any arbitrary size.

Two-dimensional arrays When you specify 1 for *<expN>*, *grow()* adds a row to the array at the end of the array. This is shown in the following figure:

Figure 10.3 Adding a row to a two-dimensional array using `aAlpha.grow(1)`**aAlpha.grow(1)****1** Original array created as:

```

aAlpha = new Array(3,4)
aAlpha[1,1] = "A"
aAlpha[1,2] = "B"
⋮
aAlpha[3,4] = "L"

```

| | | | |
|-----------------|-----------------|-----------------|-----------------|
| 1 | 2 | 3 | 4 |
| A 1,1 | B 1,2 | C 1,3 | D 1,4 |
| E 2,1 | F 2,2 | G 2,3 | H 2,4 |
| I 3,1 | J 3,2 | K 3,3 | L 3,4 |

*Initial contents of the array aAlpha.***2** `aAlpha.grow(1)` adds a new row to the array.

| | | | |
|---------------------|---------------------|---------------------|---------------------|
| 1 | 2 | 3 | 4 |
| A 1,1 | B 1,2 | C 1,3 | D 1,4 |
| E 2,1 | F 2,2 | G 2,3 | H 2,4 |
| I 3,1 | J 3,2 | K 3,3 | L 3,4 |
| false 4,1 | false 4,2 | false 4,3 | false 4,4 |

Contents of the array after issuing `aAlpha.grow(1)`

When you specify 2 for `<expN>`, `grow()` adds a column to the array and places `false` into each element in the column.

Example The following example initially declares a one-dimensional array with a single element, and then uses `grow()` to add a second element, convert the array to two dimensions, add a third row, and finally add a third column. The end result is the first nine letters in order:

```

a = {"A"}           // 1-D, 1 element
displayArray( a )
a.grow(1)           // 1-D, 2 elements
a[ 2 ] = "D"
displayArray( a )
a.grow(2)           // 2-D, 2 rows, 2 columns
a[ 1, 2 ] = "B"; a[ 2, 2 ] = "E"
displayArray( a )
a.grow(1)           // 2-D, 3 rows, 2 columns
a[ 3, 1 ] = "G"; a[ 3, 2 ] = "H"
displayArray( a )
a.grow(2)           // 2-D, 3 rows, 3 columns
a[ 1, 3 ] = "C"; a[ 2, 3 ] = "F"; a[ 3, 3 ] = "I"
displayArray( a )

```

The `displayArray()` function is used to display the contents of the array in the results pane of the Command window. It is shown in the example for *dimensions* on page 10-13.

See also `add()`, `size`

insert()

Inserts an element with the value *false* into a one-dimensional array, or inserts a row or column of elements with the value *false* into a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful. The dimensions of the array do not change, so the element(s) at the end of the array will be lost.

Syntax <oRef>.insert(<position expN> [, <row/column expN>])

<oRef> A reference to a one- or two-dimensional array in which you want to insert data.

<position expN> When <oRef> is a one-dimensional array, <position expN> specifies the number of the element in which you want to insert a *false* value.

When <oRef> is a two-dimensional array, <position expN> specifies the number of a row or column in which you want to insert *false* values. The second argument (discussed in the next paragraph) specifies whether <position expN> is a row or a column.

<row/column expN> Either 1 or 2. If you omit this argument or specify 1, a row is inserted into a two-dimensional array. If you specify 2, a column is inserted. Visual dBASE generates an error if you use <row/column expN> with a one-dimensional array.

Property of Array

Description Use *insert()* to insert elements in an array. *insert()* does the following:

- Inserts an element in a one-dimensional array, or inserts a row or column in a two-dimensional array
- Moves all remaining elements toward the end of the array (down if a row is inserted, to the right if an element or column is inserted)
- Stores *false* values in the newly created position(s)

Because the dimensions of the array are not changed, the element(s) at the end of the array—the last element for a one-dimensional array or the last row or column for a two-dimensional array—are lost. If you don't want to lose the data, use *grow()* to increase the size of the array before using *insert()*.

One-dimensional arrays When you call *insert()* for a one-dimensional array, the logical value *false* is inserted into the position of the specified element. The remaining element(s) are moved one place toward the end of the array. The element that had been in the last position is lost.

For example, if you define a one-dimensional array with:

```
aAlpha = {"A", "B", "C"}
```

the resulting array has one row and can be illustrated as follows:

```
A    B    C
```

`insert ()`

Issuing `aAlpha.insert(2)` inserts *false* into element number 2, moves the “B” that was in `aAlpha[2]` to `aAlpha[3]`, and loses the “C” that was in `aAlpha[3]` so that the array now contains these values:

A *false* B

Two-dimensional arrays When you call `insert()` for a two-dimensional array, a logical value *false* is inserted into the position of each element in the specified row or column. The elements in the remaining columns or rows are moved one place toward the end of the array. The elements that had been in the last row or column are lost.

For example, suppose you define a two-dimensional array and store letters to the array. The following illustration shows how the array is changed by `aAlpha.insert(2,2)`.

Figure 10.4 Using insert () with a two-dimensional array**aAlpha.insert(2,2)**

- 1 Original array created as:
 aAlpha = new Array(3,4)
 aAlpha[1,1] = "A"
 aAlpha[1,2] = "B"
 ⋮
 aAlpha[3,4] = "L"

| | | | | | | | |
|---|----------|----|----------|----|----------|----|----------|
| 1 | A 1,1 | 2 | B 1,2 | 3 | C 1,3 | 4 | D 1,4 |
| 5 | E 2,1 | 6 | F 2,2 | 7 | G 2,3 | 8 | H 2,4 |
| 9 | I 3,1 | 10 | J 3,2 | 11 | K 3,3 | 12 | L 3,4 |

Initial contents of the array aAlpha

- 2 aAlpha.insert(2,2)
 inserts logical false values as
 elements in the second column...

| | | | | | | | |
|---|----------|----|----------|----|----------|----|----------|
| 1 | A 1,1 | 2 | B 1,2 | 3 | C 1,3 | 4 | D 1,4 |
| 5 | E 2,1 | 6 | F 2,2 | 7 | G 2,3 | 8 | H 2,4 |
| 9 | I 3,1 | 10 | J 3,2 | 11 | K 3,3 | 12 | L 3,4 |

- 3 Shifts the elements in the
 remaining columns towards the
 end of the array, and deletes the
 elements from the last column.

| | | | | | | | |
|---|----------|----|--------------|----|----------|----|----------|
| 1 | A 1,1 | 2 | false 1,2 | 3 | B 1,3 | 4 | C 1,4 |
| 5 | E 2,1 | 6 | false 2,2 | 7 | F 2,3 | 8 | G 2,4 |
| 9 | I 3,1 | 10 | false 3,2 | 11 | J 3,3 | 12 | K 3,4 |

- 4 Resulting in this array:

| | | | | | | | |
|---|----------|----|--------------|----|----------|----|----------|
| 1 | A 1,1 | 2 | false 1,2 | 3 | B 1,3 | 4 | C 1,4 |
| 5 | E 2,1 | 6 | false 2,2 | 7 | F 2,3 | 8 | G 2,4 |
| 9 | I 3,1 | 10 | false 3,2 | 11 | J 3,3 | 12 | K 3,4 |

*Contents of the array after issuing
aAlpha.insert(2,2)*

Example Suppose you're measuring the performance of a process, keeping track of six different variables, some of which may not be used for any given request. In addition to keeping an average, you want to always display the last three measurements. You can use an array with 3 rows and 6 columns, and insert () a new row at the beginning of the array for each request. You fill () the new row with zeros to initialize the variables in case they're not used. The code, with simulated input, would look like:

```
#define SHOW_LAST      3 // Manifest constants for number of measurements
#define NUM_MEASUREMENTS 6 // to maintain
aMeasure = new Array( SHOW_LAST, NUM_MEASUREMENTS )
aMeasure.fill( "" ) // Start with all blanks

// Simulated input
```

isKey ()

```
newRequest()
aMeasure[ 1, 1 ] = 34
aMeasure[ 1, 4 ] = 16
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 3 ] = 67
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 1 ] = 27
aMeasure[ 1, 2 ] = 29
displayArray( aMeasure, 10 )
newRequest()
aMeasure[ 1, 2 ] = 31
aMeasure[ 1, 6 ] = 40
displayArray( aMeasure, 10 )
// End simulated input

function newRequest()
    aMeasure.insert( 1 )           // Insert row at top, losing last row
    aMeasure.fill( 0, 1, NUM_MEASUREMENTS ) // Fill first row with zeros
```

The *displayArray()* function is used to display the contents of the array in the results pane of the Command window. It is shown in the example for *dimensions* on page 10-13.

See also *delete()*, *fill()*, *grow()*, *resize()*

isKey()

Returns a logical value that indicates if the specified character expression is the key of an element in an associative array.

Syntax <oRef>.isKey(<expC>)

<oRef> A reference to the associative array you want to search.

<expC> The character string you want to find.

Property of AssocArray

Description Use *isKey(<expC>)* to determine if an associative array contains an element with a key value of <expC>. Key values in associative arrays are case-sensitive.

Attempting to access a non-existent key value in an associative array generates an error.

Example The following example uses some test data for the associative array *aCountry*, which associates country codes with their names. The function *countryName()* returns the corresponding country name for a particular code, but if the code is not defined, it returns “Unknown country” instead.

```
aCountry = new AssocArray()
```

```

aCountry[ "USA" ] = "United States of America" // Test data
aCountry[ "RUS" ] = "Russian Federation"
aCountry[ "GER" ] = "Germany"
aCountry[ "CHN" ] = "People's Republic of China"

? countryName( "GER" ) // "Germany"
? countryName( "XYZ" ) // "Unknown country"

function countryName( cArg )
    // Make sure code is defined before trying to reference it
    return iif( aCountry.isKey( cArg ), aCountry[ cArg ], "Unknown country" )

```

See also *firstKey*, *nextKey*(), *removeKey*()

nextKey()

Returns the key value of the element following the specified key in an associative array.

Syntax `<oRef>.nextKey(<key expC>)`

<oRef> A reference to the associative array that contains the key.

<key expC> An existing key value.

Property of AssocArray

Description Use *nextKey*() to loop through the elements in an associative array. Once you have gotten the key value for the first element with *firstKey*, use *nextKey*() to get the key values for the rest of the elements.

nextKey() returns the key value for the key following *<key expC>*. Key values in associative arrays are case-sensitive. For the last key in the associative array and for a *<key expC>* that is not an existing key value, *nextKey*() returns the logical value *false*. Because *false* is a different data type than valid key values (which are character strings), it's difficult to look for *false* to terminate a loop. It's easier to get the number of elements in the array first with *count*(); then loop through that many iterations.

Note The order of elements in an associative array is undefined. They are not necessarily stored in the order in which you add them, or sorted by their key values. You can't assume that the sequence of keys will be consistent.

To determine if a given character string is a key value in an associative array, use *isKey*().

Example The following statements loop through an associative array and display all its elements:

```

aTest = new AssocArray()
aTest[ "USA" ] = "United States of America"
aTest[ "RUS" ] = "Russian Federation"
aTest[ "GER" ] = "Germany"

```

`removeAll ()`

```
aTest[ "CHN" ] = "People's Republic of China"

cKey = aTest.firstKey           // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for nElements = 1 to aTest.count()
    ? cKey                     // Display key value
    ? aTest[ cKey ]            // Display element value
    cKey := aTest.nextKey( cKey ) // Get next key value
endfor
```

See also *firstKey, isKey()*

removeAll()

Deletes all elements from an associative array.

Syntax `<oRef>.removeAll()`

<oRef> A reference to the associative array you want to empty.

Property of AssocArray

Description Use *removeAll()* to remove all the elements from an associative array.
To remove elements for particular key values, use *removeKey()*.

Example The following example removes all elements from an associative array.

```
var aTest = new AssocArray()
aTest[ "USA" ] = "United States of America"
aTest[ "RUS" ] = "Russian Federation"
aTest[ "GER" ] = "Germany"
aTest[ "CHN" ] = "People's Republic of China"
// Array contains four elements
aTest.removeAll() // Array now contains no elements
```

See also *removeKey()*

removeKey()

Deletes an element from an associative array.

Syntax `<oRef>.removeKey(<key expC>)`

<oRef> A reference to the associative array that contains the key.

<key expC> The key value of the element you want to delete.

Property of AssocArray

Description Use *removeKey()* to remove an element from an associative array. Key values in associative arrays are case-sensitive.

If you specify a key value that does not exist in the array, nothing happens; no error occurs and no elements are removed.

To remove all the elements from an associative array, use *removeAll()*.

Example The following example loops through an associative array of country names and deletes those whose names are longer than 15 characters.

```
aTest = new AssocArray()
aTest[ "USA" ] = "United States of America"
aTest[ "RUS" ] = "Russian Federation"
aTest[ "GER" ] = "Germany"
aTest[ "CHN" ] = "People's Republic of China"

cKey = aTest.firstKey           // Get first key in AssocArray
// Get number of elements in AssocArray and loop through them
for nElements = 1 to aTest.count()
    cNextKey = aTest.nextKey( cKey ) // Get next key value before deleting element
    if len( aTest[ cKey ] ) > 15
        aTest.removeKey( cKey ) // Remove element
    endif
    cKey := cNextKey           // Use next key value
endfor
```

Note that you must get the next key value before deleting the element, and you repeat the loop based on the number of elements there were before you started deleting.

See also *isKey()*, *removeAll()*

resize()

Sets the size of an array to the specified dimensions and returns a numeric value representing the number of elements in the modified array.

Syntax `<oRef>.resize(<rows expN> [, <cols expN> [, <retain values expN>]])`

<oRef> A reference to the array whose size you want to change.

<rows expN> The number of rows the resized array should have. *<rows expN>* must always be a positive, nonzero value.

<cols expN> The number of columns the resized array should have. *<cols expN>* must always be 0 or a positive value. If you omit this option, *resize()* changes the number of rows in the array and leaves the number of columns the same.

<retain values expN> Determines what happens to the values of the array when rows are added or removed. If it is nonzero, values are retained. If you

want to specify a value for *<retain values expN>*, you must also specify a value for *<new cols expN>*.

Property of Array

Description Use *resize ()* to change the dimensions of an array, making it larger or smaller, or change the number of dimensions. To determine the number of dimensions, check the array's *dimensions* property. The *size* property of the array reflects the number of elements; for a one-dimensional array, that's all you need to know. For a two-dimensional array, you can't determine the number of rows or columns from the *size* property alone (unless the *size* is one—a one-by-one array). To determine the number of columns or rows in a two-dimensional array, use the *ALen ()* function.

For a one-dimensional array, you can change the number of elements by calling *resize ()* and specifying the number of elements as *<rows expN>* parameter. You can also set the *size* property of the array directly, which is a bit less typing.

You can also change a one-dimensional array into a two-dimensional array by specifying both a *<rows expN>* and a nonzero *<cols expN>* parameter. This makes the array the designated size.

For a two-dimensional array, you can specify a new number of rows or both row and column dimensions for the array. If you omit *<cols expN>*, the *<rows expN>* parameter sets the number of rows only. With both a *<rows expN>* and a nonzero *<cols expN>*, the array is changed to the designated size.

You can change a two-dimensional array to a one-dimensional array by specifying *<cols expN>* as zero and *<rows expN>* as the number of elements.

To change the number of columns only for a two-dimensional array, you will need to specify both the *<rows expN>* and *<cols expN>* parameters, which means that you have to determine the number of rows in the array, if not known, and specify it unchanged as the *<rows expN>* parameter.

To add a single row or column to an array, use the *grow ()* method.

If you add or remove columns from the array, you can use *<retain values expN>* to specify how you want existing elements to be placed in the new array. If *<retain values expN>* is zero or isn't specified, *resize ()* rearranges the elements, filling in the new rows or columns or adjusting for deleted elements, and adding or removing elements at the end of the array, as needed. This is shown in the following two figures. You are most likely to want to do this if you don't need to refer to existing items in the array; that is, you plan to update the array with new values.

Figure 10.5 Adding a row and a column to a 3x4 array, rearranging elements**aAlpha.resize(4,5)**

- 1 Original array created as:
 aAlpha = new Array(3,4)
 aAlpha[1,1] = "A"
 aAlpha[1,2] = "B"
 ⋮
 aAlpha[3,4] = "L"

| | | | |
|----------|----------|----------|----------|
| 1 | 2 | 3 | 4 |
| A 1,1 | B 1,2 | C 1,3 | D 1,4 |
| 5 | 6 | 7 | 8 |
| E 2,1 | F 2,2 | G 2,3 | H 2,4 |
| 9 | 10 | 11 | 12 |
| I 3,1 | J 3,2 | K 3,3 | L 3,4 |

Initial contents of the array aAlpha.

- 2 aAlpha.resize(4,5) adds a new row and column to the array and rearranges the values of the elements.

| | | | | |
|--------------|--------------|--------------|--------------|--------------|
| 1 | 2 | 3 | 4 | 5 |
| A 1,1 | B 1,2 | C 1,3 | D 1,4 | E 1,5 |
| 6 | 7 | 8 | 9 | 10 |
| F 2,1 | G 2,2 | H 2,3 | I 2,4 | J 2,5 |
| 11 | 12 | 13 | 14 | 15 |
| K 3,1 | L 3,2 | false 3,3 | false 3,4 | false 3,5 |
| 16 | 17 | 18 | 19 | 20 |
| false 4,1 | false 4,2 | false 4,3 | false 4,4 | false 4,5 |

*Contents of the array after issuing
aAlpha.resize(4,5)***Figure 10.6** Adding a column to a one-dimensional array, rearranging elements**aAlpha.resize(4,2)**

- 1 Original array created as:
 aAlpha = {"A", "B", "C", "D"}

| | | | |
|----------|----------|----------|----------|
| 1 | 2 | 3 | 4 |
| A 1,1 | B 1,2 | C 1,3 | D 1,4 |

Initial contents of the array aAlpha.

- 2 aAlpha.resize(4,2) adds a new column to the array, makes it a two dimensional array with dimensions [4,2], and reassigns the values of the elements.

| | |
|--------------|--------------|
| 1 | 2 |
| A 1,1 | B 1,2 |
| 3 | 4 |
| C 2,1 | D 2,2 |
| 5 | 6 |
| false 3,1 | false 3,2 |
| 7 | 8 |
| false 4,1 | false 4,2 |

*Contents of the array after issuing
aAlpha.resize(4,2)*

When you use `resize()` on a one-dimensional array, you might want the original row to become the first column of the new array. Similarly, when you use `resize()` on a two-dimensional array, you might want existing two-dimensional array elements to remain in their original positions. You are most likely to want to do this if you need to refer to existing items in the array by their subscripts; that is, you plan to add new values to the array while continuing to work with existing values.

If *<retain values expN>* is a nonzero value, *resize()* ensures that elements retain their original values. The following two figures repeat the statements shown in the previous two figures, with the addition of a value of 1 for *<retain values expN>*.

Figure 10.7 Adding a row and a column to a 3x4 array, “preserving elements”

aAlpha.resize(4,5,1)

- 1 Original array created as:
 aAlpha = new Array(3,4)
 aAlpha[1,1] = “A”
 aAlpha[1,2] = “B”
 ⋮
 aAlpha[3,4] = “L”

| | | | |
|---------------|----------------|----------------|----------------|
| 1 A 1,1 | 2 B 1,2 | 3 C 1,3 | 4 D 1,4 |
| 5 E 2,1 | 6 F 2,2 | 7 G 2,3 | 8 H 2,4 |
| 9 I 3,1 | 10 J 3,2 | 11 K 3,3 | 12 L 3,4 |

Initial contents of the array aAlpha.

- 2 aAlpha.resize(4,5,1) adds a new row and column to the array and maintains the values of the elements.

| | | | | |
|--------------------|--------------------|--------------------|--------------------|--------------------|
| 1 A 1,1 | 2 B 1,2 | 3 C 1,3 | 4 D 1,4 | 5 false 1,5 |
| 6 E 2,1 | 7 F 2,2 | 8 G 2,3 | 9 H 2,4 | 10 false 2,5 |
| 11 I 3,1 | 12 J 3,2 | 13 K 3,3 | 14 L 3,4 | 15 false 3,5 |
| 16 false 4,1 | 17 false 4,2 | 18 false 4,3 | 19 false 4,4 | 20 false 4,5 |

Contents of the array after issuing aAlpha.resize(4,5,1)

Figure 10.8 Adding a column to a one-dimensional array, “preserving elements”

aAlpha.resize(4,2,1)

- 1 Original array created as:
 aAlpha = {“A”, “B”, “C”, “D”}

| | | | |
|---------------|---------------|---------------|---------------|
| 1 A 1,1 | 2 B 1,2 | 3 C 1,3 | 4 D 1,4 |
|---------------|---------------|---------------|---------------|

Initial contents of the array aAlpha.

- 2 aAlpha.resize(4,2,1) adds a new column to the array, and makes it a two-dimensional array with dimensions [4,2]. Each existing element is now the first element in a row.

| | |
|---------------|-------------------|
| 1 A 1,1 | 2 false 1,2 |
| 3 B 2,1 | 4 false 2,2 |
| 5 C 3,1 | 6 false 3,2 |
| 7 D 4,1 | 8 false 4,2 |

Contents of the array after issuing aAlpha.resize(4,2,1)

Example The following example resizes a one-dimensional array with 12 elements into a two-dimensional array with 3 rows and four columns.

```
aAlpha = {"A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L" }
aAlpha.resize( 3, 4 )
displayArray( aAlpha )
```

The *displayArray()* function is used to display the contents of the array in the results pane of the Command window. It is shown in the example for *dimensions* on page 10-13.

See also *ALen()*, *grow()*, *size*

scan()

Searches an array for an expression. Returns the number of the first element that matches the expression if the search is successful, or 0 if the search is unsuccessful.

Syntax *<oRef>.scan(<exp> [, <starting element expN> [, <elements expN>]])*

<oRef> A reference to the array you want to search.

<exp> The expression you want to search for in *<oRef>*.

<starting element expN> The element number in *<oRef>* at which you want to start searching. Without *<starting element expN>*, *scan()* starts searching at the first element.

<elements expN> The number of elements in *<oRef>* that *scan()* searches. Without *<elements expN>*, *scan()* searches *<oRef>* from *<starting element expN>* to the end of the array. If you want to specify a value for *<elements expN>*, you must also specify a value for *<starting element expN>*.

Property of Array

Description Use *scan()* to search an array for the value of *<exp>*. For example, if an array contains customer names, you can use *scan()* to find the location in which a particular name appears.

scan() returns the element number of the first element in the array that matches *<exp>*. If you want to determine the subscripts of this element, use *subscript()*.

When *<exp>* is a string, *scan()* is case-sensitive; you may want to use *UPPER()*, *LOWER()*, or *PROPER()* to match the case of *<exp>* with the case of the data stored in the array. *scan()* also follows the rules established by SET EXACT to determine if *<exp>* and the array element are equal. For more information, see SET EXACT.

Example The following example uses *dir()* to store the file information for all the files and directories in the root directory of the current drive to the array *aFiles*. Then the array is searched to display only the directories in the array. Manifest constants to represent the columns are created with the *#define* preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 1    // Manifest constants for columns returned by dir()
```

```

#define ARRAY_DIR_SIZE 2
#define ARRAY_DIR_DATE 3
#define ARRAY_DIR_TIME 4
#define ARRAY_DIR_ATTR 5

aFiles = new Array()
nFiles = aFiles.dir( "\\*.*", "D" )           // Read all files and directories

nElement = 1                                // Start looking at first element
do
    nElement = aFiles.scan( "....D", nElement ) // Look for next directory
    if nElement > 0                             // Display a match
        ? aFiles[ nElement - ARRAY_DIR_ATTR + ARRAY_DIR_NAME ]
        if nElement++ >= aFiles.size             // Continue looking with next element
            exit                                  // Unless that was the last element
        endif
    endif
until nElement == 0                          // Until there's no match

```

To find all the matches in the array, you need to keep track of the last match. Here it's kept in the variable *nElement*. It starts at one, the first element, and is used in the *scan()* call as the starting element parameter. The result of each search is stored back in *nElement*. If there's a match, the directory name is displayed. Then *nElement* is incremented—otherwise *scan()* would match the same element again—and the loop continues.

A few subtleties are present in the example code. First, when incrementing *nElement*, it is compared with the *size* of the array. If the element number is equal to (or greater than, which it should never be, but it's good defensive programming to test for it anyway) the *size* of the array, that means the last match was in the last element of the array. This is possible only because the file attribute is in the last column of the array. In this case, you don't want to call *scan()* again, since the starting element number is higher than the highest element number and would cause an error. So you EXIT out of the loop instead.

The variable *nElement* is incremented after the comparison to the *size* of the array by using the postfix ++ operator. If *nElement* was pre-incremented, the comparison would be off, although the rest of the loop would work.

To display the directory name, the column number of the file attribute column is subtracted from the matching element number, and the column number for the file name column is added. This yields the element number of the file name in the same row as the matching file attribute. This would work for any combination of search or display columns.

See also SET EXACT, *sort()*, *subscript()*

size

The number of elements in an Array object.

Property of Array

Description *size* indicates the number of elements in an Array object.

For a one-dimensional array, you can assign a value to *size* to change its size.

For a array with more than one dimension, *size* is read-only.

You can use the ALLEN() function to determine the size of each dimension for a two-dimensional array. There is no built-in way to determine dimension sizes for arrays with more than two dimensions.

Example The following classes implement a simple stack based on the Array class. When items are pushed onto the stack, the *size* of the array is checked to see if the item will fit. If not, the *add* method() is used. When items are popped from the stack, the array does not shrink. If another item is pushed onto the stack, the item can be stored in an existing array element.

```
class Stack of Array
  this.ptr = 0

  function push( xArg )
    if this.ptr == this.size
      this.add( xArg )
      this.ptr++
    else
      this[ ++this.ptr ] = xArg
    endif
  endfunction

  function pop()
    if this.ptr > 0
      return this[ this.ptr-- ]
    else
      throw new StackException()
    endif
  endfunction

  function top()
    if this.ptr > 0
      return this[ this.ptr ]
    else
      throw new StackException()
    endif
  endfunction

  function empty()
    return this.ptr == 0
  endfunction
endclass

class StackException of Exception
  this.message = "Stack fault"
endclass
```

See also ALLEN(), *dimensions*, *resize*(), *subscript*()

sort()

Sorts the elements in a one-dimensional array or the rows in a two-dimensional array. Returns 1 if successful; generates an error if unsuccessful.

Syntax <oRef>.sort([<starting element expN> [, <elements to sort expN> [, <sort order expN>]])

<oRef> A reference to the array you want to sort.

<starting element expN> In a one-dimensional array, the number of the element in <oRef> at which you want to start sorting. In a two-dimensional array, the number (subscript) of the column on which you want to sort. Without <starting element expN>, sort() starts sorting at the first element or column in the array.

<elements to sort expN> In a one-dimensional array, the number of elements you want to sort. In a two-dimensional array, the number of rows to sort. Without <elements to sort expN>, sort() sorts the rows starting at the row containing element <starting element expN> to the last row. If you want to specify a value for <elements to sort expN>, you must also specify a value for <starting element expN>.

<sort order expN> The sort order:

- 0 specifies ascending order (the default)
- 1 specifies descending order

If you want to specify a value for <sort order expN>, you must also specify values for <elements to sort expN> and <starting element expN>.

Property of Array

Description sort() requires that all the elements on which you're sorting be of the same data type. The elements to sort in a one-dimensional array must be of the same data type, and the elements of the column by which rows are to be sorted in a two-dimensional array must be of the same data type.

sort() arranges elements in alphabetical, numerical, chronological, or logical order, depending on the data type of <starting element expN>. (For strings, the current language driver determines the sort order.)

One-dimensional arrays Suppose you create an array with the following statement:

```
aNums = {5, 7, 3, 9, 4, 1, 2, 8}
```

That creates an array with the elements in this order:

```
5    7    3    9    4    1    2    8
```

If you call *aNums.sort(1, 5)*, Visual dBASE sorts the first five elements so that the array elements are in this order:

```
3    4    5    7    9    1    2    8
```

If you then call *aNums.sort(5, 2)*, Visual dBASE sorts two elements starting at the fifth element so that the array elements are now in this order:

```
3    4    5    7    1    9    2    8
```


Two-dimensional arrays Using `sort()` with a two-dimensional array is similar to using the SORT command with a table. Array rows correspond to records, and array columns correspond to fields.

When you sort a two-dimensional array, whole rows are sorted, not just the elements in the column where *<starting element expN>* is located.

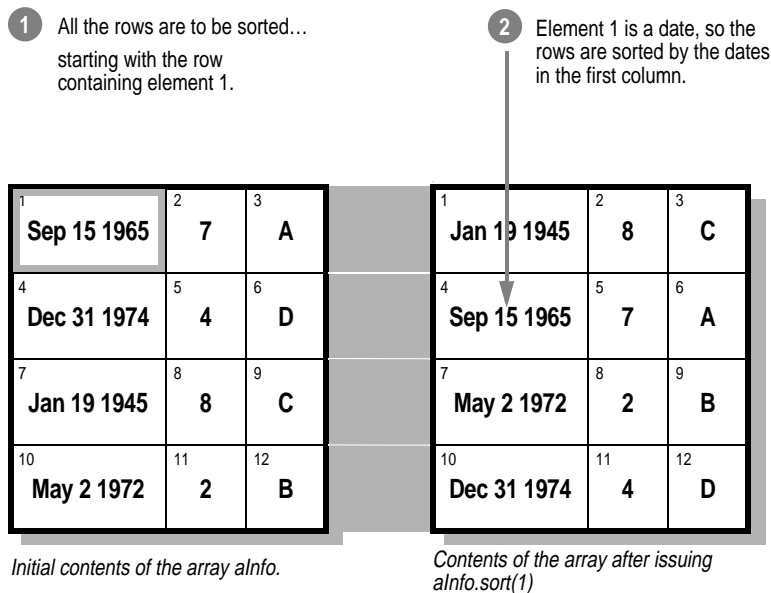
For example, suppose you create the array `aInfo` and fill it with the following data:

| | | |
|-------------|---|---|
| Sep 15 1965 | 7 | A |
| Dec 31 1965 | 4 | D |
| Jan 19 1945 | 8 | C |
| May 2 1972 | 2 | B |

If you call `aInfo.sort(1)`, Visual dBASE sorts all rows in the array beginning with element number 1. The rows are sorted by the dates in the first column because element 1 is a date. The following figure shows the results.

Figure 10.9 `aInfo.sort(1)`

`aInfo.sort(1)`



If you then call `aInfo.sort(5, 2)`, Visual dBASE sorts two rows in the array starting with element number 5, whose value is 7. `sort()` sorts the second and the third rows based on the numbers in the second column. The following figure shows the results.

subscript ()

Figure 10.10 Using sort() with a two-dimensional array

alInfo.sort(5,2)

- 1

Two rows are to be sorted (alInfo.sort(5,2)) starting with the row containing element 5 (alInfo.sort(5,2)).
- 2

Element 5 contains a number, so the rows are sorted by the numbers in the second column.

| | | | | | |
|----|-------------|----|---|----|---|
| 1 | Jan 19 1945 | 2 | 8 | 3 | C |
| 4 | Sep 15 1965 | 5 | 7 | 6 | A |
| 7 | May 2 1972 | 8 | 2 | 9 | B |
| 10 | Dec 31 1974 | 11 | 4 | 12 | D |

Initial contents of the array alInfo.

| | | | | | |
|----|-------------|----|---|----|---|
| 1 | Jan 19 1945 | 2 | 8 | 3 | C |
| 4 | May 2 1972 | 5 | 2 | 6 | B |
| 7 | Sep 15 1965 | 8 | 7 | 9 | A |
| 10 | Dec 31 1974 | 11 | 4 | 12 | D |

Contents of the array after issuing alInfo.sort(5, 2)

Example The following example uses `dir()` to store the file information for all the files in the current directory to the array `aFiles`. Then the array is sorted on the file size. Manifest constants to represent the columns are created with the `#define` preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 1 // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 2
#define ARRAY_DIR_DATE 3
#define ARRAY_DIR_TIME 4
#define ARRAY_DIR_ATTR 5

aFiles = new Array()
nFiles = aFiles.dir()
aFiles.sort( ARRAY_DIR_SIZE ) // Sort by size

for nFile = 1 to nFiles
    ? aFiles[ nFile, ARRAY_DIR_NAME ]
    ?? aFiles[ nFile, ARRAY_DIR_SIZE ] at 25
endfor
```

See also `element()`, `scan()`, `subscript()`

subscript()

Returns the row number or the column number of a specified element in an array.

Syntax <oRef>.subscript(<element expN>, <row/column expN>)

<oRef> A reference to the array.

<element expN> The element number.

<row/column expN> A number, either 1 or 2, that determines whether you want to return the row or column subscript of an array. If **<row/column expN>** is 1, `subscript()` returns the number of the row subscript. If **<row/column expN>** is 2, `subscript()` returns the number of the column subscript.

If **<oRef>** is a one-dimensional array, *Visual dBASE* returns an error if **<row/column expN>** is a value other than 1.

Property of Array

Description Use `subscript()` when you know the number of an element in a two-dimensional array and want to reference the element by using its subscripts.

If you need to determine both the row and column number of an element in a two-dimensional array, call `subscript()` twice, once with a value of 1 for **<row/column expN>** and once with a value of 2 for **<row/column expN>**. For example, if the element number is in the variable *nElement*, execute the following statements to get its subscripts:

```
nRow = aExample.subscript( nElement, 1 )
nCol = aExample.subscript( nElement, 2 )
```

In one-dimensional arrays, the number of an element is the same as its subscript, so there is no need to use `subscript()`. For example, if *aOne* is a one-dimensional array, *aOne.subscript(3)* returns 3, *aOne.subscript(5)* returns 5, and so on.

`subscript()` is the inverse of `element()`, which returns an element number when you specify the element subscripts.

Example The following example displays all the nonzero-size files in your Windows Temp directory. First it tries to find the directory where your temporary files are stored by looking for the operating system environment variable TMP. Then it uses `dir()` to store the file information for all the files in that directory (or the current directory if the TMP directory is not found) to the array *aFiles*. All the rows that have a file size of zero are deleted using a combination of `scan()`, `subscript()`, and `delete()`.

`scan()` can simply search for zeros because there are no other numeric columns in the array created by `dir()`. If it finds one, `subscript()` is called to return the corresponding row number for the matching element. Then the row number is used in the `delete()` call.

Manifest constants to represent the columns are created with the `#define` preprocessor directive to make the code more readable.

```
#define ARRAY_DIR_NAME 1 // Manifest constants for columns returned by dir()
#define ARRAY_DIR_SIZE 2
#define ARRAY_DIR_DATE 3
#define ARRAY_DIR_TIME 4
#define ARRAY_DIR_ATTR 5

// Look for OS environment variable TMP
```

subscript ()

```
cTempDir = getenv( "TMP" )
// If defined, make sure it has trailing backslash
if "" # cTempDir
    if right( cTempDir, 1 ) # "\" // No trailing backslash
        cTempDir += "\"          // so add one
    endif
endif

aFiles = new Array()
nFiles = aFiles.dir( cTempDir + ".*" ) // Read all files in TMP dir
nElement = aFiles.scan( 0 )
do while nElement > 0 // Find zero-byte files and
    aFiles.delete( aFiles.subscript( nElement, 1 ), 1 ) // delete by row
    nFiles-- // Decrement file count
    nElement := aFiles.scan( 0 )
enddo

for nFile = 1 to nFiles // Display results
    ? aFiles[ nFile, ARRAY_DIR_NAME ]
    ?? aFiles[ nFile, ARRAY_DIR_SIZE ] at 25
endfor
```

See also *element()*

File/OS

File commands and functions

Visual dBASE supports equivalent file commands and functions for all the methods in the File class, which can be organized into the following categories:

- File utility commands
- File information functions
- Functions that provide byte-level access to files, sometimes referred to as low-level file functions

The low-level file functions are maintained for compatibility. To read and write to files, you should use a File object, which better encapsulates direct file access. In contrast, the file utility commands and file information functions are easier to use, because they do not require the existence of a File object.

File utility commands

The following commands have equivalent methods in the File class:

| Command | File class method |
|-------------|-------------------|
| COPY FILE | <i>copy()</i> |
| DELETE FILE | <i>delete()</i> |
| ERASE | <i>delete()</i> |
| RENAME | <i>rename()</i> |

These commands are described separately to document their syntax. Otherwise, they perform identically to their equivalent method.

File information functions

The following file information functions are usually used instead of their equivalent methods in the File class:

| Function | File class method |
|---------------|---------------------|
| FACCESSDATE() | <i>accessDate()</i> |
| FCREATEDATE() | <i>createDate()</i> |
| FCREATETIME() | <i>createTime()</i> |
| FDATE() | <i>date()</i> |
| FILE() | <i>exists()</i> |
| FSHORTNAME() | <i>shortName()</i> |
| FSIZE() | <i>size()</i> |
| FTIME() | <i>time()</i> |

These functions are not described separately (except for FILE(), because its name is not based on the name of its equivalent method). The syntax of a file information function is identical to the syntax of the equivalent method, except that as a function, no reference to a File object is needed, which makes the function more convenient to use. For example, these two statements are equivalent

```
nSize = fsize( cFile )           // Get size of file named in variable cFile
nSize = new File().size( cFile ) // Get size of file named in variable cFile
```

Low-level file functions

The following low-level file functions are equivalent to the following methods in the File class:

| Function | File class method |
|-----------|------------------------------------|
| FCLOSE() | <i>close()</i> |
| FCREATE() | <i>create()</i> |
| FEOF() | <i>eof()</i> |
| FERROR() | <i>error()</i> |
| FFLUSH() | <i>flush()</i> |
| FGETS() | <i>gets()</i> and <i>readln()</i> |
| FOPEN() | <i>open()</i> |
| FPUTS() | <i>puts()</i> and <i>writeln()</i> |
| FREAD() | <i>read()</i> |
| FSEEK() | <i>seek()</i> |
| FWRITE() | <i>write()</i> |

These functions are not described separately. While a File object automatically maintains its file handle in its *handle* property, low-level file functions must explicitly specify a file handle, with the exception of `FERROR()`, which does not act on a specific file. The `FCREATE()` and `FOPEN()` functions take the same parameters as the *create()* and *open()* methods, and return the file handle.

The other functions use the file handle as their first parameter and all other parameters (if any) following it. The parameters following the file handle in the function are identical to the parameters to the equivalent method, and the functions return the same values as the methods.

Compare the examples for *exists()* and *FILE()* to see the difference between using a File object and low-level file functions.

class File

An object that provides byte-level access to files and contains various file directory methods.

Syntax [*<oRef>* =] new File()

<oRef> A variable or property in which to store a reference to the newly created File object.

Properties The following tables list the properties and methods of the File class. (No events are associated with this class.)

| Property | Default | Description |
|------------------|---------|--|
| <i>className</i> | FILE | Identifies the object as an instance of the File class (Property discussed in Chapter 5, "Core language.") |
| <i>handle</i> | -1 | Operating system file handle |
| <i>path</i> | | Full path and file name for open file |
| <i>position</i> | 0 | Current position of file pointer, relative to the start of the file |

| Method | Parameters | Description |
|---------------------|--|--|
| <i>accessDate()</i> | <filename expC> | Returns the last date a file was opened |
| <i>close()</i> | | Closes the currently open file |
| <i>copy()</i> | <filename expC> , <new name expC> | Makes a copy of the specified file |
| <i>create()</i> | <filename expC> [, <access rights>] | Creates a new file with optional access attributes |
| <i>createDate()</i> | <filename expC> | Returns the date when the file was created |
| <i>createTime()</i> | <filename expC> | Returns the time a file was created as a string |
| <i>date()</i> | <filename expC> | Returns the date the file was last modified |
| <i>delete()</i> | <filename expC> | Deletes the specified file |

| Method | Parameters | Description |
|--------------------|---|---|
| <i>eof()</i> | | Returns <i>true</i> or <i>false</i> indicating if the file pointer is positioned past the end of the currently open file |
| <i>error()</i> | | Returns a number indicating the last error encountered |
| <i>exists()</i> | <filename expC> | Returns <i>true</i> or <i>false</i> to indicate whether the specified disk file exists |
| <i>flush()</i> | | Writes current data in the file buffer to disk and keeps file open |
| <i>gets()</i> | [<chars read expN>] [, <eol expC>] | Reads and returns a line from a file, leaving the file pointer at the beginning of the next line. Same as <i>readln()</i> |
| <i>open()</i> | <filename expC> [, <access rights>] | Opens an existing file with optional access attributes |
| <i>puts()</i> | <input string expC> [, <max chars expN>] [, <eol expC>] | Writes a character string and end-of-line character(s) to a file. Same as <i>writeln()</i> |
| <i>read()</i> | <characters expN> | Reads and returns the specified number of characters from the file starting from the current file pointer position; leaving the file pointer at the character after the last one read |
| <i>readln()</i> | [<chars read expN>] [, <eol expC>] | Reads and returns a line from a file, leaving the file pointer at the beginning of the next line. Same as <i>gets()</i> . |
| <i>rename()</i> | <filename expC> , <new name expC> | Changes the name of the specified file to a new name |
| <i>seek()</i> | <offset expN> [, 0 1 2] | Moves the file pointer the specified number of bytes within a file, optionally allowing the movement to be from the beginning (0), end (2), or current file position (1) |
| <i>shortName()</i> | <filename expC> | Returns the short (8.3) name for a file |
| <i>size()</i> | <filename expC> | Returns the number of bytes in the specified file |
| <i>time()</i> | <filename expC> | Returns the time the file was last modified as a string |
| <i>write()</i> | <expC> [, <max chars expN>] | Writes the specified string into the file at the current file position, overwriting any existing data and leaving the file pointer at the character after the last character written |
| <i>writeln()</i> | <input string expC> [, <max chars expN>] [, <eol expC>] | Writes a character string and end-of-line character(s) to a file. Same as <i>puts()</i> . |

Description

Use a File object for direct byte-level access to files. Once you create a new File object, you can *open()* an existing file or *create()* a new one. Be sure to *close()* the file when you are done. A File object may access only one file at a time, but after closing a file, you may open or create another.

When reading or writing to a binary file, be sure to specify the "B" binary access specifier. Without it, the file is treated as a text file; if the current language driver is a multi-byte language, each character in the file may be one or two

bytes. Binary access ensures that each byte is read and written without translation.

File objects also contain information and utility methods for file directories, such as returning the size of a file or changing a file name. If you intend to call multiple methods, you can create and reuse a File object. For example,

```
ff = new File()
? ff.size( "VDB.HLP" )
? ff.accessDate( "VDB.HLP" )
```

Or you can create a File object for a WITH block. For example,

```
with new File()
? size( "VDB.HLP" )
? accessDate( "VDB.HLP" )
endwith
```

For a single call, you can create and use the File object in the same statement:

```
? new File().size( "VDB.HLP" )
```

However, unless you happen to have a File object handy, it's easier to use the equivalent built-in function or command to get the file information or perform the file operation:

```
? fsize( "VDB.HLP" )
? faccessdate( "VDB.HLP" )
```

Example Suppose you have a data file generated by a mainframe computer that has fixed length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"

fIn  = new File()
fOut = new File()

fIn.open( IN_FILE );
fOut.create( OUT_FILE );

do while not fIn.eof()
  fOut.puts( fIn.read( REC_LENGTH ) ) // Read fixed length; write with line break
enddo

fIn.close()
fOut.close()
```

See also none

!

Executes a program or operating system command from within *Visual dBASE*.

`accessDate ()`

Syntax `! <command>`

<command> A command recognized by your operating system.

Description `!` is identical to `RUN`, except that a space is not required after the `!` symbol, while a space is required after the word `RUN`. See `RUN` for details.

See Also `DOS`, `RUN`, `RUN()`

accessDate()

Returns the last date a file was opened.

Syntax `<oRef>.accessDate(<filename expC>)`

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory, then in the search path(s) you specified with `SET PATH`, if any. If you specify a file without including its extension, *Visual dBASE* assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description `accessDate()` checks the file specified by `<filename expC>` and returns the date that the file was last opened by the operating system for reading or writing.

To get the date the file was last modified, use `date()`. For the date the file was created, use `createDate()`.

Example The following example uses `accessDate()` to display the last date the Help file was used:

```
? new File().accessDate( "C:\Program Files\dBASE\Visual dBASE 75\Help\VDB.HLP" )
```

See also `createDate()`, `createTime()`, `date()`, `time()`

CD

Changes the current drive or directory.

Syntax `CD [<path>]`

<path> The new drive and/or path. Quotes (single or double) are required if the path contains spaces or other special characters; optional otherwise.

Description Use CD to change the current working directory in *Visual* dBASE to any valid drive and path. If you're unsure whether a drive is valid, use VALIDDRIVE() before issuing CD. The current directory appears in the Navigator.

CD supports the Universal Naming Convention (UNC), which starts with double backslashes for the resource name, for example:

```
\\MyServer\MyVolume\MyDir\MySubdir
```

CD without the option *<path>* displays the current drive and directory path in the result pane of the Command window. To get the current directory, use SET("DIRECTORY").

Another way to access files on different directories is with the command SET PATH. You can specify one or more search paths, and *Visual* dBASE uses these paths to locate files not on the current directory. Use SET PATH when an application's files are in several directories.

CD works like SET DIRECTORY, except SET DIRECTORY TO (with no argument) returns you to the HOME() directory, instead of displaying the current directory.

See Also HOME(), MKDIR, SET DIRECTORY, SET PATH, VALIDDRIVE()

close()

Closes a file previously opened with *create()* or *open()*.

Syntax <oRef>.close()

<oRef> A reference to the File object that created or opened the file.

Property of File

Description *close()* closes a file you've opened with *create()* or *open()*. *close()* returns *true* if it's able to close the file. If the file is no longer available (for example, the file was on a floppy disk that has been removed) and there is data in the buffer that has not yet been written to disk, *close()* returns *false*.

Always close the file when you're done with it.

To save the file to disk without closing it, use *flush()*.

Example The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"

f = new File()
if f.exists( LOG_FILE )
```

`copy ()`

```
f.open( LOG_FILE, "A" )
else
    f.create( LOG_FILE, "A" )
endif
f.puts( new Date().toLocaleString() )
f.close()
```

See also `create()`, `flush()`, `open()`

`close()` is also a method of the Database (page 14-59) and Form (page 15-74) classes.

copy()

Duplicates a specified file.

Syntax `<oRef>.copy(<filename expC>, <new name expC>)`

<oRef> A reference to a File object.

<filename expC> Identifies the file to duplicate (also known as the source file). `<filename expC>` may be a file name skeleton with wildcard characters. In that case, *Visual dBASE* displays a dialog box in which you select the file to duplicate.

If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual dBASE* assumes no extension. If the named file cannot be found, an exception occurs.

<new name expC> Identifies the target file that will be created or overwritten by `copy()`. `<new name expC>` may be a filename skeleton with wildcard characters. In that case, *Visual dBASE* displays a dialog box in which you specify the name of the target file and its directory.

Property of File

Description `copy()` lets you duplicate an existing file at the operating system level. `copy()` duplicates a single file of any type.

When running a *Visual dBASE* .EXE, `copy()` first looks for `<filename expC>` in the internal file system of the .EXE file. Any path in `<filename expC>` is ignored. If the named file is found in the .EXE, that file is copied. If the file is not found, then *Visual dBASE* searches for the file on disk. This lets you package static files, like empty tables, inside the .EXE during the build process and extract them when needed. You cannot copy files into the .EXE

If SET SAFETY is ON and a file exists with the same name as the target file, *Visual dBASE* displays a dialog box asking if you want to overwrite the file. If SET SAFETY is OFF, any existing file with the same name is overwritten without warning.

copy() does not automatically copy the auxiliary files associated with table files, such as indexes and memo files. For example, it does not copy the MDX or DBT file associated with a DBF file. When copying tables, use the Database object's *copyTable()* method.

You cannot *copy()* a file that has been opened for writing with the *open()* or *create()* methods; it must be closed first.

Example The following example makes a copy of a file in the current directory:

```
new File().copy( "AFILE", "ACOPY" )
```

Visual dBASE also offers the same functionality in the COPY FILE command. To perform the same operation as above, you could enter

```
copy file AFILE to ACOPY
```

See also *copyTable()*, *rename()*

copy() is also a method of the UpdateSet class (page 14-61).

COPY FILE

Duplicates a specified file.

Syntax COPY FILE <filename> TO <new name>

Description COPY FILE is identical to the File object's *copy()* method, except that as a command, the file name arguments are treated as names, not character expressions. They do not require quotes unless they contain spaces or other special characters. If the name is in a variable, you must use the indirection or macro operators.

See *copy()* for details on the operation of the command.

Example See *copy()*

See Also *copy()*, DELETE FILE, RENAME

create()

Creates and opens a specified file.

Syntax <oRef>.create(<filename expC>[, <access expC>])

<oRef> A reference to a File object.

<filename expC> The name of the file to create and open. By default, *create()* creates the file in the current directory. To create the file in another directory, specify a full path name for <filename expC>.

<access expC> The access level of the file to create, as shown in the following table. The access level string is not case-sensitive, and the characters in the string may be in any order. If omitted, the default is *read* and *write* text file. *Append* is a more restrictive version of *write*; the data is always added to the end of the file.

| <access expC> | Access level |
|---------------|-----------------------------|
| "R" | Read-only text file |
| "W" | Write-only text file |
| "A" | Append-only text file |
| "RW" | Read and write text file |
| "RA" | Read and append text file |
| "RB" | Read-only binary file |
| "WB" | Write-only binary file |
| "AB" | Append-only binary file |
| "RWB" | Read and write binary file |
| "RAB" | Read and append binary file |

Property of File

Description Use *create()* to create a file with a name you specify, assign the file the level of access you specify, and open the file. If *Visual dBASE* can't create the file (for example, if the file is already open), an exception occurs.

SET SAFETY has no effect on *create()*. If <filename expC> already exists, it is overwritten without warning. To see if a file with the same name already exists, use *exists()* before issuing *create()*.

To use other File methods, such as *read()* and *write()*, first open a file with *create()* or *open()*.

When you open a file with *create()*, the file is empty, so the file pointer is positioned at the first character in the file. Use *seek()* to position the file pointer before reading from or writing to a file.

Example The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the *#define* preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"

f = new File()
if f.exists( LOG_FILE )
    f.open( LOG_FILE, "A" )
else
    f.create( LOG_FILE, "A" )
endif
```

```
f.puts( new Date().toLocaleString() )
f.close()
```

See also `close()`, `error()`, `exists()`, `gets()`, `open()`, `puts()`, `read()`, `seek()`, `write()`

createDate()

Returns the date a file was created.

Syntax `<oRef>.createDate(<filename expC>)`

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description `createDate()` checks the file specified by `<filename expC>` and returns the date that the file was created.

To get the date the file was last modified, use `date()`. For the date the file was last accessed, use `accessDate()`. To get the time the file was created, use `createTime()`.

Example The following example uses `createDate()` to display the date the Help file was created:

```
? new File().createDate( "C:\Program Files\dBASE\Visual dBASE 75\Help\VDB.HLP" )
```

See also `accessDate()`, `createTime()`, `date()`, `time()`

createTime()

Returns the time a file was created.

Syntax `<oRef>.createTime(<filename expC>)`

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH,

date ()

if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description *createTime()* checks the file specified by <filename expC> and returns the time, as a character string, that the file was created.

To get the date the file was created, use *createDate()*.

Example The following example uses *createTime()* to display the time the Help file was created:

```
? new File().createTime( "C:\Program Files\dBASE\Visual dBASE 75\Help\VDB.HLP" )
```

See also *createDate()*, *time()*

date()

Returns the date stamp for a file, the date the file was last modified.

Syntax <oRef>.date(<filename expC>)

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description Use *date()* to determine the date on which the last change was made to a file on disk.

When you update a file, *Visual* dBASE changes the file's date stamp to the current operating system date when the file is written to disk. For example, when the user edits a DB table, *Visual* dBASE changes the date stamp on the table file when the file is closed. *date()* reads the date stamp and returns its current value.

To get the date the file was created, use *createDate()*. For the date the file was last accessed, use *accessDate()*. To get the time the file was last changed, use *time()*.

Example The following example uses *date()* to display the date *Visual* dBASE's INI file was last modified:


```
? new File().date( "C:\Program Files\dBASE\Visual dBASE 75\Bin\VDB.INI" )
```

See also `accessDate()`, `createDate()`, `size()`, `time()`

delete()

Removes a file from a disk, optionally sending it to the Recycle Bin.

Syntax `<oRef>.delete(<filename expC> [, <recycle expL>])`

<oRef> A reference to a File object.

<filename expC> Identifies the file to remove. *<filename expC>* may be a filename skeleton with wildcard characters. In that case, *Visual dBASE* displays a dialog box in which you select the file to duplicate.

If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual dBASE* assumes no extension. If the named file cannot be found, an exception occurs.

<recycle expL> Whether to send the file to the Recycle Bin instead of deleting it. If omitted, the file is deleted.

Property of File

Description `delete()` deletes a file from a disk, or sends it to the Recycle Bin.

If *<recycle expL>* is *true*, then SET SAFETY determines whether a dialog appears to confirm sending the file to the Recycle Bin. If *<recycle expL>* is *false* or omitted, SET SAFETY has no effect on `delete()`; the file is deleted without warning.

`delete()` does not automatically remove the auxiliary files associated with table files, such as indexes and memo files. For example, it does not delete the MDX or DBT files associated with a DBF file. When deleting tables, use the Database object's `dropTable()` method.

You cannot `delete()` a file that is open, including one opened with the `open()` or `create()` methods; it must be closed first.

Example The following examples deletes a file in the current directory:

```
new File().delete( "AFILE" )
```

See also `dropTable()`, `rename()`

`delete()` is also a method of the Array (page 10-10), Rowset (page 14-65), and UpdateSet (page 14-65) classes.

DELETE FILE

Removes a file from a disk.

Syntax DELETE FILE <filename>

Description DELETE FILE is similar to the File object's *delete()* method, except that as a command, the file name argument is treated as a name, not a character expression. It does not require quotes unless it contain spaces or other special characters. If the name is in a variable, you must use the indirection or macro operators. Also, DELETE FILE does not support sending a file to the Recycle Bin.

See *delete()* for details on the operation of the command.

The ERASE command is identical to DELETE FILE.

Example Compare this example with the equivalent example for *delete()*:

```
delete file AFILE
```

See Also COPY FILE, *delete()*, ERASE, RENAME

DIR

Performs a directory or table listing.

Syntax DIR | DIRECTORY
[[LIKE] <drive/path/filename skeleton>]

[LIKE] <drive/path/filename skeleton> Specifies a path and/or file specification to be used by DIR. The LIKE keyword is included for readability only; it has no effect on the command.

If omitted, the tables in the current directory or database are listed.

Description DIR (or DIRECTORY) is a utility command that lets you perform a directory listing. The information provided on each file includes its short (8.3) name, its size in bytes, the date of its last update, and its long file name. DIR also shows the total number of bytes used by the listed files, the number of bytes left on that drive, and the total disk space.

DIR with no arguments displays information on the tables in the current directory or database. When accessing tables in the current directory, SET DBTYPE controls the files that are displayed. If SET DBTYPE is DBASE, files with .DBF extensions in the current directory are shown; if SET DBTYPE is PARADOX, .DB files are shown instead. In addition to the information normally displayed, DIR displays the number of records in each table.

The same DBF or DB tables are listed if the database chosen by SET DATABASE is a Standard table alias (one that looks at DBF and DB tables in a specific directory). If the database chosen by SET DATABASE is any other kind of alias, only the table names and the total number of tables are shown.

DIR pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information.

If you have not used ON KEY or SET FUNCTION to reassign the *F4* key, pressing *F4* is a quick way to execute DIR.

Example The following examples use DIR:

```
set database to           // Access tables by directory, not database
set dbtype to dbase
dir                       // Displays all DBF tables in current directory
set dbtype to paradox
dir                       // Displays all DB tables in current directory
open database iblocal    // Open Interbase database
set database to iblocal  // Set active database
dir                       // Displays all tables in database
dir *.DBF                // Displays all DBF files, without # of records
dir c:\autoexec.*        // Displays all AUTOEXEC files in root directory of C:
```

See Also DISPLAY FILES, LIST FILES, ON KEY, SET DATABASE, SET DBTYPE, SET FUNCTION

DISKSPACE()

Returns the number of bytes available on the current or specified drive's disk.

Syntax DISKSPACE([<drive expN>])

<drive expN> A drive number from 1 to 26. For example, the numbers 1 and 2 correspond to drives A and B, respectively.

Without <drive expN> or if <drive expN> is 0, DISKSPACE() returns the number of bytes available on the current drive.

If <drive expN> is less than 0 or greater than 26, DISKSPACE() returns the number of bytes available on the drive that contains the home directory.

Description Use DISKSPACE() to determine how much space is left on a disk.

See Also HOME()

DISPLAY FILES

Displays information about files on disk in the results pane of the Command window.

Syntax DISPLAY FILES
 [[LIKE] <drive/path/filename skeleton>]
 [TO FILE <filename> | ? | <filename skeleton>]
 [TO PRINTER]

TO FILE <filename> | ? | <filename skeleton> Directs output to a text file as well as to the results pane of the Command window. By default, *Visual dBASE* assigns a .TXT extension. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer as well as to the results pane of the Command window.

Description DISPLAY FILES is identical to DIR, and adds the option of directing the output to a file or a printer (or both) in addition to the Command window. See DIR for details.

DISPLAY FILES is the same as LIST FILES, except that LIST FILES doesn't pause for each screenful of information but rather lists the information continuously. This makes LIST FILES more appropriate when directing output to a file or printer.

See Also DIR, LIST FILES

DOS

Open an MS-DOS or Windows NT command prompt.

Syntax DOS

Description Use the DOS command to open an operating system command prompt. This has the same effect as choosing MS-DOS Prompt or Command Prompt from the Windows Start menu. The command prompt runs as a separate process.

To execute single operating system commands use RUN. To execute applications, use RUN().

See Also RUN, RUN()

eof()

Returns *true* if the file pointer is at the end of a file previously opened with *create()* or *open()*

Syntax <oRef>.eof()

<oRef> A reference to the File object that created or opened the file.

Property of File

Description *eof()* determines if the file pointer of the file you specify is at the end of the file (EOF), and returns *true* if it is and *false* if it is not. The file pointer is considered to be at EOF if it is positioned at the byte after the last character in the file.

You can move the file pointer to the end of the file with *seek()*. If a file is empty, as it is when you first create a new file with *create()*, *eof()* returns *true*.

Example Suppose you have a data file generated by a mainframe computer that has fixed-length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"

fIn  = new File()
fOut = new File()

fIn.open( IN_FILE );
fOut.create( OUT_FILE );

do while not fIn.eof()
    fOut.puts( fIn.read( REC_LENGTH ) ) // Read fixed length; write with line break
enddo

fIn.close()
fOut.close()
```

See also *position*, *seek()*

ERASE

Removes a file from a disk.

Syntax ERASE <filename>

Description ERASE is similar to the File object's *delete()* method, except that as a command, the file name argument is treated as a name, not a character expression. It does not require quotes unless it contains spaces or other special characters. If the name is in a variable, you must use the indirection or macro operators. Also, ERASE does not support sending a file to the Recycle Bin.

See *delete()* for details on the operation of the command.

The DELETE FILE command is identical to ERASE.

Example Compare this example with the equivalent example for *delete()*:

```
erase AFILE
```

This example lets the user pick a text file to delete:

`error ()`

```
cFile = getfile( "*.txt", "Delete text file from current directory" )
if upper( cFile ) = set( "DIRECTORY" ) and right( upper( cFile ), 4 ) == ".TXT"
    erase ( cFile )
else
    msgbox( "Not a text file in the current directory", "Can't delete", 48 )
endif
```

The beginning of the returned file name is compared with the current directory returned by SET("DIRECTORY") using the equals operator (with SET EXACT OFF). The end of the file name is checked to see if it is a text file.

If the file is a text file in the current directory, the indirection operators convert the file name stored into a name the command can use. Without the indirection operators (or the macro operator, which would have the same effect), the command would attempt to erase the file named "cFile".

See Also COPY FILE, *delete()*, DELETE FILE, RENAME

error()

Returns the error number of the most recent byte-level input or output error, or 0 if the most recent byte-level method was successful.

Syntax `<oRef>.error()`

<oRef> A reference to the File object that attempted the operation.

Property of File

Description To trap errors, call the File object method in a TRY block. Use the number that *error()* returns in a CATCH block to respond to errors in the byte-level methods of the File object. The following table lists the byte-level method errors that *error()* returns.

| Error number | Cause of error |
|--------------|--------------------------------------|
| 2 | File or directory not found |
| 3 | Bad path name |
| 4 | No more file handles available |
| 5 | Can't access file |
| 6 | Bad file handle |
| 8 | No more directory entries available |
| 9 | Error trying to set the file pointer |
| 13 | No more disk space |
| 14 | End of file |

See also *close()*, *create()*, *eof()*, *flush()*, *gets()*, *open()*, *puts()*, *read()*, *seek()*, *write()*

exists()

Tests for the existence of a file. Returns *true* if the file exists and *false* if it doesn't.

Syntax <oRef>.exists(<filename expC>)

<oRef> A reference to a File object.

<filename expC> The name of the file to search for. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension.

Property of File

Description Use *exists()* to determine whether a file exists. You can use either the long file name or the short file name.

Example The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the #define preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"

f = new File()
if f.exists( LOG_FILE )
    f.open( LOG_FILE, "A" )
else
    f.create( LOG_FILE, "A" )
endif
f.puts( new Date().toLocaleString() )
f.close()
```

See also *create()*, *error()*, *GETFILE()*, *open()*, *PUTFILE()*, *shortName()*

FILE()

Tests for the existence of a file. Returns *true* if the file exists and *false* if it doesn't.

Syntax FILE(<filename expC>)

Description FILE() is identical to the File object's *exists()* method, except that as a built-in function, it does not require a File object to work.

`flush ()`

Example Compare this example with the equivalent example for *exists()*:

```
#define LOG_FILE "ACCESS.TXT"

if file( LOG_FILE )
    h = fopen( LOG_FILE, "A" )
else
    h = fcreate( LOG_FILE, "A" )
endif
fputs( h, new Date().toLocaleString() )
fclose( h )
```

See Also `FCREATE()`, `FERROR()`, `FOPEN()`, `FSHORTNAME()`, `GETFILE()`, `PUTFILE()`

flush()

Writes to disk a file previously opened with *create()* or *open()* without closing the file. Returns *true* if successful and *false* if unsuccessful.

Syntax `<oRef>.flush()`
<oRef> A reference to the File object that created or opened the file.

Property of File

Description Use *flush()* to save a file in the file buffer to disk, flush the file buffer, and keep the file open. If *flush()* is successful, it returns *true*.

Flushing a buffer to disk is similar to saving the file and continuing to work on it. Until you flush an open file buffer to disk, any data in the buffer is stored only in RAM (random-access memory). If the power to the computer fails or *Visual dBASE* ends abnormally, data in RAM is lost. However, if you have used *flush()* to write the file buffer to disk, you lose only data that was added between the time you issued *flush()* and the time the system failed.

To save the file to disk and close the file, use *close()*.

See also *close()*
flush() is also a method of the Rowset class (page 14-73)

FNAMEMAX()

Returns the maximum allowable file-name length on a given drive or volume.

Syntax `FNAMEMAX([<expC>])`

<expC> The drive letter (with a colon), or name of the volume, to check. If <expC> is not provided, the current drive/volume is assumed. If the drive/volume does not exist, an error occurs.

Description FNAMEMAX() checks the drive or volume specified by <expC> and returns the maximum file-name length (including the dot and three-letter extension) allowed for files on that drive/volume. Typical values are:

| FNAMEMAX() | Drive type |
|-------------|-------------------------------|
| 255 | Windows long file name |
| 12 | MS-DOS-compatible 8.3 name |
| 240 | Novell Netware long file name |

See Also FSHORTNAME()

F UNIQUE ()

Creates a unique file name.

Syntax F UNIQUE([<expC>])

<expC> A file-name skeleton, using ? as the wildcard character (the * character is not allowed).

Description Use F UNIQUE() when creating temporary files to generate a file name that is not being used by an existing file. The generated file name follows the file name skeleton you specify, with random numbers substituted for each ? character.

F UNIQUE() generates the new file name by replacing each wildcard character with a random number, then looking in the current or specified directory for a file name that matches the new file name. If no match is found, F UNIQUE() returns that name—but it does not create the file. If a match is found, F UNIQUE() tries again until a unique file name is found. If no combination of random numbers is successful, F UNIQUE() returns an empty string.

If you omit <expC>, F UNIQUE() returns an 8-character file name with no extension, composed entirely of random numbers, in the Windows temp directory.

Example The following example shows the top-level routine used to process a generated text file. An intermediate file is created during the process. The final result is stored in a subdirectory. Because the application is used by many people on a network, a fixed file name cannot be used. Instead it uses a temporary file whose name is generated by F UNIQUE().

```
parameter cFile           // Name of file to process
local cTmpFile
cTmpFile = funique( "T???????.TMP" ) // Letter T followed by seven digits
preProcess( cFile, cTmpFile )        // Create intermediate temp file
```

GETDIRECTORY()

```
mainProcess( cTmpFile, cFile )      // Create result file in subdirectory
erase ( cTmpFile )                  // Erase temp file when done
```

See Also *exists()*, *FILE()*

GETDIRECTORY()

Displays a dialog box from which you can select a directory for use with subsequent commands.

Syntax GETDIRECTORY([<directory expC>])

<directory expC> The initial directory to appear in the dialog box. If <directory expC> is omitted, the current directory appears as the initial directory.

Description Use GETDIRECTORY() to return a directory name for use in subsequent commands.

GETDIRECTORY() does not return a final backslash at the end of the directory name it returns. GETDIRECTORY() returns an empty string if the user clicks Cancel or presses *Esc*.

See Also CD, GETFILE(), SET DIRECTORY

GETENV()

Returns the value of an operating system environment variable.

Syntax GETENV(<expC>)

<expC> The name of the environment variable to return.

Description Use GETENV() to return the current value of an operating system environment variable.

If *Visual* dBASE can't find the environment variable specified by <expC>, it returns an empty string.

See Also OS()

GETFILE()

Displays a dialog box, from which the user can choose or enter an existing file name, and returns the file name.

Syntax GETFILE([<filename skeleton expC>
[, <title expC>
[, <suppress database expl>]]])

<filename skeleton expC> A character string that matches selected file names with the wildcard characters ? and *. The GETFILE() dialog box initially lists only those file names in the current directory that match the file name skeleton. Without <filename skeleton expC>, the dialog box lists all file names.

<title expC> A title displayed in the top of the dialog box. Without <title expC>, the GETFILE() dialog box displays the default title. If you want to specify a value for <title expC>, you must also specify a value or empty string ("") for <filename skeleton expC>.

<suppress database expl> Whether to suppress the combobox from which you can choose a database. The default is *true*; the Database combobox is not displayed. If you want to specify a value for <suppress database expl>, you must also specify a value or empty string ("") for <filename skeleton expC> and <title expC>.

Description Use GETFILE() to present the user with a dialog box from which they can choose an existing file or table. GETFILE() does not open any files.

The GETFILE() dialog box includes names of files whether they are currently open or closed. *Visual* dBASE returns the full path name of the file whether SET FULLPATH is ON or OFF.

By default, the dialog box opened with GETFILE() displays file names from the current directory the first time you issue GETFILE(). After the first time you use GETFILE() and exit successfully, the subdirectory you choose becomes the default the next time you use GETFILE().

If <suppress database expl> is *false*, you can also choose from a list of databases. When a database is selected, the dialog box displays a list of tables in that database instead of files in the current directory.

The dialog box is a standard Windows dialog box. The user can perform many Windows Explorer-like activities in this dialog box, including renaming files, deleting files, and creating new folders. They can also right-click on a file to get its context menu. These features are disabled when the dialog is displaying tables in a database instead of files in a directory.

GETFILE() returns an empty string if the user chooses the Cancel button or presses *Esc*.

See Also FILE(), GETDIRECTORY(), PUTFILE()

gets()

Returns a line of text from a file previously opened with *create*() or *open*().

Syntax <oRef>.gets([<characters expN> [, <end-of-line expC>]])

<oRef> A reference to the File object that created or opened the file.

<characters expN> The number of characters to read and return before a carriage return is reached.

<end-of-line expC> The end-of-line indicator, which can be a string of one or two characters. If omitted, the default is a hard carriage return and line feed. The following table lists standard codes used as end-of-line indicators.

| Character code (decimal) | (hexadecimal) | Represents |
|--------------------------|---------------|-------------------------------|
| CHR(141) | 0x8D | Soft carriage return (U.S.) |
| CHR(255) | 0xFF | Soft carriage return (Europe) |
| CHR(138) | 0x8A | Soft linefeed (U.S.) |
| CHR(0) | 0x00 | Soft linefeed (Europe) |
| CHR(13) | 0x0D | Hard carriage return |
| CHR(10) | 0x0A | Hard linefeed |

Use the CHR() function to create the <end-of-line expC> if needed. To designate the <end-of-line expC>, you must also specify the <characters expN>. If you don't want a line length limit, use an arbitrarily high number. For example:

```
cLine = f.gets( 10000, chr( 0x8d ) ) // Soft carriage return (U.S.)
```

Property of File

Description Use gets() to read lines from a text file. gets() reads and returns a character string from the file opened by the File object, starting at the file pointer position, and reading past but not returning the first end-of-line character(s) it encounters.

gets() will read characters until it encounters the end-of-line character(s) or it reads the number of characters you specify with <characters expN>, whichever comes first. If a file does not have end-of-line character(s) and you do not specify <characters expN>, gets() will read and return everything from the current file pointer position to the end of the file.

If the file pointer position is at an end-of-line character(s), gets() returns an empty string (""); the line is empty.

If gets() encounters an end-of-line character(s), it positions the file pointer at the character after the end-of-line character(s); that is, at the beginning of the next line. Otherwise, gets() positions the file pointer at the character after the last character it returns. Use seek() to move the file pointer before or after using gets().

If the file being read is not a text file, use read() instead. read() requires <characters expN> to be specified, and does not treat end-of-line characters specially.

To write a text file, use puts(). readln() is identical to gets().

Example The following statements display the contents of a text file in an Text component, replacing the line breaks in the text file with the HTML
 tag. The name of the file is typed into a Entryfield component named *entryfield1*, and the Text component is named *text1*.

```
f = new File() // Create File object
if f.exists( form.entryfield1.value ) // Make sure file exists
    f.open( form.entryfield1.value )
    form.text1.text = "" // Clear HTML component
    do while not f.eof()
        form.text1.text += f.gets() + "<BR>" // Write lines to HTML component
    enddo
    f.close() // Close file
else
    form.text1.text = form.entryfield1.value + " not found"
endif
```

See Also *create()*, *eof()*, *error()*, *open()*, *puts()*, *read()*, *seek()*

handle

The operating system file handle for a file previously opened with *create()* or *open()*.

Property of File

Description When a file is opened by the operating system, it is assigned a *file handle*, an arbitrary number that identifies that open file. Applications then use that file handle to refer to that file.

A File object's *handle* property reflects the file handle used by *Visual dBASE* to access a file opened with *create()* or *open()*. It is a read-only property and is generally informational only. By calling methods of the File object, *Visual dBASE* internally uses the file handle to perform its operations.

See also *path*

handle is also a property of many data access classes (page 14-76).

HOME()

Returns the directory where the VDB.EXE in use is located.

Syntax HOME()

Description There are two "home" directories:

- The directory where *Visual dBASE* is installed, by default:

```
C:\Program Files\dBASE\Visual dBASE 75\
```

- The directory where the actual executable file, VDB.EXE is installed. This is in the \Bin subdirectory of the installation directory, so by default, it's:

C:\Program Files\Visual dBASE\Visual dBASE 75\Bin\

HOME() identifies the directory in which the currently running copy of VDB.EXE is located. HOME() returns the full path name whether SET FULLPATH is ON or OFF, and always includes the trailing backslash, as shown.

To identify the *Visual* dBASE installation home directory, use _dbwinhome.

See Also CD, SET DIRECTORY, _dbwinhome

LIST FILES

Displays information about files on disk in the results pane of the Command window.

Syntax LIST FILES
 [[LIKE] <drive/path/filename skeleton>]
 [TO FILE <filename> | ? | <filename skeleton>]
 [TO PRINTER]

TO FILE <filename> | ? | <filename skeleton> Directs output to a text file as well as to the results pane of the Command window. By default, *Visual* dBASE assigns a .TXT extension. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer as well as to the results pane of the Command window.

Description LIST FILES is the same as DISPLAY FILES, except that LIST FILES doesn't pause for each screenful of information but rather lists the information continuously. This makes LIST FILES more appropriate when directing output to a file or printer.

See Also DIR, DISPLAY FILES

MD

Creates a new directory.

Syntax MD <directory>

Description MD is identical to MKDIR. See MKDIR for details.

See Also CD, MKDIR, SET DIRECTORY

MKDIR

Creates a new directory.

Syntax MKDIR <directory>

<directory> The directory you want to create.

Description Use MKDIR to create a new directory. The MD command is identical to MKDIR.

The new directory name must follow the standard naming conventions for the operating system.

If you try to make a directory that already exists or is on a path that does not exist, an error occurs.

After you create the new directory, you can use CD or SET DIRECTORY to make the new directory the current directory.

See Also CD, SET DIRECTORY

open()

Opens a specified file.

Syntax <oRef>.open(<filename expC>[, <access expC>])

<oRef> A reference to a File object.

<filename expC> The name of the file to open. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

<access expC> The access level of the file being opened, as shown in the following table. The access level string is not case-sensitive, and the characters in the string may be in any order. If omitted, the default is *read-only* text file. *Append* is a more restrictive version of *write*; the data is always added to the end of the file

| <access expC> | Access level |
|---------------|--------------------------|
| "R" | Read-only text file |
| "W" | Write-only text file |
| "A" | Append-only text file |
| "RW" | Read and write text file |

| <access expC> | Access level |
|---------------|-----------------------------|
| "RA" | Read and append text file |
| "RB" | Read-only binary file |
| "WB" | Write-only binary file |
| "AB" | Append-only binary file |
| "RWB" | Read and write binary file |
| "RAB" | Read and append binary file |

Property of File

Description Use *open()* to open a file with a name you specify and assign the file the level of access you specify. If *Visual* dBASE can't open the file (for example, if the file is already open), an exception occurs.

To use other File methods, such as *read()* and *write()*, first open a file with *open()* or *create()*.

If you open the file with append-only or read and append access, the file pointer is positioned at the end-of-file, after the last character. For other access levels, the file pointer is positioned at the first character in the file. Use *seek()* to position the file pointer before reading from or writing to a file.

Example The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of the week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the *#define* preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"

f = new File()
if f.exists( LOG_FILE )
    f.open( LOG_FILE, "A" )
else
    f.create( LOG_FILE, "A" )
endif
f.puts( new Date().toLocaleString() )
f.close()
```

See also *close()*, *create()*, *error()*

open() is also a method of the Form class (page 15-143).

OS()

Returns the name and version number of the current operating system.

Syntax OS()

Description Use `OS()` to determine the version of Windows in which *Visual* dBASE is running. To determine which version of *Visual* dBASE is running, use `VERSION()`. `OS()` returns a character string like:

Windows NT version 4.00

with the name of the operating system, the word “version” and the version number.

See Also `VERSION()`

path

The full path and file name for a file previously opened with `create()` or `open()`.

Property of File

Description When you open a file with `create()` or `open()`, the path is optional. If you use `create()` without a path, the file is created in the current directory. If you use `open()` without a path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with `SET PATH`, if any.

A File object's *path* property reflects the full path and file name for the open file. It is a read-only property.

See also *handle*

position

The position of the file pointer in a file previously opened with `create()` or `open()`.

Property of File

Description A File object's *position* property reflects the current position of the file pointer. It is a read-only property. To move the file pointer, use `seek()`. Reading and writing to a file also moves the file pointer.

The position is zero-based. The first character in the file is at position zero.

See also `seek()`

PUTFILE()

Displays a dialog box within which the user can choose an existing file to overwrite or a new file name, and returns the file name.

PUTFILE()

Syntax PUTFILE([<*title expC*>
[, <*filename expC*>
[, <*extension expC*>
[, <*suppress database expL*>]]])

<*title expC*> A title that is displayed at the top of the dialog box.

<*filename expC*> The default file name that is displayed in the dialog box's entryfield. Without <*filename expC*>, PUTFILE() displays an empty entryfield.

<*extension expC*> A default extension for the file name that PUTFILE() returns.

<*suppress database expL*> Whether to suppress the combobox from which you can choose a database. The default is *true*; the Database combobox is not displayed. If you want to specify a value for <*suppress database expL*>, you must also specify a value or empty string ("") for <*filename expC*>, <*title expC*>, and <*extension expC*>.

Description Use PUTFILE() to present the user with a dialog box from which they can choose an existing file or table or specify a new file or table name. If they choose an existing file, the user then gets the standard "Replace existing file?" dialog box. If they choose no, their choice is ignored and they are left in the PUTFILE() dialog box. PUTFILE() does not actually create or write anything to the specified file.

The PUTFILE() dialog box includes names of files whether they are currently open or closed. *Visual dBASE* returns the full path name of the file whether SET FULLPATH is ON or OFF.

By default, the dialog box opened with PUTFILE() displays file names from the current directory the first time you issue PUTFILE(). After the first time you use PUTFILE() and exit successfully, the subdirectory you choose becomes the default the next time you use PUTFILE().

If <*suppress database expL*> is *false*, you can also choose from a list of databases. When a database is selected, the dialog box displays a list of tables in that database instead of files in the current directory.

The dialog box is a standard Windows dialog box. The user can perform many Windows Explorer-like activities in this dialog box, including renaming files, deleting files, and creating new folders. They can also right-click on a file to get its context menu. These features are disabled when the dialog is displaying tables in a database instead of files in a directory.

PUTFILE() returns an empty string if the user chooses the Cancel button or presses *Esc*.

See Also FILE(), GETFILE()

puts()

Writes a character string and one or two end-of-line characters to a file previously opened with *create()* or *open()*. Returns the number of characters written.

Syntax <oRef>.puts(<string expC> [, <characters expN> [, <end-of-line expC>]])

<oRef> A reference to the File object that created or opened the file.

<string expC> The character expression to write to the specified file. If you want to write only a portion of <string expC> to the file, use the <characters expN> argument.

<characters expN> The number of characters of the specified character expression <string expC> to write to the specified file, starting at the first character in <string expC>. If omitted, the entire string is written.

<end-of-line expC> The end-of-line indicator, which can be a string of one or two characters. If omitted, the default is a hard carriage return and line feed. The following table lists standard codes used as end-of-line indicators.

| Character code (decimal) | (hexadecimal) | Represents |
|--------------------------|---------------|-------------------------------|
| CHR(141) | 0x8D | Soft carriage return (U.S.) |
| CHR(255) | 0xFF | Soft carriage return (Europe) |
| CHR(138) | 0x8A | Soft linefeed (U.S.) |
| CHR(0) | 0x00 | Soft linefeed (Europe) |
| CHR(13) | 0x0D | Hard carriage return |
| CHR(10) | 0x0A | Hard linefeed |

Use the CHR() function to create the <end-of-line expC> if needed. To designate the <end-of-line expC>, you must also specify the <characters expN>. If you don't want a line length limit, use an arbitrarily high number. For example:

```
f.puts( cLine, 10000, chr( 0x8d ) )    // Soft carriage return (U.S.)
```

Property of File

Description Use *puts()* to write text files. *puts()* writes a character string and one or two end-of-line characters to a file. If the file was opened in append-only or read and append mode, the string is always added to the end of the file. Otherwise, the string is written starting at the current file pointer position, overwriting any existing characters. You must have either write or append access to use *puts()*.

puts() returns the number of bytes written to the file, including the end-of-line character(s). If *puts()* returns 0, no characters were written. Either <string expC> is an empty string, or the write was unsuccessful.

Use *error()* to determine if an error occurred.

When *puts()* finishes executing, the file pointer is located at the character immediately after the last character written, which is the end-of-line character.

`read ()`

Successive *puts()* calls writes one line after another. Use *seek()* to move the file pointer before or after you use *puts()*.

To write to a file that is not a text file, use *write()*. *write()* does not add the end-of-line character(s). To read from a text file, use *gets()*. *writeln()* is identical to *puts()*.

Example The following example writes the current date and time to a text file, which you might do for a simple access log. The file is archived and deleted at the end of week, so you need to test for its existence to determine whether it should be created or opened. The name of the file, which is used in three different places, is set in a manifest constant created by the *#define* preprocessor directive for ease of maintenance.

```
#define LOG_FILE "ACCESS.TXT"

f = new File()
if f.exists( LOG_FILE )
    f.open( LOG_FILE, "A" )
else
    f.create( LOG_FILE, "A" )
endif
f.puts( new Date().toLocaleString() )
f.close()
```

See Also *create()*, *eof()*, *error()*, *gets()*, *open()*, *seek()*, *write()*

read()

Returns a specified number of characters from a file previously opened with *create()* or *open()*.

Syntax `<oRef>.read(<characters expN>)`

<oRef> A reference to the File object that created or opened the file.

<characters expN> The number of characters to return from the specified file.

Property of File

Description *read()* returns the number of characters you specify from the file opened by the File object. *read()* starts reading characters from the current file pointer position, leaving the file pointer at the character immediately after the last character read. Use *seek()* to move the file pointer before or after you use *read()*.

If the file to be read is a text file, use *gets()* instead. *gets()* looks for end-of-line characters, and returns the contents of the line, without the end-of-line character(s).

To write to a file, use *write()*.

Example Suppose you have a data file generated by a mainframe computer that has fixed-length records with no record breaks. You want to convert this file so that you have one record on each line. Use two File objects to read and write the file, adding line breaks as you write:

```
#define REC_LENGTH 80
#define IN_FILE    "STUFF.REC"
#define OUT_FILE   "STUFF.TXT"

fIn  = new File()
fOut = new File()

fIn.open( IN_FILE );
fOut.create( OUT_FILE );

do while not fIn.eof()
    fOut.puts( fIn.read( REC_LENGTH ) ) // Read fixed length; write with line break
enddo

fIn.close()
fOut.close()
```

See also *create(), eof(), error(), gets(), open(), seek(), write()*

readln()

Returns a line of text from a file previously opened with *create()* or *open()*.

Syntax *<oRef>.readln([<characters expN> [, <end-of-line expC>]])*

Property of File

Description *readln()* is identical to *gets()*. See *gets()* for details.

RENAME

Renames a file on disk.

Syntax RENAME *<filename>* TO *<new name>*

Description RENAME is identical to the File object's *rename()* method, except that as a command, the file name arguments are treated as names, not a character expressions. They do not require quotes unless they contain spaces or other special characters. If the name is in a variable, you must use the indirection or macro operators.

See *rename()* for details on the operation of the command.

Example Compare this example with the equivalent example for *rename()*:

```
rename AFILE to SOMETHING
```

See Also COPY FILE, DELETE FILE, ERASE, *rename()*

rename()

Renames a file on disk.

Syntax <oRef>.rename(<filename expC>, <new name expC>)

<oRef> A reference to a File object.

<filename expC> Identifies the file to rename (also known as the source file). <filename expC> may be a file name skeleton with wildcard characters. In that case, *Visual dBASE* displays a dialog box in which you select the file to rename.

If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual dBASE* assumes no extension. If the named file cannot be found, an exception occurs.

<new name expC> Identifies the new name for the source file (also known as the target file). <new name expC> may be a file name skeleton with wildcard characters. In that case, *Visual dBASE* displays a dialog box in which you specify the name of the target file and its directory.

Property of File

Description *rename()* lets you change the name of a file at the operating system level.

If SET SAFETY is ON and a file exists with the same name as the target file, *Visual dBASE* displays a dialog box asking if you want to overwrite the existing file. If SET SAFETY is OFF and a file exists with the same name as the target file, an exception occurs, and the target file is not overwritten.

If you specify a different drive or directory for the target file, *Visual dBASE* moves the source file to that location.

rename() does not automatically rename the auxiliary files associated with table files, such as indexes and memo files. For example, it does not rename the MDX or DBT files associated with a DBF file. When renaming tables, use the Database object's *renameTable()* method.

Example The following example changes the name of a file in the current directory to something else:

```
new File().rename( "AFILE", "SOMETHING" )
```

See also *copy()*

RUN

Executes a program or operating system command from within *Visual* dBASE.

Syntax RUN <command>

<command> A command recognized by your operating system.

Description Use RUN to execute a single operating system command or program from within *Visual* dBASE. Enter commands and file names exactly as you would when working in the command prompt; do not enclose them in quotes. ! is equivalent to RUN.

RUN opens a command prompt in the current directory and executes <command>. The command prompt automatically closes when the program or command is finished. Commands and programs launched by RUN execute as a separate task, as if you had started that task from the Start menu. *Visual* dBASE continues to run on its own.

To open a command prompt so you can enter multiple commands yourself, use the DOS command. To execute a Windows application, use RUN() instead; it does not open a command prompt window.

Example In the following example, clicking a button on a form runs a command line compression utility through a batch file:

```
function backupButton_onClick
run ZIPEM.BAT
```

See Also DOS, RUN()

RUN()

Executes a program or operating system command from within *Visual* dBASE, returning the instance handle of the program.

Syntax RUN([<direct expl>], <command expC>)

<direct expl> Determines whether RUN() runs a Windows program directly (*true*) or through a command prompt (*false*). If <command expC> is not a Windows program, <direct expl> must be *false*, or RUN() has no effect. If you omit <direct expl>, *Visual* dBASE assumes a value of *false*.

<command expC> A Windows program name or a command recognized by your operating system.

Description Use RUN() to execute another Windows program or an operating system command from within *Visual* dBASE.

To run another Windows program, *<direct expL>* should be *true*; otherwise, a separate command prompt is opened first, and you cannot get the returned instance handle.

See Also DOS, RUN

seek()

Moves the file pointer in a file previously opened with *create()* or *open()*, and returns the new position of the file pointer.

Syntax *<oRef>.seek(<offset expN> [, <position expN>])*

<oRef> A reference to the File object that created or opened the file.

<offset expN> The number of bytes to move the file pointer in the specified file. If *<offset expN>* is negative, the file pointer moves toward the beginning of the file. If *<offset expN>* is 0, the file pointer moves to the position you specify with *<position expN>*. If *<offset expN>* is positive, the file pointer moves toward the end of the file or beyond the end of the file.

<position expN> The number 0, 1, or 2, indicating a position relative to the beginning of the file (0), to the file pointer's current position (1), or to the end of the file (2). The default is 0.

Property of File

Description *seek()* moves the file pointer in the file you specify relative to the position specified by *<position_expN>*, and returns the resulting position of the file pointer as an offset from the beginning of the file. The File object's *position* property is also updated with this new position. If an error occurs, *seek()* returns -1.

The movement of the file pointer is relative to the beginning of the file unless you specify otherwise with *<position_expN>*. For example, *seek(5)* moves the file pointer five characters from the beginning of the file (the 6th character) while *seek(5,1)* moves it five characters forward from its current position. You can move the file pointer beyond the end of the file, but you can't move it before the beginning of the file.

To move the file pointer to the beginning of a file, use *seek(0)*. To move it to the end of a file, use *seek(0, 2)*. To move to the last character in a file, use *seek(-1,2)*.

gets(), *puts()*, *read()*, and *write()* also move the file pointer as they read from or write to the file.

Example Suppose you're exporting data from a table in a special format for another program. The export file must have the number of rows of data written in the file, starting at the 9th character. You extend the File class, adding methods to create the export file, write the data in the special format, and record the

number of rows written. The following is the method that records the number of rows.

```
function recordRowsWritten()
  this.seek( 8 )           // 9th character == offset 8
  this.write( "" + this.rowsExported ) // Convert number to string to write
```

See also *gets(), position, puts(), read(), write()*

SET DIRECTORY

Changes the current drive or directory.

Syntax SET DIRECTORY TO [*<path>*]

<path> The new drive and/or path. Quotes (single or double) are required if the path contains spaces or other special characters; optional otherwise.

Description SET DIRECTORY works like CD, except SET DIRECTORY TO (with no argument) returns you to the HOME() directory, while CD with no argument displays the current directory.

To get the current directory, use SET("DIRECTORY").

See Also CD, HOME(), SET(), VALIDDRIIVE(), _dbwinhome

SET FULLPATH

Specifies whether functions that return file names return the full path with the file name.

Syntax SET FULLPATH on | OFF

Description Use SET FULLPATH ON when you need to have functions or methods such as *shortName()*, return a file name with its full path. When SET FULLPATH is OFF, these functions include the drive letter (and colon) with the file name only.

Some functions, such as GETFILE(), always return the full path, regardless of SET FULLPATH.

See Also GETFILE(), *shortName()*

SET PATH

Specifies the directory search route that *Visual* dBASE follows to find files that are not in the current directory.

shortName ()

Syntax SET PATH TO [*<path list>*]

<path list> A list of (optional) drives and directories indicating the *search path*—one or more drives and directories you want *Visual dBASE* to search for files. Separate each directory path name with commas, semicolons, or spaces. If the path name contains spaces or other special characters, the path name should be enclosed in quotes.

Description Use SET PATH to establish a search path to access files located on directories other than the current directory. When no SET PATH setting exists and you don't provide the full path name when you specify a file name, dBASE searches for that file only in the current directory.

The order in which you list drives and directories with SET PATH TO *<path list>* is the order *Visual dBASE* searches for a file in that search path. Use SET PATH when an application's files are in several directories.

SET PATH TO without the option *<path list>* resets the search path to the default value (no path).

See Also CD, SET DIRECTORY

shortName()

Returns the short (8.3) name of a file.

Syntax *<oRef>.shortName(<filename expC>)*

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual dBASE* assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description *shortName()* checks the file specified by *<filename expC>* and returns a name for the file following the DOS file-naming convention (eight-character file name, three-character extension). If SET FULLPATH is ON, the path is also returned.

See also *exists()*

size()

Returns the size of a file in bytes.

Syntax <oRef>.size(<filename expC>)

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description Use *size()* to determine the size of a file on disk.

With the byte-level access methods of the File object, *Visual* dBASE doesn't update the size on the file recorded on the disk until you *close()* the file.

Example The following example uses *size()* to display the size of the Help file:

```
? new File().size( "C:\Program Files\dBASE\Visual dBASE 75\Help\VDB.HLP" )
```

See also *date()*, *time()*

time()

Returns the time stamp for a file, the time the file was last modified.

Syntax <oRef>.time(<filename expC>)

<oRef> A reference to a File object.

<filename expC> The name of the file to check. Wildcard characters are not allowed; you must specify the actual file name.

If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the search path(s) you specified with SET PATH, if any. If you specify a file without including its extension, *Visual* dBASE assumes no extension. If the named file cannot be found, an exception occurs.

Property of File

Description Use *time()* to determine the time of day when the last change was made to a file on disk. *time()* returns the time as a character string.

When you update a file, *Visual* dBASE changes the file's time stamp to the current operating system time when the file is written to disk. For example, when the user edits a DB table, *Visual* dBASE changes the time stamp on the table file when the file is closed. *time()* reads the time stamp and returns its current value.

To get the time the file was created, use *createTime()*. For the date the file was last modified, use *date()*.

Example The following example uses *time()* to display the time *Visual* dBASE's INI file was last modified:

```
? new File().time( "C:\Program Files\dBASE\Visual dBASE 75\Bin\VDB.INI" )
```

See also *createTime()*, *date()*

TYPE

Display the contents of a text file.

Syntax TYPE <filename 1> | ? | <filename skeleton 1>
[MORE]
[NUMBER]
[TO FILE <filename 2> | ? | <filename skeleton 2>] | [TO PRINTER]

<filename> | ? | <filename skeleton> The file whose contents to display. TYPE ? and TYPE <filename skeleton> display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. You must specify a file-name extension.

MORE Pauses output when it fills the Command window; otherwise, the output scrolls through the Command window to the end of the file.

NUMBER Precedes each line of output with its line number.

TO FILE <filename 2> | ? | <filename skeleton> Directs output to the text file <filename 2>, as well as to the results pane of the Command window. By default, *Visual* dBASE assigns a .TXT extension to <filename 2> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer, as well as to the results pane of the Command window.

Description Use TYPE to display the contents of text files. All program files in *Visual* dBASE are text files that you can display with TYPE.

If you TYPE a file TO FILE or TO PRINTER, *Visual* dBASE adds two lines of output at the beginning of the saved or printed output if SET HEADINGS is ON. The first line is a blank line, and the second line contains the full path name and date stamp of the source file. If you specify NUMBER, these two lines are not numbered; numbering begins with 1 at the first actual line of the source file.

If you specify MORE and cancel output before completion, *** INTERRUPTED *** appears in the results pane of the Command window, but does not appear in the incomplete saved or printed output.

If SET SAFETY is ON and a file exists with the same name as the target file, dBASE displays a dialog box asking if you want to overwrite the file. If SET SAFETY is OFF, any existing file with the same name is overwritten without warning.

See Also EJECT, SET ALTERNATE, SET HEADINGS, SET PRINTER, SET SAFETY

VALIDDRIVE()

Returns *true* if the specified drive exists and can be read. Returns *false* if the specified drive does not exist or cannot be read.

Syntax VALIDDRIVE(<drive expC>)

<drive expC> The drive to be tested, which can be either:

- A drive letter, optionally followed by a colon, or
- The UNC name for a drive

Description Use VALIDDRIVE() to determine if a specified drive exists and is ready before using CD, SET DEFAULT, SET DIRECTORY or SET PATH. VALIDDRIVE() is also useful if your program copies files to or from a drive, or includes drive letters in any file names.

VALIDDRIVE() can verify any drive specified, including drives created by partitioning a hard disk and mapped network drives. Checking for a floppy disk or network drive takes a few seconds, so you should display a message before you check.

Example The following example checks if a disk is inserted in drive A:

```
if not validdrive( "a:" )
  // No disk (or no floppy drive installed)
endif
```

The following example use a UNC name to check if the user is connected to a particular network drive:

```
if validdrive( "\\finance\vol2" )
  // Not connected to Finance server, or has no access to Vol2 volume
endif
```

See Also CD, SET DIRECTORY, SET PATH

write()

Writes a character string to a file previously opened with *create()* or *open()*. Returns the number of characters written.

Syntax <oRef>.write(<expC> [, <characters expN>])

<oRef> A reference to the File object that created or opened the file.

<expC> The character expression to write to the specified file. If you want to write only a portion of *<string expC>* to the file, use the *<characters expN>* argument.

<characters expN> The number of characters of the specified character expression *<string expC>* to write to the specified file, starting at the first character in *<string expC>*. If omitted, the entire string is written.

Property of File

Description *write()* writes a character string to a file. If the file was opened in append-only or read and append mode, the string is always added to the end of the file. Otherwise, the string is written starting at the current file pointer position, overwriting any existing characters. You must have either write or append access to use *write()*.

write() returns the number of bytes written to the file. If *write()* returns 0, no characters were written. Either *<expC>* is an empty string, or the write was unsuccessful.

Use *error()* to determine if an error occurred.

When *write()* finishes executing, the file pointer is located at the character immediately after the last character written. Use *seek()* to move the file pointer before or after you use *write()*.

To write to a text file, use *puts()*. *puts()* automatically adds the end-of-line character(s).

To read from a file, use *read()*.

Example Suppose you're exporting data from a table in a special format for another program. The export file must have the number of rows of data written in the file, starting at the 9th character. You extend the File class, adding methods to create the export file, write the data in the special format, and record the number of rows written. The following is the method that records the number of rows.

```
function recordRowsWritten()
    this.seek( 8 )                // 9th character == offset 8
    this.write( "" + this.rowsExported ) // Convert number to string to write
```

See also *create()*, *eof()*, *error()*, *open()*, *puts()*, *read()*, *seek()*

writeln()

Writes a character string and one or two end-of-line characters to a file previously opened with *create()* or *open()*. Returns the number of characters written.

Syntax *<oRef>.writeln(<string expC> [, <characters expN> [, <end-of-line expC>]])*

Property of File

Description *writeln()* is identical to *puts()*. See *puts()* for details.

_dbwinhome

Contains the home directory of the currently running instance of *Visual* dBASE.

Description There are two “home” directories:

- The directory where *Visual* dBASE is installed, by default:

`C:\Program Files\dBASE\Visual dBASE 75\`

- The directory where the actual executable file, VDB.EXE is installed. This is in the \Bin subdirectory of the installation directory, so by default, it's:

`C:\Program Files\dBASE\Visual dBASE 75\Bin\`

_dbwinhome contains the installation home directory, from which you can access all subdirectories. _dbwinhome contains the full path name whether SET FULLPATH is ON or OFF, and always includes the trailing backslash, as shown.

_dbwinhome is read-only.

To identify where the currently running instance of VDB.EXE is located, use HOME().

Example The following statement changes the directory to *Visual* dBASE's \Custom subdirectory:

```
cd "&_dbwinhome.Custom"
```

The macro operator is used to expand the path name contained in _dbwinhome. The period acts as the macro terminator. The resulting command looks like:

```
cd "C:\PROGRAM FILES\DBASE\VISUAL DBASE 75\Custom"
```

(The path name in _dbwinhome is all-uppercase.) The quotes are required because the path name contains spaces.

See Also CD, HOME(), SET DIRECTORY

Xbase

Every Xbase command and function includes an OODML section that lists the object-oriented DML equivalent, if there is one.

The examples in this chapter are mostly data processing and utility code. Data entry in *Visual* dBASE is done at another level, either using the form-based events that are melded into the Xbase worksets, or the new *Visual* dBASE 7 data objects, which replace most Xbase functionality and provide powerful new object-oriented capabilities.

The examples also do not use any new *Visual* dBASE 7 syntax, and thus are compatible with older versions of *Visual* dBASE.

Common command elements

The following sections detail command elements that are common to many Xbase commands and functions.

Filenames

Filenames are required for many Xbase commands. The filename may refer to a file on disk or a table in a database. A filename is indicated by *<filename>* in the syntax diagram and may be any one of the following forms:

- A filename, without the extension. When the filename refers to a table, *Visual* dBASE will assume the extension specified by the SET DBTYPE command (.DBF for dBASE and .DB for Paradox), which can be overridden in most commands with the command's TYPE clause. If the SET DATABASE command has been used to set a server database as the default, then the table name will be used as-is, without an extension. When the filename is not a table, there is always a default extension, which is listed in each command description.

- A filename, with the extension.
- A table in a database. Use the BDE Administrator to create database aliases. Specify the database alias in colons before the table name as follows:

```
:databaseAlias;tableName
```

If the database is not already open, *Visual* dBASE displays a dialog box in which you specify the parameters, such as a login name and password, necessary to establish a connection to that database.

- A filename skeleton. Use the ? and * as wildcard characters. A single ? is the same as *, meaning any filename. A dialog box is displayed from which you can choose a table, either a file on disk or a table from a database.

In all cases, the *<filename>* may be enclosed in string delimiters (single quotes, double quotes, or square brackets). Delimiters are required if *<table name>* contains spaces or other special characters. If the *<filename>* is contained in a variable and is not defined as an expression—functions expect filenames that are character expressions, commands do not—use the parentheses as indirection operators on the variable containing the *<filename>*.

If the *<filename>* refers to a file and does not contain a path and the file is not found in the current directory, then the path specified by SET PATH is also searched.

In many commands, the *<filename>* does not have to be specified in the statement. If it is omitted, *Visual* dBASE will display a dialog box from which you can choose a file to execute the command.

For commands that specifically create files and not tables, the database options are not allowed. If a dialog box is displayed, it will not include the controls to choose a database.

If you are about to overwrite a file, you will get a confirmation dialog box if SET SAFETY is ON. If SET SAFETY is OFF, the file will be overwritten without a warning.

Aliases

While some commands work only in the current work area, others allow you to specify the work area in which they perform their function. Work areas are referenced by their *alias*, which may take one of the following forms:

- The work area number, from 1 to 225
- A character string that contains a single letter from A through J, which correspond to work area 1 through 10. This is supported for compatibility.
- A character string containing the name of the work area: the name of the table, or the alias assigned to the work area when the table was opened. See the USE command for information on assigning aliases.

When using a letter or work area name as the alias in a function, the alias must be a character expression, usually the string enclosed in string delimiters. In a

command, the delimiters are optional and usually not used, unless the alias contains spaces or other special characters. In addition to the normal string delimiters (single quotes, double quotes, and square brackets), colons may be used to delimit aliases in commands.

The alias option is indicated by *<alias>* in the syntax tables. When you do not specify an alias, the command or function works on the current work area.

Command scope

Many Xbase commands have a *scope* option (not to be confused with the scope resolution operator) that dictates which records to process. The scope honors the current index order, filter, and key constraints. Three clauses comprise a command's scope:

- *<scope>*
- FOR *<for condition>*
- WHILE *<while condition>*

There are four different options for *<scope>*:

ALL All records, starting with the first.

REST Starting with the current record, processes all subsequent records in the table

NEXT *<expN>* Starting with the current record, processes the next *<expN>* records. NEXT 1 processes the current record only.

RECORD *<bookmark>* The individual record referenced by the bookmark *<bookmark>*. You may also specify a record number for DBF tables.

Different commands have different default scopes. In conjunction with *<scope>*, many commands have one or both of the following conditional clauses:

FOR *<for condition>* Specifies a condition that must evaluate to *true* for each record to be processed. If the *<for condition>* fails, that record is skipped and the next record is tested.

WHILE *<while condition>* Specifies a condition that must evaluate to *true* for processing to continue. The test is performed before processing each record. If the *<while condition>* fails, processing stops.

If you specify a FOR clause, the default scope of the command becomes ALL. If you specify a WHILE clause, with or without a FOR clause, the default scope of the command becomes REST.

ALIAS()

Returns the alias name of the current or a specified work area.

Syntax ALIAS([*<alias>*])

<alias> The work area you want to check. (If *<alias>* is a work area alias name, there is no reason to use this function because that alias name is what the function will return.)

Description ALIAS() returns the alias name of any work area within the current workset, in all uppercase. If no table is opened in the specified work area, ALIAS() returns an empty string ("").

Routines that do work in other work areas usually save the current work area before switching, and then switch back when done. Use ALIAS() to get the name of the current work area, then switch back using the SELECT command.

OODML There is no concept of the "current" Query object. You may refer to any Query object at any time through its object reference.

Example In this example, a function changes the index order of table of classes at a school:

```
PROCEDURE ClassesByRoom
  local cAlias
  cAlias = alias()
  select CLASSES
  set order to ROOM
  select ( cAlias )
```

This function saves the alias name of the currently selected table—which might be the table of teachers, students, or even the classes table—in the variable cAlias. When the function is done, that alias is reselected with the SELECT command, using the parentheses as indirection operators.

See Also DBF(), SELECT, SELECT(), USE, WORKAREA()

APPEND

Adds a new record to a table.

Syntax APPEND [BLANK]

BLANK Adds a blank record to the end of the table and makes the blank record the current record.

Description APPEND displays the currently selected table in an auto-generated data entry form and puts the form in Append mode. This has the same effect as using the EDIT command to display the data entry form and manually choosing Add Row from the menu or toolbar. This interactive APPEND is rarely used in applications because you have no control over the appearance of the data entry form.

The APPEND BLANK command adds a blank record to the current table and positions the record pointer on the new record, but it doesn't display a window

to edit the data. This is often done in an older style of dBASE programming, and is typically followed by REPLACE statements to store values into the newly-created record.

When accessing SQL tables, some database servers do not allow you to enter blank records. Also, constraints on tables created with non-null fields, including DBF7 tables, prevent entering records with fields left blank. In these cases, APPEND BLANK will fail and cause an error.

OODML Use the Rowset object's *beginAppend()* method. While APPEND BLANK creates a blank record first that you must delete if you decide to discard the new record, *beginAppend()* blanks the row buffer and creates a new row only if the row is modified and saved.

Example The following function is used when adding data to a table. It attempts to recycle records by looking for a blank deleted record. If one is not found the APPEND BLANK command is used to create a new record.

```
PROCEDURE NewRec
  set deleted off
  if seek( " " ) .and. deleted() .and. rlock()
    recall
  else
    append blank
  endif
  set deleted on
```

First, DELETED is turned OFF so that deleted records can be found. (The normal operation of the application has DELETED ON.) The SEEK() function looks for a record with a character key that starts with a blank space, which indicates a blank record; a valid index key cannot be blank. The table must be ordered on a character field when the function is called. If a blank record is found, the DELETED() function makes sure it's deleted, and an RLOCK() is attempted to prevent anyone else from grabbing the same record at the same time.

If all of these things are successful, the record is RECALLED and made available for use. Otherwise, a new blank record is created with APPEND BLANK. Either way, DELETED is turned back ON and the function is completed, leaving the record pointer at the new or recycled record.

To see the function that deletes records for recycling, see the example for BLANK.

See Also APPEND AUTOMEM, APPEND FROM, EDIT, SET CARRY, SET RELATION

APPEND AUTOMEM

Adds a new record to a table using the values stored in automem variables.

Syntax APPEND AUTOMEM

Description APPEND AUTOMEM adds a new record to the currently selected table and then replaces the value of fields in the table with the contents of corresponding automem variables. Automem variables are variables that have the same names and data types as the fields in the current table. Automem variables must be private or public; they cannot be local or static. If a field does not have a matching variable, that field is left blank.

APPEND AUTOMEM is used as part of data entry in an older style of dBASE programming. In *Visual* dBASE, controls in data entry forms are *dataLinked* to fields; there is no need for a set of corresponding variables. APPEND AUTOMEM is also used for programatically adding records to table. It is more convenient than using APPEND BLANK and REPLACE.

To use APPEND AUTOMEM to add records to a table, first create a set of automem variables. The USE...AUTOMEM command opens a table and creates the corresponding empty automem variables for that table. CLEAR AUTOMEM creates a set of empty automem variables for the current table or reinitializes existing automem variables to empty values. STORE AUTOMEM copies the values in the current record to automem variables. You may also create the individual variables manually.

When referring to the value of automem variables you need to prefix the name of an automem variable with M-> to distinguish the variable from the corresponding fields, which have the same name. The M-> prefix is not needed during variable assignment; the STORE command and the = and := operators do not work on Xbase fields.

OODML The Rowset object's contains an array of Field objects, accessed through its *fields* property. These Field objects have *value* properties that may be programmed like variables.

Example The following function is used to record access to an application as part of its startup process:

```
PROCEDURE LogRec
    private user, date, time
    use LOGREC in select()
    select LOGREC
    user = user()
    date = date()
    time = time()
    append automem
    use
```

The variables that will be used as automem variables are first declared private, to hide any variables of the same name that might exist. Then the table is opened in an unused work area and selected. The automem variables are created manually, using built-in functions. Finally, the values are appended to the table, and the table is closed.

See Also APPEND, CLEAR AUTOMEM, REPLACE AUTOMEM, STORE AUTOMEM, USE

APPEND FROM

Copies records from an existing table to the end of the current table.

Syntax APPEND FROM <filename>
 [FOR <condition>]
 [[TYPE] DBASE | PARADOX | SDF |
 DELIMITED [WITH
 <char> | BLANK]]
 [REINDEX]

<filename> The name of the file whose records you want to append to the current table.

FOR <condition> Restricts APPEND FROM to records in <filename> that meet <condition>. You can specify a FOR <condition> only for fields that exist in the current table. *Visual* dBASE pretends that the record is appended, then evaluates the <condition>. If it fails, the records is not actually appended.

[TYPE] DBASE | PARADOX | SDF | DELIMITED [WITH <char> | BLANK] Specifies the default file extension, and for text files, the text file format. For example, if you specify a .DBF file as the <filename> and TYPE PARADOX, the TYPE is ignored because the file is really a dBASE file. The TYPE keyword is included for readability only; it has no effect on the operation of the command. The following table provides a description of the different file formats that are supported:

| Type | Description |
|-----------------------|--|
| DBASE | A dBASE table. If you don't include an extension for <filename>, <i>Visual</i> dBASE assumes a .DBF extension. |
| PARADOX | A Paradox table. If you don't include an extension for <filename>, <i>Visual</i> dBASE assumes a .DB extension. |
| SDF | A System Data Format text file. Records in an SDF file are fixed-length, and the end of a record is marked with a carriage return and a linefeed. If you don't specify an extension, <i>Visual</i> dBASE assumes .TXT. |
| DELIMITED | A text file with fields separated by commas. These files are also referred to as CSV (Comma Separated Value) files. Character fields may be delimited with double quotation marks; the quotes are required if the field itself contains a comma. Each carriage return and linefeed indicates a new record. If you don't specify an extension, <i>Visual</i> dBASE assumes .TXT. |
| DELIMITED WITH <char> | Indicates that character data is delimited with the character <char> instead of with double quotes. For example, if delimited with a single quote instead of a double quote, the clause would be: DELIMITED WITH ' |
| DELIMITED WITH BLANK | Indicates that data is separated with spaces instead of commas, with no delimiters. |

REINDEX Rebuilds all open index files after APPEND FROM finishes executing. Without REINDEX, *Visual* dBASE updates all open indexes after appending each record from <filename>. When the current table has multiple open indexes or contains many records, APPEND FROM executes faster with the REINDEX option.

Description Use the APPEND FROM command to add data from another file or table to the end of the current table. You can append data from dBASE tables or files in other formats. Data is appended to the current table in the order in which it is stored in the file you specify.

When you specify a table as the source of data, fields are copied by name. If a field in the current table does not have a matching field in the source table, those fields will be blank in the appended records. If the field types do not match, type conversion is attempted. For example, if a field named ID in the current table is character field, but the ID field in the source table is numeric, the number will be converted into a string when it is appended.

When appending text files, SDF or DELIMITED, there is no data type in the source file; everything is a string. For non-character fields, the strings should be in the following format to match the data type in the table:

- For logical or boolean fields, the letters T, t, Y, and Y indicate *true*. All other letters and blanks are considered *false*.
- Dates must be in the format YYYYMMDD.

If the field of the current table is shorter than the matching field of the source table, *Visual* dBASE truncates the data.

If SET DELETED is OFF, dBASE adds records from a source dBASE table that are marked for deletion and *doesn't* mark them for deletion in the current table. If SET DELETED is ON, dBASE doesn't add records from a source dBASE table that are marked for deletion.

When importing data from other files, remove column headings and leading blank rows and columns; otherwise, this data is also appended.

ODML Use the UpdateSet object's *append()* or *appendUpdate()* method to append data from other tables.

See Also APPEND, APPEND AUTOMEM, COPY, REINDEX, SET DELETED

APPEND FROM ARRAY

Adds to the current table one or more records containing data stored in a specified array.

Syntax APPEND FROM ARRAY <array>
[FIELDS <field list>]

[FOR <condition>]
[REINDEX]

<array> A reference to the array containing the data to store in the current table as records.

FIELDS <field list> Appends <array> data only to the fields in <field list>. Without FIELDS <field list>, APPEND FROM ARRAY appends to all the fields in the table, starting with the first field.

FOR <condition> Restricts APPEND FROM ARRAY to array rows in <array> that meet <condition>. The FOR <condition> should reference the fields in the current table. *Visual* dBASE pretends that the record is appended, then evaluates the <condition>. If it fails, the records is not actually appended.

REINDEX Rebuilds open indexes after all records have been changed. Without REINDEX, *Visual* dBASE updates all open indexes after appending each record from <array>. When the current table has multiple open indexes or contains many records, APPEND FROM ARRAY executes faster with the REINDEX option.

Description APPEND FROM ARRAY treats one- and two-dimensional arrays as tables, with columns corresponding to fields and rows corresponding to records. A one-dimensional array works as a table with only one row; therefore, you can append only one record from a one-dimensional array. A two-dimensional array works as a table with multiple rows; therefore, you can append as many records from a two-dimensional array as it has rows.

When you append data from an array to the current table, *Visual* dBASE appends each array row as a single record. If the table has more fields than the array has columns, the excess fields are left empty. If the array has more columns than the table has fields, the excess columns are ignored. The data in the first column is added to the first field's contents, the data in the second column to the second field's contents, and so on.

The data types of the array must match those of corresponding fields in the table you are appending. If the data type of an array element and a corresponding field don't match, *Visual* dBASE returns an error.

If the current table has a memo field, *Visual* dBASE ignores this field. For example, if the second field is a memo field, *Visual* dBASE adds the data in the array's first column to the first field's contents, and the data in the array's second column to the third field's contents.

Use APPEND FROM ARRAY as an alternative to automem variables when you need to transfer data between tables where the structures are similar but the field names are different.

OODML Use two nested loops to first call the Rowset object's *beginAppend()* method to create the new rows, and then to copy the elements of the array into the *value* properties of the Field objects in the rowset's *fields* array.

See Also APPEND AUTOMEM, COPY TO ARRAY, DECLARE, REPLACE FROM ARRAY, STORE AUTOMEM

APPEND MEMO

Appends a text file to a memo field.

Syntax APPEND MEMO <memo field> FROM <filename>
[OVERWRITE]

<memo field> The memo field to append to.

FROM <filename> The text file to append. The default extension is .TXT.

OVERWRITE Erases the contents of the current record memo field before copying the contents of <filename>.

Description Use the APPEND MEMO command to insert the contents of a text file into a memo field. You may use an alias name and the alias operator (that is, *alias->memofield*) to specify a memo field in the current record of any open table.

APPEND MEMO is identical to REPLACE MEMO, except that APPEND MEMO defaults to appending the file to the current contents of the memo field and has the option of overwriting, while REPLACE MEMO is the opposite.

While memo fields may contain types of information other than text, binary fields are recommended for storing images, sound, and other user-defined binary type information. Use OLE fields for linking to OLE documents from other Windows applications.

OODML Use the Field object's *replaceFromFile()* method.

Example The following event handler displays a dialog to pick a text file, then adds the contents of that file to a memo field. The date and time are written to the memo field before the added file.

```
PROCEDURE addTextButton_onClick
    local cFile, cCRLF
    cCRLF = chr( 13 ) + chr( 10 )
    cFile = getfile( "*.txt", "Text file to import" )
    if "" # cFile
        replace MEMO_FIELD with cCRLF + dtoc( date() ) + " " + time() + cCRLF additive
        append memo MEMO_FIELD from ( cFile )
    endif
```

The date and time, with a line break before and after, is written to the memo field using the REPLACE command with the ADDITIVE option for memo fields.

GETFILE() will return an empty string if no file is selected. In the IF statement, the order of the empty string and the variable cFile is important. If they were

the other way around and SET EXACT is OFF, then the IF statement would always be *false*.

The parentheses are used as indirection operators to get the name of the file from the variable. Without them, *Visual* dBASE would attempt to append a file named cFile.

See Also COPY MEMO, REPLACE BINARY, REPLACE MEMO, REPLACE MEMO...WITH ARRAY, REPLACE OLE

AVERAGE

Computes the arithmetic mean (average) of specified numeric fields in the current table.

Syntax AVERAGE [*<exp list>*]
 [*<scope>*]
 [FOR *<condition 1>*]
 [WHILE *<condition 2>*]
 [TO *<memvar list>* | TO ARRAY *<array>*]

<exp list> The numeric fields, or expressions involving numeric fields, to average.

<scope>

FOR *<condition 1>*

WHILE *<condition 2>* The scope of the command. The default scope is ALL.

TO *<memvar list>* | TO ARRAY *<array>* Initializes and stores averages to the variables (or properties) of *<memvar list>* or stores averages to the existing array *<array>*. If you specify an array, each field average is stored to elements in the order in which you specify the fields in *<exp list>*. If you don't specify *<exp list>*, each field average is stored in field order. *<array>* can be a single- or multidimensional array; the array elements are accessed via their element numbers, not their subscripts.

Description The AVERAGE command computes the arithmetic means (averages) of numeric expressions and stores the results in specified variables or array elements. If you store the values in variables, the number of variables must be exactly the same as the number of fields or expressions averaged. If you store the values in an array, the array must already exist, and the array must contain at least as many elements as the number of averaged expressions.

If SET TALK is ON, AVERAGE also displays its results in the results pane of the Command window. The SET DECIMALS setting determines the number of decimal places that AVERAGE displays. Numeric fields in blank records are evaluated as zero. To exclude blank records, use the ISBLANK() function in defining a FOR condition. EMPTY() excludes records in which a specified expression is either 0 or blank.

BEGINTRANS()

OODML Loop through the rowset to calculate the average.

Example The following example uses AVERAGE to calculate the average year to date sales for all companies in the Company table and displays it in Text control on a form:

```
select COMPANY
average YTD_SALES to form.ytdSalesText.text
```

See Also CALCULATE, COUNT, SUM, TOTAL

BEGINTRANS()

Begins transaction logging.

Syntax BEGINTRANS([<database name expC> [,<isolation level expN>]])

<database name expC> The BDE alias of the SQL database in which to begin the transaction.

- If <database name expC> is omitted but a SET DATABASE statement has been issued, BEGINTRANS() refers to the database in the SET DATABASE statement.
- If <database name expC> is omitted and no SET DATABASE statement has been issued, the default database, which supports DBF and DB tables is used.

<isolation level expN> Specifies a pre-defined server-level transaction isolation scheme.

- Valid values for <isolation level> are:

| Value | Description |
|-------|---|
| 0 | Server's default isolation level |
| 1 | Uncommitted changes read (dirty read) |
| 2 | Committed changes read (read committed) |
| 3 | Full read repeatability (repeatable read) |

- <isolation level> is not supported for DBF and DB tables.
- If an invalid value is given for <isolation level>, a "Value out of range" error is generated.
- The <isolation level> is server-specific; a "Not supported" error will result from the database engine if an unsupported level is specified.

Note If you include <database name expC> when you issue BEGINTRANS(), you must also include it in subsequent COMMIT() or ROLLBACK() statements within that transaction. If you don't, dBASE ignores the COMMIT() or ROLLBACK() statement.

- Description** Separate changes that must be applied together are considered to be a transaction. For example, transferring money from one account to another means debiting one account and crediting another. If for whatever reason one of those two changes cannot be done, the whole transaction is considered a failure and any change that was made must be undone.
- Transaction logging records all the changes made to all the tables in a database. If no errors are encountered while making the individual changes in the transaction, the transaction log is cleared with COMMIT() and the transaction is done. If an error is encountered, all changes made so far are undone by calling ROLLBACK().
- All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.
- You can't nest transactions with BEGINTRANS(). If you issue BEGINTRANS() against an SQL database that does not support transactions, or if a server error occurs, BEGINTRANS() returns *false*. Otherwise, it returns *true*. If BEGINTRANS() returns *false*, use SQLERROR() or SQLMESSAGE() to determine the nature of the server error that might have occurred.
- OODML** Call the *beginTrans*() method of the Database object.
- See Also** COMMIT(), FLOCK(), RLOCK(), ROLLBACK(), SET EXCLUSIVE, SQLERROR(), SQLMESSAGE()

BINTYPE()

Returns the predefined type number of a specified binary field.

Syntax BINTYPE([<*field name*>])

<*field name*> The name of a field in the current table.

Description BINTYPE() returns the predefined binary type number of a binary field in the current table. Using this command, you can determine the type of data stored in the field. The values returned by BINTYPE() are the following:

| Predefined binary type numbers | Description |
|--------------------------------|-------------------------|
| 1 to 32K – 1 (1 to 32,767) | User-defined file types |
| 32K (32,768) | .WAV files |
| 32K + 1 (32,769) | .BMP and .PCX files |

BINTYPE() returns an error if a non-binary field is specified. It returns a value of 0 if the binary field is empty.

OODML No direct equivalent. You may be able to ascertain the data type by examining the data in the *value* of the Field object.

See Also COPY BINARY, PLAY SOUND REPLACE BINARY, RESTORE IMAGE

BLANK

Fills fields in records with blanks.

Syntax

```
BLANK
[<scope>]
[FOR <condition 1>]
[WHILE <condition 2>]
[FIELDS
    <field list> | [LIKE <skeleton 1>] [EXCEPT <skeleton 2>]]
[REINDEX]
```

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is NEXT 1, the current record only.

FIELDS <field list> | LIKE <skeleton 1> | EXCEPT <skeleton 2> The fields to blank. Without FIELDS, BLANK replaces all fields. If you specify FIELDS LIKE <skeleton 1>, the BLANK command restricts the fields that are blanked to the fields that match <skeleton 1>. Conversely, if you specify FIELDS EXCEPT <skeleton 2>, the BLANK command makes all fields blank except those whose names match <skeleton 2>.

REINDEX Rebuilds all open indexes after BLANK finishes executing. Without REINDEX, Visual dBASE updates all open indexes after each record is made blank. When the current table has multiple open indexes or contains many records, BLANK executes faster with the REINDEX option.

Description

Use BLANK to blank-out fields or records in the current table. BLANK has the same effect as using REPLACE on each field with a *null* value. For DBF7, DB, and SQL tables, the fields are replaced with *null* values. For earlier versions of DBF tables, the fields are replaced with blanks (spaces). EMPTY() and ISBLANK() return *true* for a field whose value has been replaced using BLANK. BLANK fills an existing record with the same values as APPEND BLANK. Updates to open indexes are performed after each record or a set of records is blanked.

The distinction between blank and zero values in numeric fields can be significant when you use commands such as AVERAGE and CALCULATE.

For earlier DBF tables, blank numeric fields evaluate to zero and blank logical or boolean fields evaluate to *false*. In DBF7 tables, which support true null values, the value of the field is *null*, although some commands may display the null value as zero or *false*.

OODML Use a loop to assign *null* values to the *value* properties of the Field objects.

Example The followin function blanks records before deleting them, making them available for recycling:

```
PROCEDURE DelRec
  blank
  delete
```

To see the function that reclaims recycled records, see the example for APPEND.

See Also APPEND, ISBLANK(), EMPTY(), REPLACE

BOF()

Indicates if the record pointer in a table is at the beginning of the file.

Syntax BOF([<alias>])

<alias> The work area you want to check.

Description BOF() returns *true* when the record pointer has just moved before the first logical record of the table in the specified work area; otherwise, it returns *false*. For example, if you issue SKIP -1 when the record pointer is on the first record, BOF() returns *true*. If you attempt to navigate backwards when BOF() is *true*, an error occurs.

However, unlike EOF(), the record pointer can never stay before the first record. After the record pointer has moved past the first record, it is automatically moved back to the first record, even though BOF() remains *true*. Subsequent navigation will cause BOF() to return *false* unless the navigation moves the record pointer before the first record again.

When you first USE a table, BOF() can never be *true*, but EOF() can if the table is empty, or you are using a conditional index with no matching records.

If no table is open in the specified work area, BOF() also returns *false*.

OODML The Rowset object's *endOfSet* property is *true* when the row pointer is past either end of the rowset. Unlike BOF() and EOF(), there is symmetry with the *endOfSet* property. You can determine which end you're on based on the direction of the last navigation.

There is also an *atFirst*() method that determines whether you are on the first row in the rowset.

Example The following is an event handler for a button that navigates backward through a table:

```
PROCEDURE prevButton_onClick
  if .not. bof()
```

BOOKMARK()

```
        skip -1
    endif
    if bof()
        msgbox( "First record", "Navigation", 64 )
    endif
```

This example demonstrates an atypical programming construct: instead of using IF and ELSE, there is an IF statement for a condition, followed by a separate IF statement for the opposite condition. In this case, it works like this: if you are already at BOF() you do not want to attempt to navigate backwards, because that will cause an error. But if you end up at BOF(), or if you're already at BOF(), then you will get a message.

See Also EOF(), RECNO(), SKIP

BOOKMARK()

Returns a bookmark for the current record.

Syntax BOOKMARK([<alias>])
<alias> The work area you want to check.

Description BOOKMARK() returns a value for the current record. The value returned by BOOKMARK() is of a special unprintable data type called bookmark. BOOKMARK() returns an empty bookmark if no table is open in the current work area.

When used with the GO command, bookmarks let you navigate to particular records.

Unlike record numbers, which work only with DBF tables, bookmarks work with all tables, including DBFs. Bookmarks are only guaranteed to be valid for the table from which they are created.

Bookmark values can be used in all commands and functions that can otherwise use a record number, and with relational operators to check for equality and relative position in a table.

OODML Use the Rowset object's *bookmark()* method.

See Also GO, RECNO()

BROWSE

Provides display and editing of records in a table format.

Syntax BROWSE
[COLOR <color>]


```
[FIELDS <field 1> [<field option list 1>] |
    <calculated field 1> = <exp 1> [<calculated field option list 1>]
    [, <field 2> [<field option list 2>] |
    <calculated field 2> = <exp 2> [<calculated field option list 2>]...]]
[FREEZE <field 3>]
[LOCK <expN 1>]
[NOAPPEND]
[NOEDIT | NOMODIFY]
```

COLOR <color> Specifies the color of the cells in the BROWSE. The current highlighted cell has its own, fixed color. The <color> is made up of a foreground color and a background color, separated by a forward slash (/). You may use a Windows-named color, one of the basic 16-color color codes, or a user-defined color name. For more information on colors, see *colorNormal* (page 15-75).

FIELDS <field 1> [<field option list 1>] |
<calculated field 1> = <exp 1> [<calculated field option list 1>]
[, <field 2> [<field option list 2>] |
<calculated field 2> = <exp 2> [<calculated field option list 2>] ...]] Displays the specified fields, in the order they're listed, in the Table window. Options for <field option list 1>, <field option list 2>, which apply to <field 1>, <field 2>, and so on, affect the way these fields are displayed. These options are as follows:

| Option | Description |
|-------------------------------------|---|
| \<column width> | The width of the column within which <field 1> appears when <field 1> is character type |
| \B = <exp 1>, <exp 2> | RANGE option; forces any value entered in <field 1> to fall within <exp 1> and <exp 2>, inclusive. |
| \C=<color> | COLOR option; sets the foreground and/or background colors of the column according to the values specified in <color> |
| \H = <expC> | HEADER option; causes <expC> to appear above the field column in the Table window, replacing the field name |
| \P = <expC> | PICTURE option; displays <field 1> according to the PICTURE or FUNCTION clause <expC> |
| \V = <condition> [\E = <expC>] | VALID option; allows a new <field 1> value to be entered only when <condition> evaluates to <i>true</i> ERROR MESSAGE option; \E = <expC> causes <expC> to appear when <condition> evaluates to <i>false</i> |

Note You may also use the forward slash (/) instead of the backslash (\) when specifying only a single option in a field option list.

Read-only calculated fields are composed of an assigned field name and an expression that results in the calculated field value, for example:

```
browse fields commission = RATE * SALEPRICE
```

Options for calculated fields affect the way these fields are displayed. These options are as follows:

| Option | Description |
|-----------------|--|
| \<column width> | The width of the column within which <calculated field 1> is displayed |
| \H = <expC> | Causes <expC> to appear above the calculated field column in the Table window, replacing the calculated field name |

FREEZE <field 3> Restricts editing to <field 3>, although other fields are visible.

LOCK <expN 2> Keeps the first <expN 2> fields in place onscreen as you move the cursor to fields on the right.

NOAPPEND Prevents records from being added when you cursor down past the last record in the Table window.

NOEDIT | NOMODIFY Prevents you from modifying records from the Table window.

Description The BROWSE command opens a table grid in a window, displaying the fields in the currently selected work area. It is intended more for interactive use; in an application, you have more control over a Browse or Grid object in a form.
The BROWSE command is modeless. After the window is opened, the next statement is executed.

OODML Use a Grid control on a form.

See Also APPEND, EDIT, SET FIELDS, SET MEMOWIDTH, SET RELATION

CALCULATE

Performs financial and statistical operations for values of records in the current table.

Syntax CALCULATE <function list>
[<scope>]
[FOR <condition 1>]
[WHILE <condition 2>]
[TO <memvar list> | TO ARRAY <array>]

<function list> You can use one or more of the following functions:

| Function | Purpose |
|-------------|---|
| AVG(<expN>) | Calculates the average of the specified numeric expression. |
| CNT() | Counts the number of records in the current table. |

| Function | Purpose |
|--------------------------------------|--|
| MAX(<expC> <expN> <expD>) | Calculates the maximum value of the specified numeric, character, or date expression. |
| MIN(<expC> <expN> <expD>) | Calculates the minimum value of the specified numeric, character, or date expression. |
| NPV(<expN 1>, <expN 2> [, <expN 3>]) | Calculates the net present value of the numeric values in <expN 2>; <expN 1> is the periodic interest rate, expressed as a decimal; <expN 3> is the initial investment and is generally a negative number. |
| STD(<expN>) | Calculates the standard deviation of the specified numeric expression. |
| SUM(<expN>) | Calculates the sum of the specified numeric expression. |
| VAR(<expN>) | Calculates the variance of the specified numeric expression. |

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

TO <memvar list> | TO ARRAY <array> Initializes and stores the results to the variables (or properties) of <memvar list> or stores results to the existing array <array>. <array> can be a single- or multidimensional array; the array elements are accessed via their element numbers, not their subscripts.

Description

CALCULATE uses one or more of the eight associated functions listed in the previous table to calculate and store sums, maximums, minimums, averages, variances, standard deviations, or net present values of specified expressions. The expressions are usually, but not required to be, based on fields in the current table. You can calculate values in a work area other than the current work area if you set a relation between the work areas.

CALCULATE can also return the count or number of records in the current table. These special functions, with the exception of MAX() and MIN(), can be used only with CALCULATE.

CALCULATE can use the same function on different expressions or different functions on the same expression. For instance, if your table contains a Salary field and a Bonus field, you can issue the command:

```
calculate sum(SALARY), sum(BONUS), avg(SALARY), avg(12 * (SALARY + BONUS))
```

CALCULATE stores results to variables or to an existing array in the order of the specified functions. If you store the results to memory variables, specify the same number of variables as the number of functions in the CALCULATE command line. If you store the values in an array, the array must already exist, and the array must contain at least as many elements as the number calculations.

If SET TALK is ON, CALCULATE displays the results in the result pane of the Command window. The SET DECIMALS setting determines the number of decimal places that CALCULATE displays.

CALCULATE treats a blank numeric field as containing 0 and includes the field in its calculations. For example, if you calculate the average of a numeric field in a table containing ten records, five of which are blank, CALCULATE divides the sum by 10 to find the average. Furthermore, if you calculate the minimum of the same table field and five records contain positive non-zero numbers and the five others are blank in the same fields, CALCULATE returns 0 as the minimum. If you want to exclude blank fields when using CALCULATE, be sure to specify a condition such as FOR .NOT. ISBLANK(numfield).

Although you can use the SUM or AVERAGE commands to find sums and averages, if you are mixing sums and averages, CALCULATE is faster because it runs through the table just once while making all specified calculations.

OODML Loop through the rowset to calculate the values.

See Also AVERAGE, DECLARE, MAX(), MIN(), SET FIELDS, SET RELATION, SUM

CHANGE()

Returns *true* if another user has changed a record since it was read from disk.

Syntax CHANGE([<alias>])

<alias> The work area you want to check.

Description Use CHANGE() to determine if another user has made changes to a record since it was read from disk. If the record has been changed, you might want to display a message to the user before allowing the user to continue.

Note CHANGE() only works with DBF tables.

For CHANGE() to return information, the table being checked must have a _DBASELOCK field. Use CONVERT to add a _DBASELOCK field to a table. If the table doesn't contain a _DBASELOCK field, CHANGE() returns *false*.

CHANGE() compares the counter in the workstation's memory image of _DBASELOCK to the counter stored on disk. If they are different, the record has changed, and CHANGE() returns *true*.

You can reset the value of CHANGE() to *false* by moving the record pointer. GOTO BOOKMARK() rereads the current record's _DBASELOCK field, and a subsequent CHANGE() returns *false*, unless another user has changed the record in the interim between moving to it and issuing CHANGE().

OODML Call rowset.fields["_DBASELOCK"].changed()

See Also CONVERT, FLOCK(), LKSYS(), RLOCK(), SET EXCLUSIVE, SET REFRESH

CLEAR AUTOMEM

Initializes automem variables with empty values for the current table.

Syntax CLEAR AUTOMEM

Description Use CLEAR AUTOMEM to initialize a set of automem variables containing empty values for the current table. CLEAR AUTOMEM creates any automem variables that don't exist already. If the variables exist, CLEAR AUTOMEM reinitializes them. If no table is in use, CLEAR AUTOMEM doesn't create any variables.

CLEAR AUTOMEM creates normal variables. They default to private scope when CLEAR AUTOMEM is executed in a program or function. If there is a danger of overwriting previously created public or private variables with the same name, you must declare the new automem variables PRIVATE individually by name before issuing CLEAR AUTOMEM, just as you would if you created the variables manually.

Automem variables have the same names and data types as the fields in an active table. You can create empty automem variables automatically for the current table by using CLEAR AUTOMEM or USE...AUTOMEM, or manually by using STORE or the assignment operators.

OODML The Rowset object's contains an array of Field objects, accessed through its *fields* property. These Field objects have *value* properties that may be programmed like variables.

See Also APPEND, STORE, USE

CLEAR FIELDS

Removes the fields list defined with the SET FIELDS TO command.

Syntax CLEAR FIELDS

Description Use CLEAR FIELDS to remove the SET FIELDS TO *<field list>* setting in all work areas and automatically turn SET FIELDS to OFF, thus making all fields in all open tables accessible. You can use CLEAR FIELDS prior to specifying a new fields list with SET FIELDS TO. You might also want to use CLEAR FIELDS at the end of a program. CLEAR FIELDS has the same effect as SET FIELDS TO with no options.

OODML No direct equivalent. When accessing the *fields* array, you may include program logic to include or exclude specific fields.

See Also SET FIELDS

CLOSE DATABASES

Closes databases, including their tables and indexes.

Syntax CLOSE DATABASES [<database name list>]

<database name list> The list of database names, separated by commas. If no list is specified, all open databases are closed.

Description Closing a database closes all the open tables in the database, including all the index, memo, and other associated files. For the default database, which gives access to DBF and DB tables, this means all open tables in all work areas.

CLOSE DATABASES only closes those tables opened in the current workset. For more information on worksets, see CREATE SESSION.

OODML Set the *active* property of the Database object (or all its Query objects) to *false*.

See Also CLOSE TABLES, USE

CLOSE INDEXES

Closes DBF index files in the current work area.

Syntax CLOSE INDEXES

Description Closes index (.MDX and .NDX) files open in the current work area. This option does not close the production .MDX file.

OODML Clear the *indexName* property of the Rowset object.

See Also CLOSE TABLES, SET INDEX TO

CLOSE TABLES

Closes all tables.

Syntax CLOSE TABLES

Description Closes all tables in all work areas or all tables in the current database, if one is selected.

CLOSE TABLES only closes those tables opened in the current workset. For more information on worksets, see CREATE SESSION.

OODML Set the *active* property of all the Query objects to *false*.

See Also CLOSE DATABASES, USE

COMMIT()

Clears the transaction log, committing all logged changes.

Syntax COMMIT([<database name expC>])

<database name expC> The name of the database in which to complete the transaction.

- If you began the transaction with BEGINTRANS(<database name expC>), you must issue COMMIT(<database name expC>). If instead you issue COMMIT(), Visual dBASE ignores the COMMIT() statement.
- If you began the transaction with BEGINTRANS(), <database name expC> is an optional COMMIT() argument. If you include it, it must refer to the same database as the SET DATABASE TO statement that preceded BEGINTRANS().

Description A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling ROLLBACK(). Otherwise, COMMIT() is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

For more information on transactions, see BEGINTRANS().

ODML Call the *commit()* method of the Database object.

See Also BEGINTRANS(), ROLLBACK(), SET EXCLUSIVE

CONTINUE

Continues a search for the next record that meets the conditions specified in a previously issued LOCATE command.

Syntax CONTINUE

Description CONTINUE continues the search of the last LOCATE issued in the selected work area. When you issue the LOCATE command, the current table is searched sequentially for the first record that matches the search criteria.

If a record is found, the record pointer is left at the matching record. To continue the search, issue the CONTINUE command. Whenever a match is found, FOUND() returns *true*. If match is not found, the record pointer is left after the last record checked, which usually leaves it at the end-of-file. Also, FOUND() returns *false*.

If SET TALK is ON, CONTINUE will display the record number of the matching record in the result pane of the Command window if you are searching a DBF table. If no match is found, CONTINUE will display "End of Locate scope".

If you issue CONTINUE without first issuing a LOCATE command for the current table, an error occurs..

OODML Use the Rowset object's *locateNext()* method. This method also allows going backwards or to the *n*th match.

See Also EOF(), FOUND(), LOCATE, SEEK, SEEK()

COPY

Copies records from the current table to another table or text file.

Syntax COPY TO <filename>
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [FIELDS <field list>]
 [[TYPE] DBASE | DBMEMO3 | PARADOX | SDF |
 DELIMITED [WITH
 <char> | BLANK]] |
 [[WITH] PRODUCTION]

TO <filename> Specifies the name of the table or file you want to create.

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

FIELDS <field list> Specifies which fields to copy to the new table.

[TYPE] DBASE | DBMEMO3 | PARADOX | SDF |

DELIMITED [WITH <char> | BLANK] Specifies the format of the file to which you want to copy data. The TYPE keyword is included for readability only; it has no effect on the operation of the command. The following table provides a description of the different file formats that are supported:

| Type | Description |
|---------|--|
| DBASE | A dBASE table. If you don't include an extension for <filename>, Visual dBASE assumes a .DBF extension. |
| DBMEMO3 | A table (.DBF) and memo (.DBT) files in dBASE III PLUS format. |
| PARADOX | A Paradox table. If you don't include an extension for <filename>, Visual dBASE assumes a .DB extension. |

| Type | Description |
|--------------------------|---|
| SDF | A System Data Format text file. Records in an SDF file are fixed-length, and the end of a record is marked with a carriage return and a linefeed. If you don't specify an extension, <i>Visual</i> dBASE assumes .TXT. |
| DELIMITED | A text file with fields separated by commas. These files are also referred to as CSV (Comma Separated Value) files. Character fields are delimited with double quotation marks. Each carriage return and linefeed indicates a new record. If you don't specify an extension, <i>Visual</i> dBASE assumes .TXT. |
| DELIMITED WITH <char> | Indicates that character data is delimited with the character <char> instead of with double quotes. For example, if delimited with a single quote instead of a double quote, the clause would be: DELIMITED WITH ' |
| DELIMITED WITH BLANK | Indicates that data is separated with spaces instead of commas, with no delimiters. |

[WITH] PRODUCTION Specifies copying the production .MDX file along with the associated table. This option can be used only when copying to another dBASE table.

Description Use COPY to copy all or part of a table to a file of the same or a different type. If an index is active, COPY arranges the records of the new table or file according to the indexed order. The COPY command does not copy a _DBASELOCK field in a table that you've created with CONVERT.

When COPYING to text files, SDF or DELIMITED, non-character fields are written as follows:

- Numbers are written as-is.
- Logical or boolean fields use the letter T for *true* and F for *false*.
- Dates are written in the format YYYYMMDD.

If you COPY a table containing a memo field to another dBASE table, *Visual* dBASE creates another file with the same name as the table but having a .DBT extension, and copies the contents of the memo field to it. If, however, you use the SDF or DELIMITED options and COPY to a text file, *Visual* dBASE doesn't copy the memo fields.

Deleted records are copied to the target file (if it's a dBASE table) unless a FOR or WHILE condition excludes them or unless SET DELETED is ON. Deleted records remain marked for deletion in the target dBASE table.

You can use COPY to create a file containing fields from more than one table. To do that, open the source tables in different work areas and define a relation between the tables. Use SET FIELDS TO to select the fields from each table that you want to copy to a new file. Before you issue the COPY command, SET FIELDS must be ON and you must be in the work area in which the parent table resides.

The COPY command does not verify that the files you build are compatible with other software programs. You may specify field lengths, record lengths,

number of fields, or number of records that are incompatible with other software. Check the file limitations of your other software program before exporting tables using COPY.

OODML Use the UpdateSet object's *copy()* method. Set filter options in the *source* rowset.

See Also APPEND FROM, CONVERT, COPY FILE, COPY STRUCTURE, COPY TABLE, COPY TO...STRUCTURE EXTENDED, SET DELETED, SET FIELDS

COPY BINARY

Copies the contents of the specified binary field to a file.

Syntax COPY BINARY <field name> TO <filename>
[ADDITIVE]

<field name> The binary field to copy.

TO <filename> The name of the file where the contents of the binary field are copied. For predefined binary file types, *Visual* dBASE assigns the appropriate extension, for example, .BMP, .WAV, and so on. For user-defined binary type fields, dBASE assigns a .TXT extension by default.

ADDITIVE Appends the contents of the binary field to the end of an existing file. Without the ADDITIVE option, *Visual* dBASE overwrites the previous contents of the file.

Description Use COPY BINARY to export data from a binary field in the current record to a file. You can use binary fields to store text, images, sound, video, and other user-defined binary data.

If you specify the ADDITIVE option, *Visual* dBASE appends the contents of the binary field to the end of the named file, which lets you combine the contents of binary fields from more than one record. When you don't use ADDITIVE, *Visual* dBASE displays a warning message before overwriting an existing file if SET SAFETY is ON. Note that you can't combine the data from more than one field for many of the predefined binary data types. For example, you can store only a single image in a binary field or file, so do not use the ADDITIVE option of COPY BINARY when copying an image to a file.

OODML Use the Field object's *copyToFile()* method.

See Also APPEND MEMO, BINTYPE(), CLASS IMAGE, COPY, COPY FILE, COPY MEMO, PLAY SOUND, REPLACE BINARY, RESTORE IMAGE

COPY MEMO

Copies the contents of the specified memo field to a file.

Syntax COPY MEMO <memo field> TO <filename>
[ADDITIVE]

<memo field> The memo field to copy.

TO <filename> | ? The name of the text file where text will be copied. The default extension is .TXT.

ADDITIVE Appends the contents of the memo field to the end of an existing text file. Without the ADDITIVE option, *Visual* dBASE overwrites any previous text in the text file.

Description Use COPY MEMO to export memo file text in the current record to a text file. You can also use COPY MEMO to copy images or other binary-type data to a file; however, binary fields are recommended for storing images, sound, and other user-defined binary information.

If you specify the ADDITIVE option, *Visual* dBASE appends the contents of the memo field to the end of the named file, which lets you combine the contents of memo fields from more than one record. When you don't use ADDITIVE, dBASE displays a warning message before overwriting an existing file if SET SAFETY is ON. You can store only a single image in either a memo field or in a file, so do not use the ADDITIVE option of COPY MEMO when copying an image to a file. (RESTORE IMAGE can display an image stored in either a memo field or a text file.)

OODML Use the Field object's *copyToFile()* method.

See Also APPEND MEMO, COPY, COPY BINARY, COPY FILE, REPLACE BINARY, REPLACE OLE

COPY STRUCTURE

Creates an empty table with the same structure as the current table.

Syntax COPY STRUCTURE TO <filename>
[[TYPE] PARADOX | DBASE]
[FIELDS <field list>]
[[WITH] PRODUCTION]

<filename> The name of the table you want to create.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if <filename> does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

FIELDS <field list> Determines which fields *Visual* dBASE includes in the structure of the new table. The fields appear in the order specified by <field list>.

[WITH] PRODUCTION Creates a production .MDX file for the new table. The new index file has the same index tags as the production index file associated with the original table.

Description The COPY STRUCTURE command copies the structure of the current table but does not copy any records. If SET SAFETY is OFF, *Visual* dBASE overwrites any existing tables of the same name without issuing a warning message.

The COPY STRUCTURE command copies the entire table structure unless limited by the FIELDS option or the SET FIELDS command. When you issue COPY STRUCTURE without the FIELDS *<field list>* option, *Visual* dBASE copies the fields in the SET FIELDS TO list to the new table. The _DBASELOCK field created with the CONVERT command is not copied to new tables.

You can use COPY STRUCTURE to create an empty table structure with fields from more than one table. To do so,

- 1 Open the source tables in different work areas.
- 2 Use the FIELDS *<field list>* option, including the table alias for each field name not in the current table.

OODML No equivalent.

See Also APPEND, APPEND FROM, COPY, COPY STRUCTURE EXTENDED, DISPLAY STRUCTURE, MODIFY STRUCTURE, SET FIELDS, SET SAFETY

COPY STRUCTURE EXTENDED

Creates a new table whose records contain the structure of the current table.

Syntax COPY STRUCTURE EXTENDED TO *<filename>*
[[TYPE] PARADOX | DBASE]

or

COPY TO *<filename>*
STRUCTURE EXTENDED
[[TYPE] PARADOX | DBASE]

<filename> The name of the table that you want to create to contain the structure of the current table.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if *<filename>* does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

Description COPY STRUCTURE EXTENDED copies the structure of the current table to records in a new table.

COPY STRUCTURE EXTENDED first defines a table, called a structure-extended table, containing five fields of fixed names, types, and lengths. Once the structure-extended table is defined, COPY STRUCTURE EXTENDED appends records that provide information about each field in the current table. The fields in the structure-extended table store the following information about fields in the current table:

| Field | Contents |
|------------|--|
| FIELD_NAME | Character field that contains the name of the field. |
| FIELD_TYPE | Character field that contains the field's data type. |
| FIELD_LEN | Numeric field that contains the field length. |
| FIELD_DEC | Numeric field that contains the number of decimal places for numeric fields. |
| FIELD_IDX | Character field that indicates if index tags were created on individual fields in the table. |

When the process is complete, the structure-extended table contains as many records as there are fields in the current table. You can then use CREATE...FROM to create a new table from the information provided by the structure-extended table.

No record is created in the structured-extended table for the _dbaselock field created with the CONVERT command.

OODML Use the Database object's *executeSQL()* method to call the SQL command CREATE TABLE (see CREATE STRUCTURE EXTENDED) to create the structure-extended table. Then use a loop to populate the table with information from the array of Field objects.

See Also COPY, COPY STRUCTURE, CREATE, CREATE...FROM, CREATE STRUCTURE EXTENDED, DISPLAY STRUCTURE, LIST STRUCTURE, MODIFY STRUCTURE, SET SAFETY

COPY TABLE

Makes a copy of a table.

Syntax COPY TABLE <source tablename> TO <target tablename>
[[TYPE] PARADOX | DBASE]

<source table name> The name of the table that you want to copy. You can also copy a table in a database (defined using the BDE Administrator) by specifying the database as a prefix (enclosed in colons) to the name of the table, that is, *:database name:table name*. If the database is not already open, Visual dBASE displays a dialog box in which you specify the parameters, such as a login name and password, necessary to establish a connection to that database.

<target table name> The name of the table you want to create. The table type is the same as the source table. If you copy a table in a database, you must specify the same database as the destination of the target table.

[TYPE] PARADOX | DBASE Specifies the default extension for the both the source table and target table: .DB for Paradox and .DBF for dBASE. This overrides the current setting of DBTYPE. You cannot change the table type during the copy; this clause is useful only when using filenames that do not have extensions.

Description Use the COPY TABLE command to make a copy of a table, including all its index, memo, and other associated files, if any. Unlike the COPY command, the table does not have to be open, and an exact copy of all the records is always made.

OODML Use the Database object's *copyTable()* method.

See Also COPY, COPY FILE, DELETE FILE, DELETE TABLE, ERASE

COPY TO ARRAY

Copies data from non-memo fields of the current table, overwrites elements of an existing array, and moves the record pointer to the last record copied.

Syntax COPY TO ARRAY <array>
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [FIELDS <field list>]

<array> A reference to the target array

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL, until <array> is filled.

FIELDS <field list> Copies data from the fields in <field list> in the order of <field list>. Without FIELDS, dBASE copies all the fields the array can hold in the order they occur in the current table.

Description Use COPY TO ARRAY to copy records from the current table to an existing array. COPY TO ARRAY treats the columns in a one-dimensional array like a single record of fields; and treats a two-dimensional array like a table, with the rows (the first dimension) of the array like records, and the columns (the second dimension) like fields.

To copy the fields from a single record, create a one-dimensional array the same size as the number of fields to copy. To copy all the fields in the record, use FLDCOUNT() to get the number of fields; for example

```
a = new Array( fldcount() )
```

To copy multiple records, create a two-dimensional array. The first dimension will indicate the number of records. The second dimension indicates the maximum number of fields. To copy all the records, use RECCOUNT() to get the number of records; for example

```
a = new Array( reccount(), fldcount() )
```

If the array has more columns than the table has fields, the additional elements will be left untouched. Similarly, if a two-dimensional array has more rows than the table, the additional rows are left untouched.

COPY TO ARRAY does not copy memo (or binary) fields; these fields should not be counted when sizing the target array.

COPY TO ARRAY copies records in their current order and, within each record, in field order unless you use the FIELDS option to specify the order of the fields to copy.

After copying, the record pointer is left at the last record copied, unless the array has more rows than the table has records. In this case, the record pointer is left at the end-of-file.

OODML Use two nested loops, the first to traverse the rowset, and the second to copy the *value* properties of the Field objects in the rowset's *fields* array to the target array's elements.

Example The following example uses COPY TO ARRAY and APPEND FROM ARRAY to copy records between tables where the fields are the same data type, but may not have the same field names. (If the field names were the same, the APPEND FROM command would be easier.) To minimize disk access, records are read in blocks of 100.

```
PROCEDURE AppendByPosition( cSource )
#define BLOCK_SIZE 100
local cTarget, aRec, nRecs, nCopied
*-- Get alias for current table
cTarget = alias()
use ( cSource ) in select() alias SOURCE
if reccount( "SOURCE" ) == 0
    *-- If source table is empty, do nothing
    return 0
endif
*-- Create array with default block size
aRec = new Array( BLOCK_SIZE, FLDCOUNT( "SOURCE" ) )
nCopied = 0
do while .not. eof( "SOURCE" )
    *-- Calculate number of records to copy, the smaller of
    *-- the block size and the number of records left
    nRecs = min( BLOCK_SIZE, reccount( "SOURCE" ) - nCopied )
    if nRecs < BLOCK_SIZE
        *-- Resize array for last block to copy
        aRec.resize( nRecs, FLDCOUNT( "SOURCE" ) )
    endif
endwhile
```

```

select SOURCE
*-- Copy next block
copy to array aRec rest
*-- Move from last record copied to first record in next block
skip
select ( cTarget )
append from array aRec
nCopied = nCopied + nRecs
enddo
use in SOURCE
return nCopied

```

The COPY TO ARRAY command uses the REST scope to copy the next block of records. Because the number of records to copy is known (it's calculated for the nRec variable), NEXT nRec would also work, but it's redundant, because the array has been sized to copy the right number of records. The array sizing is important because that determines the number of records that get appended with APPEND FROM ARRAY.

See Also APPEND FROM ARRAY, DECLARE, REPLACE FROM ARRAY, SET FIELDS, STORE MEMO

COUNT

Counts the number of records that match specified conditions.

Syntax COUNT
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [TO <memvar>]

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

TO <memvar> Stores the result of COUNT, a number, to the specified variable (or property).

Description Use COUNT to total the number of visible records. The current index, filter, key constraints, DELETED setting, and other factors control which records are visible at any time. You may specify further criteria with the <scope> and FOR and WHILE conditions.

If the COUNT is not stored to a memvar, the result is displayed in a dialog box. If the COUNT is stored to a memvar and SET TALK is ON, the result is also displayed in the status bar.

COUNT automatically locks the table during its operation if SET LOCK is ON (the default), and unlocks it after the count is finished. If SET LOCK is OFF, you

can still perform a count; however the result may change if another user changes the table.

You can also count the total number of records in a table using the RECCOUNT() function. However, unlike COUNT, RECCOUNT() does not let you specify conditions to qualify the records it counts.

OODML Use the Rowset object's *count*() method.

See Also AVERAGE, CALCULATE, RECCOUNT(), SUM, TOTAL

CREATE SESSION

Creates a new session—now referred to as a *workset*—and immediately selects it.

Syntax CREATE SESSION

Description Use CREATE SESSION in an application that uses form-based data handling and the Xbase DML. Applications that only use the data access objects generally do not need CREATE SESSION.

A workset is the more precise term for what was called a session in earlier versions of *Visual* dBASE and is used to encapsulate separate user tasks. It consists of the set of all 225 work areas and the current settings of most of the SET commands. There is always an active workset. When *Visual* dBASE starts, the settings are read from the VDB.INI file and all work areas are empty. This is sometimes referred to as the *startup workset*.

Whenever you open or close a table or change a setting, that occurs in the current workset. Commands that affect all work areas, like CLOSE DATABASES, affect all work areas in the current workset only. Record locks are workset-based. If a record is locked in one workset, you cannot lock that same record from another workset; but you could lock that record if the same table is open in another work area in the same workset.

When you issue CREATE SESSION, a new workset is created and made active. A new unused set of work areas is created and all settings are reread from the .INI file. Any previously existing worksets are unaffected, except that they are no longer active. In fact, you cannot change anything about a dormant workset; you must make it active first.

Whenever a form is created, it is bound to the currently active workset. Any number of forms may be bound to a single workset. Each workset has a *reference count* that indicates the number of forms bound to it. The Command window and Navigator are both bound to the startup workset.

Whenever a form receives focus or any of its methods are called, its workset is activated. This means that all commands, functions, and methods take place in the context of a specific workset and have no effect on the tables or settings in other worksets.

Note Worksets have no effect on variables.

When a form is released (either explicitly or when there are no more references to the form) its workset's reference count is reduced by one. If that reduces the reference count to zero, the workset is also released.

Whenever a workset is released, any tables that are open in it are closed automatically.

The active workset's reference count is also checked:

- Just before another workset is activated (usually by giving focus to a form in another workset)
- Whenever CREATE SESSION is executed (before the new workset is created)
- When a form method has finished executing.

If the count is zero, the active workset is released. When a form method is finished, it also selects the workset that was active when the method started. So if you click a button on a form that currently does not have focus, and that button's *onClick* event handler (all event handlers are methods) has a CREATE SESSION command then the sequence of events is as follows:

- 1 Clicking the form causes a focus change. The active workset is checked; if its reference count is zero, it is released.
- 2 The form's workset is activated.
- 3 The *onClick* executes, creating and activating a new workset.
- 4 The *onClick* ends. If the reference count of the just-created workset is zero, which it would be if the method didn't create any forms after the CREATE SESSION, it is released.
- 5 The form's workset, the one that was active when the method was executed, is reactivated. It is now the active workset.

Clicking the button again would only go through steps 3 through 5, because the form still has focus, so there is no focus change.

OODML Use Session objects.

Example The following is the *onClick* event handler for a menu item that opens a customer form:

```
function View_Customer_onClick
  create session
  do CUSTOMER.WFM
```

By using the CREATE SESSION command, each customer form operates independently, as if they were being viewed by different people on different workstations. Navigation in one form does not affect the other. A record locked in one form will be respected by another form.

This example demonstrates some of the details of CREATE SESSION. Go to the *Visual dBASE \Samples* subdirectory. Select the Tables tab of the Navigator, and type the following statements in the Command window:

```
clear all      && Release all variables and close all tables
create session && Nothing appears to happen
f = new Form()
use FISH       && FISH appears in status bar, table is italicized in Navigator
create session && The status bar is cleared
use SAMPLES    && SAMPLES appears in status bar, table is italicized in Navigator
```

This creates two worksets—call them WS1 and WS2 for reference. The order of the statements within each workset is irrelevant; they are simply executed in the currently active workset. The Fish table is the currently selected table in WS1 and the Samples table is the currently selected table in WS2. Form F is bound to WS1. WS2 has no forms bound to it, so its reference count is zero. The table names are now italic in the Navigator to indicate that they are open somewhere.

Now watch the italic Samples.dbf while you click the Navigator. Clicking the Navigator selected the startup workset. In switching worksets, the current workset, WS2, was checked. Its reference count was zero, so it was released, closing all the tables in it. Now click the Command window. This checks the Navigator's workset, the startup workset. Its reference count is at least two for both the Command window and the Navigator, so it is never released. Now type:

```
f.alias = {; ? alias() }
```

This attaches a codeblock to the form so that it becomes a method of the form. Now execute the method:

```
f.alias()      && Displays FISH
? alias()      && Blank, no table selected in startup workset
```

Executing a form's method selects the form's workset, where the Fish table is the currently selected table. After the method is complete, the previously active workset is reselected (note that there is no focus change here). Finally, watch the italic Fish.dbf in the Navigator as you execute:

```
release f
```

The variable is the only reference to the form (it's not open on-screen), so the form is destroyed, reducing WS1's reference count to zero, which releases WS1 and closes all the tables in it.

CREATE...FROM

Creates a table with the structure defined by using the COPY STRUCTURE EXTENDED or CREATE...STRUCTURE EXTENDED commands.

Syntax CREATE <filename 1>
 [[TYPE] PARADOX | DBASE]

CREATE...STRUCTURE EXTENDED

FROM *<filename 2>*
[[TYPE] PARADOX | DBASE]

<filename 1> The name of the table you want to create.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if *<filename>* does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

FROM *<filename 2>*

[TYPE] PARADOX | DBASE Identifies the table that contains the structure of the table you want to create.

Description The CREATE...FROM command is most often used with the COPY STRUCTURE EXTENDED command in a program to create a new table from another table that defines its structure, instead of using the interactive CREATE or MODIFY STRUCTURE commands. To do this, you can

- 1 Use COPY STRUCTURE EXTENDED to create a table whose records provide information on each field of the original table.
- 2 Optionally, modify the structural data in the new table with any dBASE command used to manipulate data, such as REPLACE.
- 3 Use CREATE...FROM to create a new table from the structural information in the structure extended file. The new table is active when you exit CREATE...FROM.

The table created with CREATE...FROM becomes the current table in the currently selected work area. If the CREATE...FROM operation fails for any reason, no table remains open in the current work area.

If any fields in the table created with COPY STRUCTURE EXTENDED have index flag fields set, CREATE...FROM also creates a production .MDX file with the specified index tags.

OODML No equivalent.

See Also COPY STRUCTURE, COPY STRUCTURE EXTENDED, CREATE, DISPLAY STRUCTURE, LIST STRUCTURE, MODIFY STRUCTURE

CREATE...STRUCTURE EXTENDED

Creates and opens a table that you can use to design the structure of a new table.

Syntax CREATE *<filename>* STRUCTURE EXTENDED
[[TYPE] PARADOX | DBASE]

<tablename> | ? The name of the table you want to create.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if *<filename>* does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

Description CREATE...STRUCTURE EXTENDED creates an empty table, called a *structure-extended table*, containing five fields of fixed names, types, and lengths. The fields correspond to attributes that describe fields in the table you want to create:

| Field | Contents |
|------------|--|
| FIELD_NAME | Character field that contains the name of the field. |
| FIELD_TYPE | Character field that contains the field's data type. |
| FIELD_LEN | Numeric field that contains the field length. |
| FIELD_DEC | Numeric field that contains the number of decimal places for numeric fields. |
| FIELD_IDX | Character field that indicates if index tags were created on individual fields in the table. |

The CREATE...STRUCTURE EXTENDED command is similar to the COPY STRUCTURE EXTENDED command. However, unlike COPY STRUCTURE EXTENDED, which creates a table with records providing information on fields in the current table, CREATE...STRUCTURE EXTENDED creates an empty structure-extended table. After using CREATE...STRUCTURE EXTENDED to create a new table, add records to define the structure of a new table. Then use the CREATE...FROM command to create a new table from the field definitions stored in the structure-extended table.

OODML Use the SQL command CREATE TABLE to create the STRUCTURE EXTENDED table.

See Also COPY STRUCTURE EXTENDED, CREATE, CREATE...FROM

DATABASE()

Returns the name of the current database from which tables are accessed.

Syntax DATABASE()

Description DATABASE() returns the name of the current default database selected with the SET DATABASE command. If no database is open, the DATABASE() function returns an empty string ("").

Note: Databases are defined with the BDE Administrator.

OODML Check the Database object's *databaseName* property.

See Also CLOSE..., OPEN DATABASE, SET DATABASE, SET DBTYPE

DBF()

Returns the name of a table open in the current or a specified work area.

Syntax DBF([<alias>])

<alias> The work area to check.

Description DBF() returns the name of the table open in a specified work area. If the table is a file on disk, as it is with DBF and DB tables, the filename includes the extension and the drive letter. If SET FULLPATH is ON, the DBF() function also returns the directory location of the table in addition to the table name.

If no table is in use in the current or specified work area, DBF() returns an empty string ("").

OODML There is no concept of the "current" Query object. In most cases, you can ascertain the name of the table by parsing the SQL SELECT statement in the Query object's *sql* property.

See Also ALIAS(), MDX(), NDX(), SET FULLPATH, TAG(), WORKAREA(), USE

DELETE

Deletes records from the current table.

Syntax DELETE
[<scope>]
[FOR <condition 1>]
[WHILE <condition 2>]

<scope>
FOR <condition 1>
WHILE <condition 2> The scope of the command. The default scope is NEXT 1, the current record only.

Description DBF tables support the concept of *soft deletes*, where the record is marked as deleted and normally hidden when SET DELETED is ON (the default). If you SET DELETED OFF, you can see the deleted records along with the records that are not marked as deleted. You can RECALL the record to undelete it. To actually remove the record from the table, you must PACK the table. If you use the LIST or DISPLAY commands to display records, records marked as deleted are displayed with an asterisk.

For other table types, when you delete a record, it is removed from the table and cannot be recovered. (Some tables still require you to perform a

maintenance operation on the table to reclaim the unused space. For more information, refer to your database server documentation.)

Relying on soft deletes to be able to recover information from deleted records is not recommended. This technique does not scale well to other databases, because they don't support soft deletes. If you want to make data available for recover, consider using an identically-structured purge table that stores copies of the records that you have deleted.

Soft deletes are useful when you want to recycle deleted records. This obviates the need to PACK the table. You BLANK the record before you DELETE it. Then whenever you need to add a new record, you can search for a deleted record and reuse it.

To delete all records from a table, use ZAP.

OODML Use the Rowset object's *delete()* method. There is no support for soft deletes; if you *delete()* a row in a DBF table, there is no corresponding method to recall it. You may still use the RECALL command.

Example The following function makes a copy of the record to be deleted from the table named Main in the purge table named Purge:

```
PROCEDURE DelMainRec
  store automem
  select PURGE
  append automem
  select MAIN
  blank
  delete
```

See Also PACK, RECALL, SET DELETED, ZAP

DELETE TABLE

Deletes a specified table.

Syntax DELETE TABLE <filename> [[TYPE] PARADOX | DBASE]

<filename> The name of the table that you want to delete.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if <filename> does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

Description Use the DELETE TABLE command to delete a table and its index, memo, and other associated files. Make sure the table is not in use before you attempt to delete it.

OODML Use the Database object's *dropTable()* method.

See Also DELETE FILE, DELETE TAG, ERASE

DELETE TAG

Deletes index tags from tables.

Syntax DELETE TAG <tag name 1>
 [OF <filename 1>
 [, <tag name 2>
 [OF <filename 2>]...]

<tag name 1>, <tag name 2>, ... <tag name n> The index tag names to delete.

OF <filename 1> | ? | <filename skeleton 1> For DBF tables, specifies the .MDX file containing the tag name to delete. If you specify a file without including an extension, *Visual* dBASE assumes an .MDX extension. If you don't specify an index file, *Visual* dBASE assumes the index tag you want to delete is in the index file with the same name as the current table.

Description Use DELETE TAG to delete index tags from .MDX files for dBASE tables or secondary indexes on a Paradox table. *Visual* dBASE allows a maximum of 47 index tags in a single .MDX file, so deleting unneeded tags frees slots for new tags as well as reducing the amount of disk space and memory that an .MDX file requires.

For dBASE tables, the .MDX file must be open when you delete the tags. If you delete all tags in an .MDX file, the .MDX file is also deleted. If you delete the production .MDX file by deleting all index tags, the table file header is updated to indicate there is no longer a production index associated with the table.

The table associated with the indexes you want to delete must be opened in exclusive mode. When accessing a Paradox table, specifying DELETE TAG without an argument deletes the primary index.

OODML Use the Database object's *dropIndex()* method.

Example The following function deletes all the tags in the current table, which you would do before rebuilding all the tags from scratch.

```
PROCEDURE ZapTags
do while "" # tag( 1 )
  delete tag tag( 1 )
enddo
```

See Also CLOSE INDEXES, COPY INDEXES, SET INDEX, TAG()

DELETED()

Indicates if the current record is marked as deleted.

Syntax DELETED([<*alias*>])
<*alias*> A work area to check.

Description DELETED() returns *true* if the current record in the specified work area is marked as deleted otherwise, DELETED() returns *false*.
 If no table is open in the current or specified work area, DELETED() also returns *false*.

OODML No support for soft deletes.

See Also DELETE, PACK, RECALL, SET DELETED

DESCENDING()

Indicates if a specified index is in descending order.

Syntax DESCENDING([<*.mdx filename expC*>] [<*index position expN*> [<*alias*>]])
<*.mdx filename expC*> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.
<*index position expN*> the numeric position of the index tag in the specified .MDX file, or the position of the index in the list of open indexes.
<*alias*> The work area you want to check.

Note Unlike most functions, the first parameter is optional. If you omit the first parameter, the remaining parameters shift forward one; the second parameter becomes the first parameter, and so on.

Description DESCENDING() returns *true* if the index tag specified by the <*index position expN*> parameter was created with the DESCENDING keyword; otherwise, it returns *false*.

The index must be referenced by number. If you do not specify an .MDX file to check, index numbering in the list of open indexes is complicated if you have open .NDX indexes or you have open non-production .MDX files. For more information on index numbering, see SET INDEX. Either way, it is often easier to reference an index tag by name by using the TAGNO() function to get the corresponding position number.

If you do not specify an index tag, DESCENDING() checks the current master index tag and returns *false* if the master index is an .NDX file or there is no master index.

If the specified .MDX file or index tag does not exist, DESCENDING() returns *false*.

OODML No equivalent

Example See MDX().

See Also FOR(), INDEX, KEY(), MDX(), ORDER(), TAGCOUNT(), TAGNO(), UNIQUE()

DISPLAY

Displays records from the current table in the result pane of the Command window.

Syntax DISPLAY
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [[FIELDS] <exp list>]
 [OFF]
 [TO FILE <filename>]
 [TO PRINTER]

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is NEXT 1, the current record only.

FIELDS <exp list> Field names or expressions whose contents (values) you want to display; the names of the fields in the list are separated by commas. If you omit <exp list>, *Visual* dBASE displays all fields in the current table. The FIELDS keyword is included for readability only; it has no affect on the operation of the command.

OFF Suppresses display of the record number when displaying records from a DBF table.

TO FILE <filename> Directs output to a file, as well as to the results pane of the Command window. By default, *Visual* dBASE assigns a .TXT extension to <filename>.

TO PRINTER Directs output to the default printer, as well as to the results pane of the Command window.

Description Use DISPLAY to view one or more records of the current table in the results pane of the Command window. If SET HEADINGS is OFF, *Visual* dBASE doesn't display field names when you issue DISPLAY. DISPLAY pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information.

Use the TO FILE clause to send the information to a file. Use the TO PRINTER clause to send the information to the printer. In either case, you can use SET CONSOLE OFF to suppress the display of the information in the results pane.

The LIST command is almost identical to DISPLAY, except that:

- The default scope for LIST is ALL.
- LIST doesn't pause for each screenful of information but rather lists the information continuously. This makes LIST more appropriate when directing output to a file or printer.

Memo fields are displayed as "MEMO" if they contain data or "memo" if they are empty; unless the field is listed in *<exp list>*, in which case the contents of the memo field is displayed.

OODML No equivalent

See Also LIST, SET CONSOLE, SET FIELDS, SET HEADINGS

EDIT

Displays fields in the current table for editing.

Syntax EDIT
[COLUMNAR]
[FIELDS *<field list>*]

COLUMNAR Creates a form with the field names in one column and the field controls in another column.

[FIELDS *<field list>*] Displays the fields specified in *<field list>* for editing. Field names are separated by commas. The field list may include calculated fields in the format:

<calculated field name> = <expression>

Description EDIT displays the current record in the current table in a wizard-generated form for editing.

OODML Use a form.

See Also APPEND, BROWSE

EOF()

Indicates if the record pointer is at the end-of-file.

Syntax EOF([*<alias>*])
<alias> The work area to check.

Description EOF() returns *true* when the record pointer in the current or specified work area is positioned past the last record; otherwise it returns *false*. If you attempt to navigate forward when EOF() is *true*, an error occurs.

FDECIMAL()

When you first USE a table, EOF() is *true* if the table is empty, or you are using a conditional index with no matching records.

Many operations leave the record pointer at the end-of-file when they are complete or when they fail. For example, EOF() returns *true* after SCAN processes the last record in a table, when you use SKIP to pass the last record in a table, when you use LIST with no options, or when SEEK() or SEEK fails to find the specified record (and SET NEAR is OFF).

The position at the end-of-file is sometimes referred to as the *phantom record*. When you get the values of the fields at the phantom record, they are always blank. Attempting to REPLACE field values in the phantom record causes an error.

If no table is open in the specified work area, EOF() returns *false*.

OODML The Rowset object's *endOfSet* property is *true* when the row pointer is past either end of the rowset. Unlike BOF() and EOF(), there is symmetry with the *endOfSet* property. You can determine which end you're on based on the direction of the last navigation.

There is also an *atLast*() method that determines whether you are on the last row in the rowset, the row before EOF().

Example The following is an event handler for a button that navigates forward through a table:

```
PROCEDURE nextButton_onClick
  if .not. eof()
    skip
  endif
  if eof()
    msgbox( "Last record", "Navigation", 64 )
  endif
```

This example demonstrates an atypical programming construct: instead of using IF and ELSE, there is an IF statement for a condition, followed by a separate IF statement for the opposite condition. In this case, it works like this: if you are already at EOF() you do not want to attempt to navigate forward, because that will cause an error. But if you end up at EOF(), or if you're already at EOF(), then you will get a message.

Many processes require traversing the entire table. The SCAN loop is designed to visit each record automatically, but sometimes you may want to manually code a loop and check for EOF() to see when you are done. For an example of this, see the example for COPY TO ARRAY.

See Also BOF(), FIND, FOUND(), LOCATE, RECNO(), SEEK, SEEK()

FDECIMAL()

Returns the number of decimal places in a specified field of a table.

Syntax FDECIMAL(<field number expN> [, <alias>])

<field number expN> The position of the field that you want to evaluate. The first field in a table is field number 1.

<alias> The work area that contains the field to check.

Description FDECIMAL() returns the number of decimal places in a specified field of a table. FDECIMAL() returns zero if the field has no decimal places, if the field is not a numeric field, or if the table doesn't contain a field in the specified position.

OODML Check the *decimalLength* property of the Field object.

See Also FIELD(), FLENGTH()

FIELD()

Returns the name of the field in a specified position of a table.

Syntax FIELD(<field number expN> [, <alias>])

<field number expN> The position of the field whose name you want returned. The first field in a table is field number 1.

<alias> The work area to check.

Description FIELD() returns the name of a field in a table based on the specified <field number expN> parameter. The example shows a function that performs the reverse operation, returning the field number for a specified field name. If the field name has spaces, FIELD() returns the name enclosed in colons, for example:

```
:Primary power coupling:
```

FIELD() returns an empty string ("") if the table does not contain a field in the specified position.

OODML Check the *fieldName* property of the Field object.

Example The following uses the FIELD() function to perform the reverse operation: return the number of a field with the given name.

```
PROCEDURE FieldNum( cName, xAlias )
  local nWork, nFld
  if argcount() < 2
    xAlias = workarea()
  endif
  for nFld = 1 to fldcount( xAlias )
    if upper( cName ) == upper( field( nFld, xAlias ))
      return nFld
    
```

FLDCOUNT()

```
endif  
endfor  
return 0
```

This function takes an optional alias parameter, just like the FIELD() function. If the alias is not specified, the current work area number is used.

The names are converted to uppercase for comparison, so the field name specified does not have to match the case of the field in the table.

See Also DBF(), FLENGTH()

FLDCOUNT()

Returns the number of fields in a table.

Syntax FLDCOUNT([<alias>])

<alias> The work area you want to check.

Description FLDCOUNT() returns the number of fields for the table opened in the current or specified work area. FLDCOUNT() returns a value of 0 if no table is open in that work area.

OODML Check the *size* property of Rowset object's *fields* array.

See Also FIELD(), DISPLAY STRUCTURE, LIST STRUCTURE, RECCOUNT(), TYPE()

FLDLIST()

Returns the fields and calculated field expressions of a SET FIELDS TO list.

Syntax FLDLIST([<field number expN>])

<field number expN> The position of the field or calculated field expression in a SET FIELDS TO list whose name you want returned. If you do not specify a field number, FLDLIST() returns the entire field list.

Description FLDLIST() returns the field or calculated field expression in a SET FIELDS TO list that corresponds to a specified field number. If you do not specify a field number, FLDLIST() returns the entire field list. Each field name or expression in the field list is separated by a comma. FLDLIST() always returns fully-qualified field names, that is, it includes the table or alias name. For read-only fields, FLDLIST() appends "/R" to the field name.

FLDLIST() returns the field list even if SET FIELDS is OFF. If there is no SET FIELDS TO list, or the specified field number exceeds the number of items in the field list, FLDLIST() returns an empty string ("").

OODML Check the *fieldName* property of the Field object for a normal field. A calculated field is defined by either its *value* property, or by its *beforeGetValue* event.

See Also SET FIELDS

FLENGTH()

Returns the length of the field in a specified position of a table.

Syntax FLENGTH(<field number expN> [, <alias>])

<field number expN> The position of the field whose length you want returned. The first field in a table is field number 1.

<alias> The work area you want to check.

Description FLENGTH() returns the length of a field in a table based on the specified <field number expN> parameter. The field length for numeric fields includes the decimal digits and one for the decimal point character. Certain field types have fixed lengths. For example, in a DBF table, FLENGTH() returns 8 for date fields and 10 for memo fields.

FLENGTH() returns 0 if the table does not contain a field in the specified position.

OODML Check the *length* property of the Field object.

Example The following routine is used to read the data in a generated text file into the corresponding fields of a table. Character fields in the text file are the same length as in the table. Dates are formatted in six characters as MMDDYY (which matches the current SET DATE format). Numbers are always twelve characters and represent currency stored in cents, so it needs to be divided by 100.

```
function decodeLine( cLine, aDest )
#define YEAR_LEN 2
#define NUM_LEN 12
local nPtr, nFld, cFld, nLen
nPtr = 1                && Pointer into string
for nFld = 1 to fldcount()
  cFld = field( nFld )    && Store name of field in string variable for reuse
do case
  case type( cFld ) == "C"
    aDest[ nFld ] = substr( cLine, nPtr, flength( nFld ) )
    nPtr += flength( nFld )
  case type( cFld ) == "D"
    aDest[ nFld ] = ctod( substr( cLine, nPtr, 2 ) + "/" + ;
                          substr( cLine, nPtr + 2, 2 ) + "/" + ;
                          substr( cLine, nPtr + 4, YEAR_LEN ) )
    nPtr += 2 + 2 + YEAR_LEN
  case type( cFld ) == "N"
    aDest[ nFld ] = val( substr( cLine, nPtr, NUM_LEN ) ) / 100
```

```

        nPtr += NUM_LEN
    endcase
endfor

```

An array is passed to the routine along with the line to read. The field values are stored in the array, which is appended to the table with APPEND FROM ARRAY in the calling routine (not shown here). The function defines some manifest constants for the size of a numeric field and whether the year is two or four digits in case this changes in the future. A FOR loop goes through each field in the table. The name of each field is stored in a variable for convenience; it's used repeatedly in the DO CASE structure.

See Also FDECIMAL(), FIELD()

FLOCK()

Locks a table.

Syntax FLOCK([<alias>])
<alias> The work area you want to lock.

Description Use FLOCK() to lock the table in the current work area, or in another specified work area, preventing others from using the table.

When you lock a table with FLOCK(), only you can make changes to it. However, unlike USE...EXCLUSIVE and SET EXCLUSIVE ON, FLOCK() lets other users view the locked table while you are using it. When you lock a table with FLOCK(), it remains locked until you issue UNLOCK or close the table.

FLOCK() is similar to RLOCK(), except that FLOCK() locks an entire table, while RLOCK() lets you lock specific records of a table. Use FLOCK(), therefore, when you need to have sole access to an entire table or related tables—for example, when you need to update multiple tables related by a common key.

FLOCK() can lock a table even if another user is viewing data contained in the table. FLOCK() is unsuccessful only if another user has explicitly locked the table or a record in the table, or is using a command that automatically locks the table or a record in the table. FLOCK() returns *true* if it is successful, and *false* if it is not.

All commands that change table data cause *Visual* dBASE to attempt an automatic record or file lock. If *Visual* dBASE fails to get an automatic record or file lock, it an error occurs. You might want to use FLOCK() to handle a lock failure yourself, instead of letting the error occur.

When SET REPROCESS is set to 0 (the default) and FLOCK() can't immediately lock a table, *Visual* dBASE prompts you to attempt the lock again or cancel the attempt. Until you choose to cancel the function, FLOCK() repeatedly attempts

to lock the table. Use SET REPROCESS to eliminate being prompted to cancel the FLOCK() function, or to set the number of locking attempts.

When you set a relation to a parent table with SET RELATION and then lock the table with FLOCK(), *Visual* dBASE attempts to lock all child tables. For more information about relating tables, see SET RELATION.

OODML Use the Rowset object's *lockSet()* method.

See Also BEGINTRANS(), LOCK(), RLOCK(), SET EXCLUSIVE, SET LOCK, SET RELATION, SET REPROCESS, UNLOCK, USE

FLUSH

Writes data buffers for the current work area to disk.

Syntax FLUSH

Description Use FLUSH to protect data integrity.

When you open a table, *Visual* dBASE loads a certain number of records from that table into a memory buffer, along with the portion of each open index that pertains to those records. When another block of records needs to be read or when you close tables, *Visual* dBASE writes the records in the buffer back to disk, storing any modifications you have made.

FLUSH allows you to save information from the data buffer to disk on-demand, without closing the table. Use FLUSH when you need to store critical information to disk that could otherwise be lost. However, don't use FLUSH too frequently, as it slows execution. For example, in an order-entry application in which only a few orders are entered each hour, FLUSH can save data that might be lost if the power is inadvertently turned off; since orders are entered infrequently, the time needed to execute FLUSH is not important.

OODML Use the Rowset object's *flush()* method.

See Also CLOSE TABLES, SET AUTOSAVE

FOR()

Returns the FOR clause of a specified index tag.

Syntax FOR([<.mdx filename expC>.] [<index position expN> [,<alias>]])

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.

<index position expN> the numeric position of the index tag in the specified .MDX file, or the position of the index in the list of open indexes.

<alias> The work area you want to check.

Note Unlike most functions, the first parameter is optional. If you omit the first parameter, the remaining parameters shift forward one; the second parameter becomes the first parameter, and so on.

Description FOR() returns a string containing the FOR expression of the specified .MDX tag. FOR() returns an empty string ("") if the specified index tag does not have a FOR expression.

The index must be referenced by number. If you do not specify an .MDX file to check, index numbering in the list of open indexes is complicated if you have open .NDX indexes or you have open non-production .MDX files. For more information on index numbering, see SET INDEX. Either way, it is often easier to reference an index tag by name by using the TAGNO() function to get the corresponding position number.

If you do not specify an index tag, FOR() checks the current master index tag and returns an empty string if the master index is an .NDX file or there is no master index.

If the specified .MDX file or index tag does not exist, FOR() returns an empty string.

OODML No equivalent.

Example See MDX().

See Also INDEX, DESCENDING(), TAG(), TAGCOUNT(), TAGNO(), UNIQUE(), USE

FOUND()

Indicates if the last-issued search command found a match.

Syntax FOUND([<alias>])

<alias> The work area you want to check.

Description FOUND() returns *true* if LOCATE, CONTINUE, SEEK, LOOKUP(), or SEEK() found a match in the current or specified table. FOUND() returns *false* if no previous search has been performed in that work area, or if the last search was unsuccessful. You can perform searches in different work areas and maintain the status of each FOUND() operation, independent of the other work areas.

If tables are linked by a SET RELATION TO command, Visual dBASE searches the related tables as you move in the active table with normal navigation or

with a search command. This allows you to determine if there is a match in related tables.

When SET NEAR is ON and you use SEEK or SEEK(),

- FOUND() returns *true* if an exact match occurs.
- FOUND() returns *false* for a near match, and the record pointer is moved to the record whose key immediately follows the value searched for.

When SET NEAR is OFF, FOUND() returns *false* if a match does not occur.

OODML Check the return value of the Rowset object's *findKey()* or *findKeyNearest()* method.

Example The following statements create a relation between a table of customers and orders, and use FOUND() to show only those customers that have orders:

```
use CUSTOMER
use ORDERS in select() order CUST_DATE
set relation to CUST_ID into ORDERS
set filter to found( "ORDERS" )
```

See Also CONTINUE, EOF(), LOCATE, LOOKUP(), SEEK, SEEK(), SET NEAR, SET RELATION

GENERATE

Adds random records to the current table.

Syntax GENERATE [<expN>]

<expN> A number of random-data records to add to the current table. If you specify a <expN> value that is less than or equal to 0, no records are generated. If you don't specify a value for <expN>, *Visual dBASE* prompts you for a number.

Description GENERATE fills a table with sample data. If a table contains existing records, GENERATE leaves them intact and adds <expN> records to the table.

GENERATE does not create data for memo or binary fields.

OODML No equivalent.

See Also none

GO

Moves the record pointer to the specified position in a table.

Syntax GO[TO]
BOTTOM | TOP | <bookmark> | [RECORD] <expN>
[IN <alias>]

TO Include for readability only; you may use GO or GOTO.

BOTTOM | TOP | <bookmark> | [RECORD] <expN> Specifies where to move the record pointer. The following table describes each of the available keywords or options.

| Option | Moves the record pointer to |
|-----------------|--|
| BOTTOM | The last record in the table, using the current index order, if any. |
| TOP | The first record in the table, using the current index order, if any |
| <bookmark> | The record saved in <bookmark> |
| [RECORD] <expN> | That record number. Entering a number in the Command window is equivalent to GO <expN>. The RECORD keyword is included for readability only; it has no affect on the operation of the command. |

IN <alias> The work area where you want to move the record pointer.

Description GO positions the record pointer in a table.

GO <expN> or GO RECORD <expN> moves the record pointer to a specific record, regardless of whether a master index is open or where that record number occurs in an indexed order. It works only for DBF tables. For tables that do not support record numbers (that is, Paradox and SQL tables), GO <expN> causes an error.

To go to a specific record, use the BOOKMARK() to get a bookmark for that record and store it in a variable or property. Then when you need to go back to that record, issue GO <bookmark>.

If an index isn't in use, TOP and BOTTOM refer to the first and last records in a table. If an index is in use for a table, TOP and BOTTOM refer to the first and last records in the index order.

If a relation is set up among several tables, moving the record pointer in the parent table with GOTO repositions the record pointer in a child table to a related record. If there is no related record, the child table record pointer is positioned at the end of the file. Moving the record pointer in a child table, however, doesn't reposition the record pointer in the parent table.

OODML Use the Rowset object's first(), last(), and goto() methods.

See Also BOOKMARK(), EOF(), RECNO(), SET DELETED, SET FILTER, SET RELATION, SKIP

INDEX

Creates an index for the current table.

Syntax For DBF tables:

```

INDEX ON <key exp>
TAG <tag name>
    [OF <.mdx filename>]
[FOR <condition>]
[DESCENDING]
[UNIQUE | DISTINCT | PRIMARY ]

```

or to create dBASE III-compatible .NDX index files:

```
INDEX ON <key exp> TO <.ndx filename> [UNIQUE]
```

For DB and SQL tables:

```

INDEX ON <field list>
PRIMARY | TAG <tag name> [UNIQUE]

```

<key exp> For DBF tables, <key exp> can be a dBASE expression of up to 220 characters that includes field names, operators, or functions. The maximum length of the key—the result of the evaluated index <key exp>—is 100 characters.

<field list> For Paradox and SQL tables, indexes can't include expressions; however, you can create indexes based on one or more fields. In that case, you specify the index key as a <field list>, separating the name of each field with a comma.

TAG <tag name> Specifies the name of the index tag for the index

OF <.mdx filename> Specifies the .MDX multiple index file that *Visual* dBASE adds new index tags to. If you do not specify an .MDX file, index tags are added to the production .MDX file. If you specify a file that doesn't exist, *Visual* dBASE creates it and adds the index tag name. By default, *Visual* dBASE assigns an .MDX extension and saves the file in the current directory.

TO <.ndx filename> Specifies the name of an .NDX index file.

FOR <condition> Restricts the records *Visual* dBASE includes in the index to those meeting the specified <condition>.

DESCENDING Creates the index in descending order (Z to A, 9 to 1, later dates to earlier dates). Without DESCENDING, INDEX creates an index in ascending order.

UNIQUE For DBF tables, prevents multiple records with the same <key exp> value from being included in the index; *Visual* dBASE includes in the index only the first record with that value. For DB and SQL tables, specifies creating a distinct index which prevents entry of duplicate index keys in a table.

DISTINCT Prevents multiple records with the same <key exp> value from being included in the table; any such attempt causes a key violation error. Records marked as deleted are never included in a DISTINCT index. DISTINCT indexes may be created for DBF tables only.

PRIMARY Specifies that the index is the primary key for the table. For DBF tables, the PRIMARY index is a distinct index that is designated as the primary index; it currently has no other special meaning. For DB and SQL tables, the primary key has a specific meaning. A table may have only one primary key.

Description Use INDEX to organize data for rapid retrieval and ordered display. INDEX doesn't actually change the order of the records in a table but rather creates an index in which records are arranged in numeric, alphabetical, or date order based on the value of a key expression. Like the index of a book, with ordered entries and corresponding page numbers, an index file contains ordered key expressions with corresponding record numbers. When the table is used with an index, the contents of the table appear in the order specified by the index.

DBF expression indexes To index on multiple fields in a DBF table, you must create an expression index. When combining fields with different data types, use conversion functions to convert all the fields to the same data type. Most multi-field expression indexes are character type; numeric and date fields are converted to strings using the STR() and DTOS() functions. When using the STR() function, be sure to specify the length of the resulting string so that it matches the numeric field.

Note Do not use the DTOC() function to convert a date to a string. In many date formats, the day comes before the month, or the month comes before the day and year, resulting in records in the wrong order.

To concatenate the fields, use the + or - operators.

Warning Do not create an index where the length of the index key expression varies from record to record. Specifically, do not use TRIM() or LTRIM() to remove blanks from strings unless you compensate by adding enough spaces to make sure the index key values are all the same length. The - operator concatenates strings while rearranging trailing blanks. Varied key lengths may cause corrupted indexes.

If a function is used in a key expression, keep in mind that the index is ordered according to the function output. Thus, when you use search for a particular key, you must search for the key expression as it was generated. For example, INDEX ON SOUNDEX(Name) TO Names creates an index ordered by the values SOUNDEX() returns. When attempting to find data by the key value, you would have to use something like SEEK SOUNDEX("Jones") rather than SEEK "Jones".

FOR <condition> limits the records that are included in the index to those meeting the specified condition. For example, if you use INDEX ON Lastname + Firstname TO Salaried FOR Salary > 24000, Visual dBASE includes only records of employees with salaries higher than \$24,000 in the index. The FOR condition can't include calculated fields.

The following built-in functions may be used in the index *<key exp>* and FOR *<condition>* expressions of a DBF index tag.

Table 12.1 List of functions supported in DBF expression indexes

| | | | | |
|-------------|--------------|-----------|-------------|-----------|
| ABS() | CHR() | FLOOR() | MEMLINES() | RTOD() |
| ACOS() | COS() | FV() | MIN() | RTRIM() |
| ANSI() | CTOD() | HTOI() | MLINE() | SECONDS() |
| ASC() | DATABASE() | ID() | MOD() | SIGN() |
| ASIN() | DATE() | INT() | MONTH() | SIN() |
| AT() | DAY() | ISALPHA() | OEM() | SOUNDEX() |
| ATAN() | DBF() | ISBLANK() | OS() | SPACE() |
| ATN2() | DELETED() | ISLOWER() | PAYMENT() | SQRT() |
| BITAND() | DIFFERENCE() | ISUPPER() | PI() | STR() |
| BITLSHIFT() | DOW() | ITOH() | PROPER() | STUFF() |
| BITNOT() | DTOC() | LEFT() | PV() | SUBSTR() |
| BITOR() | DTOR() | LEN() | RAND() | TAN() |
| BITRSHIFT() | DTOS() | LIKE() | RAT() | TIME() |
| BITSET() | ELAPSED() | LOG() | RECNO() | TRIM() |
| BITXOR() | EMPTY() | LOG10() | RECSIZE() | UPPER() |
| BITZSHIFT() | EXP() | LOWER() | REPLICATE() | VAL() |
| CEILING() | FCOUNT() | LTRIM() | RIGHT() | VERSION() |
| CENTER() | FIELD() | MAX() | ROUND() | YEAR() |

Index sort order In an index, records are usually arranged in ascending order, with lowest key values at the beginning of the index. Using the DOS Code Page 437 (U.S.) character set, character keys are ordered in ASCII order (from A to Z and then from a to z); numeric keys are ordered from lowest to highest numbers; and date keys are ordered from earliest to latest date (a blank date is higher than all other dates). Use the UPPER() function on the key expression to convert all lowercase letters to uppercase and achieve alphabetical order for character-type indexes.

Note Most non-U.S. character sets provide a different sort order for characters than the DOS Code Page 437 character set.

You can reverse the order of an index, arranging records in descending order, by including the DESCENDING keyword. (You can use DESCENDING only when building .MDX tags.)

Distinct, primary, and unique indexes You may use an index to ensure that there are no duplicate key values. For example, in a table of customers, each customer is assigned their own unique customer ID number. To prevent an existing customer ID number from being used by another customer, you can create a special kind of index on the customer ID field. For DB and SQL tables, this type of index is called a *unique* index; the key value for each record in the table must be unique. For DBF tables, this type of index is called a *distinct* index; a unique index for a DBF table has a different meaning. For clarity, the DBF terms are used.

A distinct index is created with the `DISTINCT` option for DBF tables, and the `UNIQUE` option for DB and SQL tables. When a table has a distinct index, any attempt to create a duplicate key entry, either by adding a new record with a duplicate value or by changing an existing record so that its key field(s) duplicates another record, causes a key violation error. The new or changed record is not written to the table. Distinct indexes for DBF tables never include records that are marked as deleted.

A table may also have one distinct index designated as its *primary* index, or *primary key*. A primary index is usually created for the ID field or fields that uniquely identify each record in the table. For example, while you may index on the customer's name, their ID field is what uniquely identifies each customer, and that is the field you use for the primary key. For DB tables, a table's primary key determines the default order for the records in the table, and you must have a primary key to create other secondary indexes. For DBF tables, a primary key currently has no special meaning, other than self-documenting the primary key field(s) of the table. The `PRIMARY` clause is used to create the primary index. For DB and SQL tables, a primary index may have no other options other than the field list.

DBF tables support a kind of index that allows duplicate key values in the table, but only shows the first such record in the index. These are called unique indexes, not to be confused with the distinct unique indexes used by DB and SQL tables. For example, you may be interested in the names of the cities in which your customers reside. By using a unique index, each city is listed once (alphabetically), no matter how many customers you have in that city.

A record's index key value is tested for uniqueness only when the record is added or updated. For example, suppose you have a unique index on the City field, and have records in both "Bismark" and "Fargo". If you append another record in "Bismark", it does not appear in the index, although the table is updated with the new record. If you then change the first record, which was listed in the index, from "Bismark" to "Fargo", then it too becomes hidden because there is already a "Fargo" in the index. It also does not automatically expose the other record with "Bismark", because that record was not updated; no records in "Bismark" are in the index at that moment. `REINDEX` explicitly updates all key values in a unique index.

Indexing a table with `SET UNIQUE ON` has the same effect as `INDEX` with the `UNIQUE` option. With DB and SQL tables, it creates a distinct index. With DBF tables, it creates a unique index.

Using indexes Once a table has been indexed, use `LOOKUP()`, `SEEK`, and `SEEK()` to retrieve data. The structure of an index file allows these commands to quickly locate values of the key expression.

Whenever data in key fields is modified, *Visual* dBASE automatically updates all open index files. Index files closed when changes are made in a table can be opened and then updated using `REINDEX`.

Multiple index files simplify updating indexes, since *Visual* dBASE updates all indexes with tag names listed in .MDX files specified with `USE...ORDER` or `SET`

ORDER. *Visual* dBASE automatically opens a production .MDX file, if one exists, when you use the associated table.

INDEX...TAG creates an index and adds the tag name to a multiple index file. If you don't include OF <filename>, INDEX...TAG adds the tag name to the production .MDX file. *Visual* dBASE creates the production .MDX or the specified file if it doesn't already exist.

INDEX is similar to SORT, another command that allows ordering of a table. Unlike INDEX, though, SORT physically rearranges the table records, a time-consuming process for large files. To maintain the sorted order, either new records must be inserted in their proper position, which is also very time-consuming, or the entire table must be resorted. Also, SORT doesn't support LOOKUP(), SEEK, or SEEK(), making the process of locating data in a sorted table much slower.

At the end of an indexing operation, the new index file is the master index, and the record pointer is positioned at the first record of the new indexed.

OODML Use the Database object's *createIndex()* method.

Example The following example creates an index based on the last name and first name in a table:

```
index on upper( LAST_NAME + FIRST_NAME ) tag FULL_NAME
```

The next example indexes on a customer ID and the order date as the primary key for an Orders table:

```
index on CUST_ID + dtos( ORDER_DATE ) tag CUST_ORD primary
```

See Also FIND, KEY(), LOOKUP(), ORDER(), REINDEX, SEEK, SEEK(), SET INDEX, SET ORDER, SET UNIQUE, SORT, TAG(), USE

ISBLANK()

Determines if a specified field or expression is blank.

Syntax ISBLANK(<exp>)

<exp> An expression of any data type.

Description ISBLANK() returns *true* if a specified expression is blank or *null*; *false* if it contains data. A field is blank if it has never contained a value or if you used the BLANK command on it. ISBLANK() returns a different result from EMPTY() when used on numeric fields; ISBLANK() differentiates between zero and blank values, while EMPTY() does not.

ISBLANK() is especially useful when performing functions such as averaging, since it ensures that blank values are not included in the calculation. If you don't need to differentiate between 0 or blank values in numeric fields, you can use either ISBLANK() or EMPTY().

ISTABLE()

OODML No equivalent.

See Also APPEND, BLANK, EMPTY(), SPACE(), TYPE()

ISTABLE()

Tests for the existence of a table in a specified database.

Syntax ISTABLE(<filename>)

<filename> The name of the table to search for. You cannot use a filename skeleton for ISTABLE(). If you do, it will return *false*.

Description Use ISTABLE() to confirm the existence of a table. If the table exists, ISTABLE() returns *true*; otherwise it returns *false*.

OODML Use the Database object's *tableExists()* method.

See Also DIR, DISPLAY FILES, FILE(), GETFILE(), PUTFILE(), SET DEFAULT, SET DATABASE, SET DBTYPE, SET DIRECTORY, SET PATH

KEY()

Returns the key expression of the specified index.

Syntax KEY([<.mdx filename>.] [<index position expN> [,<alias>]])

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.

<index position expN> the numeric position of the index tag in the specified .MDX file, or the position of the index in the list of open indexes.

<alias> The work area you want to check.

Note Unlike most functions, the first parameter is optional. If you omit the first parameter, the remaining parameters shift forward one; the second parameter becomes the first parameter, and so on.

Description KEY() returns a string containing the key expression of the specified index. To see the value of the key expression for a given record, store the string returned by KEY() in a private variable. Then use macro substitution to evaluate the expression.

The index must be referenced by number. If you do not specify an .MDX file to check, index numbering in the list of open indexes is complicated if you have open .NDX indexes or you have open non-production .MDX files. For more information on index numbering, see SET INDEX. Either way, it is often easier

to reference an index tag by name by using the TAGNO() function to get the corresponding position number.

If you do not specify an index tag, KEY() checks the current master index tag and returns an empty string if there is no master index.

If the specified .MDX file or index tag does not exist, KEY() returns an empty string.

OODML No equivalent.

Example The following example displays the current index key value during navigation for debugging purposes:

```
PROCEDURE Form_onNavigate
  private cKey
  cKey = key()
  ? "Key value: [" + &cKey + "]"
```

See Also INDEX, NDX(), ORDER(), SET INDEX, SET ORDER, TAG(), TAGCOUNT(), TAGNO(), USE

KEYMATCH()

Indicates if a specified expression is found in an index.

Syntax KEYMATCH (<exp> [,<index number> [,<alias>]])

where <index number> is:

<index position expN> | [<.mdx filename expC>,<tag expN>

<exp list> The expression, of the same data type as the index, that you want to look for.

<index position expN> The numeric position of the index in the list of open indexes for the table.

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area.

<tag expN> The numeric position of the index tag in the specified .MDX file.

<alias> The work area you want to check.

Description The KEYMATCH() function determines if a specified key expression is found in a particular index. KEYMATCH() returns *true* or *false* to indicate whether the specified expression was found. SET EXACT controls whether exact matches of character string data is required.

A primary use of the KEYMATCH() function is to check for duplicate values when adding records. Unlike SEEK(), KEYMATCH() looks for a matching

index value without moving the record pointer and disturbing the current state of the record buffer.

KEYMATCH() ignores the settings for SET FILTER and SET KEY TO, ensuring the integrity of data in a table even when you work with a subset of the table records. KEYMATCH() honors SET DELETED, so that when SET DELETED is ON, existing key values in records marked as deleted are ignored, as if those records did not exist.

If you specify only an expression (<exp>) whose value you want to match, KEYMATCH() searches the current master index for an index key with the same value.

To search indexes other than the current master index, you must specify the index by index position. There are two ways to do this:

- By the index's position in the list of open indexes. Index numbering is complicated if you have open .NDX indexes or open non-production .MDX files. For information on index numbering, see SET INDEX.
- By an index tag's position in an .MDX file. If you do not specify <.mdx filename expC>, the production .MDX is used.

Either way, it is often easier to reference an index tag by name by using the TAGNO() function to get the corresponding position number.

OODML No equivalent.

See Also INDEX, KEY(), MDX(), NDX(), ORDER(), SET INDEX, SET ORDER, TAGNO(), USE

LIST

Displays records from the current table in the result pane of the Command window.

Syntax LIST
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [[FIELDS] <exp list>]
 [OFF]
 [TO FILE <filename>]
 [TO PRINTER]

Description Both LIST and DISPLAY display records in the results pane of the Command window. There are two differences between the commands:

- LIST displays continuously until complete, while DISPLAY pauses after each screenful of information.

- The default scope of LIST is ALL, while the default scope of DISPLAY is NEXT 1, the current record only.

Because LIST does not pause between screens, it is more appropriate when directing output to a file or printer. For more information on the options of LIST, see DISPLAY.

OODML No equivalent.

See Also DISPLAY, SET CONSOLE, SET HEADING

LKSYS()

Returns information about a locked record or file.

Syntax LKSYS(<expN>)

<expN> A number representing the information for LKSYS() to return:

| Value | Returns |
|-------|--|
| 0 | Time when lock was placed |
| 1 | Date when lock was placed |
| 2 | Login name of user who locked record or file |
| 3 | Time of last update or lock |
| 4 | Date of last update or lock |
| 5 | Login name of user who last updated or locked record or file |

Description LKSYS() returns multiuser information contained in a _DBASELOCK field of a DBF table. For LKSYS() to return information, the current table must have a _DBASELOCK field. Use CONVERT to add a _DBASELOCK field to a table. If the current table doesn't contain a _DBASELOCK field, LKSYS() returns an empty string for any value of <expN>.

Note LKSYS() works only with DBF tables.

LKSYS() always returns a string. When LKSYS() returns a date, it is a string containing the date in the current date format dictated by SET DATE and SET CENTURY. Use CTOD() to convert the date string to a date.

When a record is locked, either explicitly or automatically, the time, date, and login name of the user placing the lock are stored in the _DBASELOCK field of that record. When a file is locked, this same information is stored in the _DBASELOCK field of the first physical record in the table.

Passing 0, 1, or 2 as arguments to LKSYS() returns values only after an attempted file or record lock has failed. If a file or record lock on a converted table fails, the information for LKSYS() arguments 0, 1, and 2 is written to a buffer from the record's _DBASELOCK field. If you then pass 0, 1, or 2 to LKSYS(), the information is read from the buffer. The buffer isn't overwritten

until you attempt another lock that fails. Thus, 0, 1, and 2 always return the information that was current at the time of the last lock failure.

You can pass 3, 4, or 5 as arguments to LKSYS() whether or not the current record or file is currently locked. These arguments return information about the last successful record or file lock. When you pass any of these arguments to LKSYS(), it returns information directly from the _DBASELOCK field rather than from an internal buffer.

If you pass 2 or 5 to obtain a user login name, and the _DBASELOCK field is only 8 characters wide, LKSYS() returns an empty string. The first 8 characters of a _DBASELOCK field are the count, time, and date information of the last update or lock, so the field must be wider than 8 characters to fit part or all of the login user name. Set the width of the field with CONVERT.

OODML Check the properties of the rowset.fields["_DBASELOCK"] field.

Example The following function is used to lock individual records. If it fails, it uses LKSYS() to display information on who has the lock and when they got it. SET REPROCESS must be changed to 1 instead of 0 so that if the lock attempt fails the standard dialog, which does not have as much information, will not be displayed.

```
PROCEDURE RecLock
  local cMsg
  do while .t.
    if rlock()
      return .t.
    else
      cMsg = "Locked by: " + lksys(2) + chr(13) + ;
        "since: " + lksys(1) + " " + lksys(0)
      if msgbox( cMsg, "Record is locked by another", 5 + 48 ) == 2
        return .f.
      endif
    endif
  enddo
```

The MSGBOX() used is a Retry/Cancel dialog box. The button number, which MSGBOX() returns, is 2 if the Cancel button is clicked or the user presses *Esc*.

See Also CHANGE(), CONVERT, FLOCK(), RLOCK(), SET LOCK, UNLOCK

LOCATE

Searches a table for the first record that matches a specified condition.

Syntax LOCATE
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL. LOCATE is usually used with a FOR condition.

Description LOCATE performs a sequential search of a table and tests each record for a match to the specified condition. If a match is found the record pointer is left at that record. Issuing CONTINUE resumes the search, allowing additional records meeting the specified condition to be found.

Whenever a match is found, FOUND() returns *true*. If match is not found, the record pointer is left after the last record checked, which usually leaves it at the end-of-file. Also, FOUND() returns *false*.

If SET TALK is ON, LOCATE will display the record number of the matching record in the result pane of the Command window if you are searching a DBF table. If no match is found, LOCATE will display "End of Locate scope"

Because the default scope of the command is ALL, issuing LOCATE with no options will move the record pointer to the first record in the table, because that is the first matching record in that scope. However, there is no practical reason to use LOCATE in this manner. A FOR condition is usually used with LOCATE to find records that match a condition. An understanding of command scope, as explained on page 12-3, is essential to using LOCATE effectively.

LOCATE does not require an indexed table; however, if an index is in use, LOCATE follows its index order. When using the = operator to compare strings, LOCATE uses the rules established by SET EXACT to determine whether the strings match. Use the == operator to perform exact matches regardless of SET EXACT.

The search commands LOCATE and SEEK are each designed for use under particular conditions. LOCATE is the most flexible, accepting expressions of any data type as input and searching any field of a table. For large tables, however, a sequential search using LOCATE might be slow.

Use SEEK or SEEK() for greater speed. Both conduct an indexed search, similar to looking up a topic in a book index and turning directly to the appropriate page, allowing information to be found almost immediately. Once you use the INDEX command to create an index for a table, SEEK uses this index to quickly identify an appropriate record.

You can use SEEK and LOCATE in combination. Use SEEK to quickly narrow down a search and then use LOCATE with the appropriate scope to find the exact you're looking for.

OODML Use the Rowset object's *beginLocate()* and *applyLocate()* methods.

Example The following example uses SEEK and LOCATE in combination to find the first vendor in Texas that is not in either Dallas or Houston. The table is indexed on state and city, but you cannot use SEEK alone to find a matching record.

```
use VENDOR order STATE_CITY
```

LOCK()

```
if seek( "TX" )
  locate while STATE == "TX" for CITY # "Dallas" .and. CITY # "Houston"
  if found()
    *-- Do something
  endif
endif
```

If SEEK() finds a vendor in Texas, the WHILE clause of the LOCATE command restricts the sequential search to Texas. The FOR clause looks for the city match, (or non-match in this case).

See Also CONTINUE, FIND, FOUND(), LOOKUP(), SEEK, SEEK(), SET EXACT

LOCK()

Locks the current record or a specified list of records in a table.

Syntax LOCK([<record list expC>,<alias>] | [<alias>])
<list expC> The list of record numbers to lock, separated by commas.
<alias> The work area in which to lock records.

Description LOCK() is identical to RLOCK(). For more information, see RLOCK().

See Also FLOCK(), RLOCK(), SET LOCK, SET RELATION, SET REPROCESS, UNLOCK

LOOKUP()

Searches a field for a specified expression and, if the expression is found, returns the value of a field within the same record.

Syntax LOOKUP(<return field>, <exp>, <lookup field>)
<return field> The field whose value you want to return if a match is found.
<exp> The expression to look for in the <lookup field>. Specify an alias when referring to fields outside the current work area.
<lookup field> The field you want to search for the value <exp>.

The <return field> and <lookup field> are usually fields in the same table, a table that is not in the current work area. Use the alias name and alias operator (->) to reference fields in other tables.

Description LOOKUP() looks for the first record where <lookup field> matches the specified expression <exp>. The record pointer is left at the matching record. If no match is found, the record pointer is left at the end-of-file. Either way, LOOKUP() then returns the value of <return field>.

Therefore, if no match is found, LOOKUP() returns the blank value for that field, either an empty string (""), zero, a blank date, or *false*, depending on the data type of <lookup field>. Calling FOUND() will also return *true* or *false* to indicate if the search was successful.

LOOKUP() performs a sequential search, unless an index whose key matches <lookup field> is available in the lookup table. To minimize the time LOOKUP() takes to search a table, you should create index keys for your most common lookups.

Because LOOKUP() moves the record pointer you can perform a lookup with related tables, where the <lookup field> is in the parent table, and <return field> is in the child table.

OODML No equivalent.

Example The following event handler displays the city for a zip code that is typed into the control:

```
PROCEDURE zipCode_onChange
    form.city.text = lookup( ZIPCODE->CITY, this.value, ZIPCODE->ZIP_CODE )
```

See Also FOUND(), LOCATE, SEEK, SEEK()

LUPDATE()

Returns the date of the last change to a table.

Syntax LUPDATE([<alias>])

<alias> The work area you want to check.

Description LUPDATE() returns the last update date of the specified table. If no table is open, LUPDATE() returns a blank date.

OODML No equivalent. You may use functions to check the last update date of the table file.

See Also DTOC(), SET CENTURY, SET DATE

MDX()

Returns the names of a DBF table's open .MDX index files.

Syntax MDX([<mdx expN>[, <alias>]])

<mdx expN> A number indicating which open .MDX file whose name to return.

<alias> The work area you want to check.

Description MDX() returns the name of an .MDX file open in the current or specified work area. .MDX files are numbered in the order in which they were opened. The production .MDX file, the one with the same name as the DBF file, is number 1.

If *<mdx expN>* is omitted, the name of the .MDX file containing the current master index tag is returned.

MDX() includes the drive letter (and colon) in the filename. If SET FULLPATH is ON, MDX() also returns the directory location of the .MDX file in addition to the drive and name.

If *<mdx expN>* is higher than the number of open .MDX files, or if you do not specify an index order number and the master index is an .NDX file, MDX() returns an empty string (""). MDX() also returns an empty string if there is no .MDX file open.

OODML No equivalent.

Example The following utility function generates a program file that will recreate all indexes from scratch. This generated program also documents the index tags. It uses the MDX() function to get the name of the .MDX file that contains the active index tag. If there is no active index tag, the production .MDX is used.

```
PROCEDURE GenMDX( cFile )
  local cMdx, cMdxFile
  cMdx = mdx()
  if cMdx == ""
    *-- If no active index tag, try production .MDX
    cMdx = mdx( 1 )
    if cMdx == ""
      msgbox( "No MDX file", "Nothing to do", 48 )
      return
    endif
  endif

  *-- Set OF clause for non-production .MDX
  cMdxFile = iif( cMdx == mdx( 1 ), "", [ of ] + cMdx + [ ] )
  *-- Remove drive and/or path from .MDX filename
  *-- (after setting OF clause, because that checks cMdx)
  cMdx = substr( cMdx, max( rat( ":", cMdx ), rat( "\", cMdx ) ) + 1 )

  local lSafety
  lSafety = ( set( "SAFETY" ) == "ON" )
  set safety on

  if argcount() < 1
    cFile = left( cMdx, rat( ".", cMdx ) - 1 ) + ".PRG"
  endif

  set alternate to ( cFile )
  set console off
  ?
```

```

set alternate on

??"* " + cFile
? "*"
? "** Index file for " + cMdx
if cMdxFile == ""
    ?? " (production .MDX)"
endif
? "*"
? "** Generated on " + dtoc( date() ) + " " + time()
? "*"
?
? [ *-- Delete all current tags from specific .MDX only]
? [do while "" # tag( " ] + cMdx + [ ", 1 )]
? [ delete tag tag( " ] + cMdx + [ ", 1 )]
? [enddo]
?

nNdx = 1
do while "" # key( cMdx, nNdx )
    ? [index tag ] + transform( tag( cMdx, nNdx ), "@! XXXXXXXXXXX" ) + ;
    cMdxFile + [ on ] + key( cMdx, nNdx )
    if "" # for( cMdx, nNdx )
        ?? [ for ] + for( cMdx, nNdx )
    endif
    if descending( cMdx, nNdx )
        ?? [ descending]
    endif
    if unique( cMdx, nNdx )
        ?? [ unique]
    endif
    nNdx = nNdx + 1
enddo

close alternate
if .not. lSafety
    set safety off
endif

```

The function uses the KEY(), FOR(), DESCENDING(), and UNIQUE() functions to get the definition of each index.

See Also INDEX, NDX(), SET FULLPATH, SET INDEX, SET ORDER, TAG(), TAGCOUNT(), TAGNO(), USE

MEMLINES()

Returns the number of lines in a memo field.

Syntax MEMLINES(<memo field> [, <line length expN>])

<memo field> The memo field the MEMLINES() function operates on.

<line length expN> Specifies the line length used in calculating the number of lines in a memo field. *<expN>* can be set to any number from 8 to 255. If *<expN>* is not specified, MEMLINES() calculates each line using the memo width specified using the SET MEMOWIDTH command.

Description The MEMLINES() function returns the number of lines in a memo field based on the memo width specified by the line length parameter. If you don't specify a line length, MEMLINES() uses the width specified by SET MEMOWIDTH, which defaults to 50.

If a word doesn't completely fit within the remainder of a line, MEMLINES() wraps that word and everything on the line following it to the beginning of the next line. If the number of characters in a word is longer than the default or specified memo field line length, MEMLINES() truncates the word at the end of the line and includes the remainder of the word at the beginning of the next line.

A carriage return/line feed combination in the memo text always starts a new line. Note that if the carriage return/line feed is at the end of the memo field contents, the empty blank line that follows it is counted in the total line count.

OODML No equivalent. You cannot accurately determine the amount of text that can fit on a line when using proportional fonts.

See Also MLINE(), SET MEMOWIDTH, STORE MEMO

MLINE()

Extracts a specified line of text from a memo field in the current record.

Syntax MLINE(*<memo field>* [, *<line number expN>* [, *<line length expN>*]])

<memo field> The memo field the MLINE() function operates on.

<line number expN> The number of the line in the memo field returned by the MLINE() function. The default for *<line number expN>* is 1, the first line.

<line length expN> The number that determines the length of a line in the memo field. *<line length expN>* can be set to any number from 8 to 255. If *<line length expN>* is not set, the SET MEMOWIDTH setting specifies the length of the line.

Description MLINE() returns a specified line of text from a memo field. MLINE() treats the text of the memo field as if it were wordwrapped within a display width specified by the SET MEMOWIDTH setting or by *<line length expN>*.

If a word doesn't completely fit within the remainder of a line, MLINE() wraps that word and everything on the line following it to the beginning of the next line. If the number of characters in a word is longer than the default or specified

memo field line length, MLINE() truncates the word at the end of the line and includes the remainder of the word at the beginning of the next line.

OODML No equivalent.

Example

See Also MEMMLINES(), REPLACE MEMO, SET MEMOWIDTH, STORE MEMO

NDX()

Returns the names of a DBF table's open .NDX files.

Syntax NDX([<ndx expN> [, <alias>]])

<ndx expN> A number indicating which open .NDX file whose name to return.

<alias> The work area you want to check.

Description NDX() returns the name of the .NDX file open in the current or specified work area. .NDX files are numbered in the order in which they were opened. The first one is number 1.

If <ndx expN> is omitted, the name of the .NDX file containing the current master index tag is returned.

NDX() includes the drive letter (and colon) in the filename. If SET FULLPATH is ON, NDX() also returns the directory location of the .NDX file in addition to the drive and name.

If <ndx expN> is higher than the number of open .NDX files, or if you do not specify an index order number and the master index is an index tag in an .MDX file, NDX() returns an empty string ("").

OODML No equivalent.

See Also DBF(), FIELD(), KEY(), MDX(), ORDER(), SET FULLPATH, SET INDEX, SET ORDER, TAG(), USE

OPEN DATABASE

Establishes a connection to a database server or a database defined for a specific directory location.

Syntax OPEN DATABASE <database name>
[LOGIN <username>/<password>]

ORDER ()

[WITH <option string>]
[AUTOEXTERN]

<database name> The name, or alias, of the database you want to open. Database aliases are created using the BDE Administrator.

<user name>/<password> The user name and password, separated by a slash, required to access the database.

WITH <option string> Character string specifying server-specific information required to establish a database server connection. For information about establishing database server connections, refer to your Borland SQL Link documentation, and contact your network or database administrator for specific connection information.

AUTOEXTERN Treat all stored procedures as EXTERN. This eliminates the need for the user to EXTERN SQL any stored procedure calls. Once the database is open, the stored procedures are immediately available. For use with Interbase and Oracle databases only.

Description The OPEN DATABASE command is used to establish a connection with a database defined with the BDE Administrator. When opening a database, you need to specify whatever login parameters and database-specific information that connection requires. Typically, your network or system administrator can provide you with the information necessary to establish connections to established databases and database servers at your site.

OODML Use a Database object.

See Also CLOSE..., DATABASE(), SET DATABASE, SET DBTYPE

ORDER()

Returns the name of the current master index.

Syntax ORDER([<alias>])

<alias> The work area you want to check.

Description ORDER() returns the name of the current master index. For DBF tables, this could be either the name of an index tag in an .MDX file, or the name of an .NDX file (the name only, no drive or extension as returned by the NDX() function). For all other table types, the name is the name of an index tag.

ORDER() returns an empty string ("") if the table is in its natural order: either its primary key order, if it has a primary key; or no active index.

Some routines need to use a specific index. Use ORDER() to get the name of the current master index before switching to the desired index and then use the SET ORDER command to later restore the master index.

OODML Check the *indexName* property of the Rowset object.

Example In this example, a function switches to a specific index tag before calling another function that creates a new record:

```
PROCEDURE NewStudent
  local cOrder
  cOrder = order()
  set order to STUDENT_ID
  NewRec()
  set order to ( cOrder )
```

This example function assumes that the Students table is the currently selected table, which might be ordered according to name, average test score, or some other index. This function saves the index order of the table in the variable cOrder. At the end of the function, that index is restored with the SET ORDER command using the parentheses as indirection operators.

The NewRec function is shown in the example for APPEND.

See Also ALIAS(), KEY(), MDX(), NDX(), SELECT(), SET INDEX, SET ORDER, TAG(), USE

PACK

Removes all records from a DBF table that have been marked as deleted.

Syntax PACK

Description Use PACK to remove records from the current DBF table that were previously marked as deleted by the DELETE command. You need to open a table for exclusive use before using PACK.

After you execute a PACK command, the disk space used by the deleted records is reclaimed when the table is closed. All open index files are automatically re-indexed after PACK is executed. (Use REINDEX to update closed indexes.)

Space in .DBT memo files associated with deleted records is not reduced, however, when you use the PACK command. To reclaim space in a memo file, you need to use COPY to copy the original table.

Use PACK with caution. Records that have been marked for deletion but not yet eliminated with PACK can be undeleted and restored to a table using RECALL. Records eliminated with PACK are permanently lost and can't be recovered.

SET DELETED ON provides many of the benefits of PACK without actually removing records from a table. With SET DELETED ON, most commands function as if records marked for deletion had been eliminated from a table.

Because PACK requires the exclusive use of a table, it may be difficult to find a time to PACK a table for applications that run continuously. Also for large

tables, PACK is time-consuming and requires enough disk space to make a copy of the table. Consider recycling deleted records instead, which is quicker and safer. For an example of how to implement record recycling, see the examples for APPEND and BLANK.

To permanently remove all records of the current table in one step, use the ZAP command.

OODML Use the Database object's *packTable()* method.

See Also DELETE, RECALL, SET DELETED, ZAP

RECALL

Restores records that were previously marked as deleted in the current DBF table.

Syntax RECALL
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]

<scope>
FOR <condition 1>
WHILE <condition 2> The scope of the command. The default scope is NEXT 1, the current record only.

Description Use RECALL to undelete records that have been marked as deleted in the current DBF table with DELETE but have not yet been removed with PACK. Executing DELETE marks the record as deleted but doesn't physically remove them from the table. If SET DELETED is ON (the default), these deleted records cannot be seen. RECALL reverses this process, unmarking the records and fully restoring them to the table.

Records eliminated with PACK or ZAP are permanently removed and can't be recovered using RECALL.

When using RECALL, SET DELETED should be OFF; otherwise you will not be able to see the deleted records you want to recall. Using RECALL on records that are not marked as deleted has no effect.

OODML Soft deletes are not supported.

Example See APPEND for an example of RECALL being used in record recycling.

See Also DELETE, PACK, SET DELETED, ZAP

RECCOUNT()

Returns the number of records in a table.

Syntax RECCOUNT([<alias>])

<alias> The work area you want to check.

Description RECCOUNT() retrieves a count of a table's records from the table header, which holds information about the table structure. In contrast, COUNT with no options yields a record count by actually counting the table's records using the table's current filter, key constraints, the setting of SET DELETED and so on. RECCOUNT() includes all records, even those marked as deleted, and is always instantaneous; COUNT is not.

If no table is active in the specified work area, RECCOUNT() returns zero.

You can use RECSIZE() in combination with RECCOUNT() to determine the approximate size, in bytes, of a table.

OODML In some cases, the Rowset object's *rowCount*() method will return the same value.

See Also DIR, DBF(), DISKSPACE(), DISPLAY STRUCTURE, RECNO(), RECSIZE()

RECNO()

For DBF tables, returns the current record number. For all other table types, returns a bookmark of the current position in a table.

Syntax RECNO([<alias>])

<alias> The work area you want to check.

Description RECNO() returns the current record number of the table in the current or a specified work area, if that table is a DBF table. For all other table types, RECNO() behaves like BOOKMARK(), returning a bookmark. If no table is open in the specified work area, RECNO() returns a value of 0.

If the record pointer move to the end-of-file (past the last record in the table), RECNO() returns a value that is one more than the total number of records in the table. Therefore, RECNO() returns a value of 1 if there are no records in the table—RECCOUNT() would return zero.

The use of BOOKMARK() is recommended instead of RECNO(). Besides returning a consistent data type with all tables, with BOOKMARK() you can bookmark the position at the end-of-file and GO back to it. Although RECNO() will return a record number for the end-of-file, you cannot GO to that record number, because there actually is no record with that number.

RECSIZE()

OODML Use the Rowset object's *bookmark()* method.

See Also BOF(), BOOKMARK(), EOF(), RECCOUNT()

RECSIZE()

Returns the number of bytes in a record of a table.

Syntax RECSIZE([<alias>])

<alias> The work area you want to check.

Description RECSIZE() returns the number of bytes in a record of a table in the current or specified work area. If no table is open in the specified work area, RECSIZE() returns a value of zero.

LIST STRUCTURE and DISPLAY STRUCTURE also show the size of a table's records.

OODML Use a loop to add up the *length* properties of the Field objects in the *fields* array.

Example The following example uses RECSIZE() to determine if there is enough disk space to append the records contained in another file (which might be on a CD-ROM or other large capacity disk). The other file is opened with the alias "OTHERFILE":

```
if reccount( "OTHERFILE" ) * recsize() > diskspace()
    msgbox( "Insufficient disk space to append records", "Error", 48 )
endif
```

See Also DBF(), DIR, DISPLAY STRUCTURE, LIST STRUCTURE, RECCOUNT(), RECNO()

REFRESH

Updates data buffers to reflect the latest changes to data.

Syntax REFRESH [<alias>]

<alias> The work area to refresh.

Description Use REFRESH to update specified work area data buffers so that data you display reflects the latest changes made to tables by other users.

OODML Use the Rowset object's *refresh()* method or the Query object's *requery()* method.

See Also SET REFRESH

REINDEX

Regenerates all open index files in the current work area.

Syntax REINDEX

Description Use REINDEX to manually regenerate all open indexes in the current work area. In a normal application, indexes remain open whenever their table is open. These indexes are automatically updated whenever there is a change to the table, so there is no need to manually REINDEX.

You would use REINDEX if your application uses non-production .MDX files or .NDX files that are not always open. To update these indexes, open them with the corresponding table and issue REINDEX.

You might also use REINDEX if you suspect that the index files have been damaged. REINDEX rebuilds the entire index file from scratch.

You must have exclusive use of a table to REINDEX it.

OODML Use the Database object's *reindex()* method.

Example In the following example, a DBF file is generated and downloaded from a mainframe on a weekly basis, overwriting the previous week's file. The DBF has an .MDX file that must be manually regenerated for each week's download before processing the downloaded data. The beginning of the process looks like this:

```
use DOWNLOAD index PROCESS.MDX exclusive
reindex
set order to NAME
*-- Rest of process
```

See Also INDEX, SET INDEX, SET ORDER, USE

RELATION()

Returns the link expression defined with the SET RELATION command.

Syntax RELATION(<expN> [, <alias>])

<expN> The number of a relation that you want to return.

<alias> The work area you want to check.

Description RELATION() returns a string containing the expression that links one table with another that was defined with the SET RELATION command. You must specify the number of the relation; if the table in the current or specified work area is linked to only one table, that <expN> is the number 1. RELATION() returns an empty string ("") if no relation is set in the <expN> position.

Use `RELATION()` to save the link expressions of all `SET RELATION` settings for later use when restoring relations. To save the target table (the table into which you `SET` a `RELATION`), use the `TARGET()` function.

OODML Check the detail Rowset object's *masterFields* and *masterRowset* properties, or the detail Query object's *masterSource* property to determine the nature of the master-detail linkage.

See Also `ALIAS()`, `CREATE QUERY`, `CREATE VIEW`, `CREATE VIEW...FROM ENVIRONMENT`, `SELECT()`, `SET RELATION`, `SET VIEW`, `SET()`, `TARGET()`

RELEASE AUTOMEM

Clears automem variables from memory.

Syntax `RELEASE AUTOMEM`

Description Automem variables are private or public memory variables that have the same name as the fields of the currently selected table. These variables can be created manually, or with the `STORE AUTOMEM`, `CLEAR AUTOMEM`, or `USE...AUTOMEM` commands.

`RELEASE AUTOMEM` releases any private or public memory variables that have the same name as one of the fields in the currently selected table, no matter how or for what purpose the variable was created.

Closing a table or moving to another work area doesn't automatically release a table's associated automem variables. *Visual dBASE* doesn't recognize a variable as an automem variable, even if it was created as an automem variable, if it doesn't have the same name as a field in the current table. But because automem variables are usually private variables, and private variables are automatically released when the routine that created them is complete, there is rarely any reason to issue `RELEASE AUTOMEM` in an application.

OODML The Rowset object's contains an array of Field objects, accessed through its *fields* property. These Field objects have *value* properties that may be programmed like variables.

See Also `CLEAR AUTOMEM`, `RELEASE`, `STORE AUTOMEM`, `USE`

RENAME TABLE

Changes the name of a specified table.

Syntax `RENAME TABLE <old table name> TO <new table name>`
`[[TYPE] PARADOX | DBASE]`

<old table name> The table you want to rename.

<new table name> The new name of the table. If you rename a table in a database, you must specify the same database as the destination of the new table. Also, the new table name must be the same type as the old table.

[TYPE] PARADOX | DBASE Specifies the type of table you want to rename, in case you do not specify a file extension with *<old table name>*. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

Description Use the RENAME TABLE command to change the name of a table and its associated files, if any. You cannot rename an open table, and the new table name cannot already exist in the same directory or database.

OODML Use the Database object's *renameTable()* method.

See Also CLOSE..., COPY, COPY FILE, USE

REPLACE

Replaces the contents of fields with data from expressions.

Syntax REPLACE
 <field 1> WITH <exp 1> [ADDITIVE]
 [, <field 2> WITH <exp 2> [ADDITIVE]...]
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [REINDEX]

<field> WITH <exp> Designates fields to be replaced by the value of the specified expressions. Multiple fields of a record may be changed by including additional <field n> WITH <exp n> expressions, separated by commas.

ADDITIVE Adds text to the end of memo field text instead of replacing existing text. You can use ADDITIVE only when the specified field is a memo field in a DBF table.

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is NEXT 1, the current record only.

REINDEX Specifies that all affected indexes are rebuilt once the REPLACE operation finishes. Without REINDEX, Visual dBASE updates all open indexes after replacing each record in the scope. When replacing many or all records in a table that has multiple open indexes, REPLACE executes faster with the REINDEX option.

Description The REPLACE command overwrites a specified field with new data. The field you select can be any type, including memo fields. (To replace binary or OLE

fields, use REPLACE BINARY and REPLACE OLE.) The field and the expression specified by the WITH clause must have the same data type.

When storing a long string or the contents of a memo field into a shorter character field, the data is truncated. Use the ADDITIVE option to add a character string to the end of existing memo field text. You can leave a blank space at the beginning of the string to provide proper spacing.

Be careful when replacing data in the key fields of the current master index, in more than one record (that is, with the *<scope>*, WHILE, or FOR options). *Visual* dBASE automatically updates all open index files after a REPLACE operation finishes. After replacing data that changes the value in the key field in the master index, the record and the record pointer immediately move to the position in the index based on the new value of a key. If replacement in the key field causes a record (and the pointer) to move down past other records that fall within the scope or meet the specified conditions, those records are not replaced. If replacement in the key field causes a record to move up before records that have already been replaced, those records may be replaced again.

To make replacements to an indexed table's key field, you may place the table in natural order with the SET ORDER TO command, or use other techniques, one of which is shown in the example. Replacements in key fields other than the key fields of the master index don't affect the order of the current index and can be made over multiple records without complications.

When replacing a Numeric or Float field in a DBF table, the magnitude of the new value should not exceed the integer portion of the field. For example, if the Numeric field is defined as width 4 with 1 decimal place, you cannot have a number greater than 99.9. If so, the field contents are replaced with an approximation to the new value in scientific notation, if it will fit; otherwise the field contents are replaced with asterisks, destroying stored data. Scientific notation requires a field width of at least 7 characters. This condition is not an error, but *Visual* dBASE will display a numeric overflow warning message in the results pane of the Command window.

Other field types that store numbers, including Long and Short integers, have a numeric range they support. Make sure that the number you attempt to store does not exceed those ranges.

Use the alias operator in the *<field>* (that is, *alias->field*) to REPLACE fields in tables other than the currently selected table. You may mix fields from different tables in the same REPLACE statement, although the scope of the command is based on the current table. If there is no relation between the current table and other tables, traversing the current table—for example, because of an ALL scope—does not move the record pointer in the other tables.

OODML Assign values directly to the *value* properties of the Field objects in the Rowset object's *fields* array (in a loop that traverses the rowset if necessary).

Example The following statement updates salaries to conform to the new minimum wage, which is in the variable nMinWage:

```
replace for SALARY < nMinWage SALARY with nMinWage
```

The following statement gives everyone a 10% raise, and two weeks (80 hours) extra vacation:

```
replace all SALARY with SALARY * 1.1, VACATION with VACATION + 80
```

The next example replaces all instances of an inadvertently assigned duplicate customer ID number with their actual customer number.

```
PROCEDURE ChangeCustID( oldCust, newCust )
  local cOrder
  cOrder = order()
  set order to CUST_ID
  do while seek( oldCust )
    replace CUST_ID with newCust
  enddo
  set order to ( cOrder )
```

The routine uses the `SEEK()` function to find any records that have the old customer ID number. The number is replaced in that record only, which moves the record into its updated position in the index. This movement doesn't matter because the loop uses the `SEEK()` function again to find another match.

The current index order is saved before the loop, and restored at the end of the routine.

See Also APPEND, BLANK, BROWSE, EDIT, REINDEX, REPLACE AUTOMEM, REPLACE BINARY, REPLACE MEMO, REPLACE OLE, SET RELATION, UPDATE

REPLACE AUTOMEM

Replaces fields in the current table that have corresponding automem variables.

Syntax REPLACE AUTOMEM

Description Automem variables are private or public memory variables that have the same name as the corresponding fields of the current table. Automem variables are used to hold data that will be stored in the fields of records. You can manipulate data stored in automem variables as memory variables rather than as field values, and you can validate the data before storing the data to a table.

Create a set of automem variables for the fields in a table with `USE...AUTOMEM`, `CLEAR AUTOMEM`, or `STORE AUTOMEM` (or create the variable manually). To add new records to a table and fill the fields with values from corresponding automem variables, use `APPEND AUTOMEM`. To update the fields of existing records with values from corresponding automem variables, use `REPLACE AUTOMEM`.

Use `REPLACE AUTOMEM` to update all the fields of a record without having to name the fields. By contrast, with the `REPLACE` command, you need to name every field you want updated.

Remember that an automem variable and its corresponding field have the same name. When a command allows an argument that could be either a field or a private or public memory variable, *Visual dBASE* assumes the argument refers to a field. To distinguish the memory variable from the field, prefix the names of automem variables with M->.

REPLACE AUTOMEM updates the current record. It can't update all records within a specified scope or all records matching a condition, as the REPLACE command can with the options <scope>, FOR <condition>, and WHILE <condition>.

REPLACE AUTOMEM doesn't replace field data with data from a memory variable with the same name but of a different data type. Those variables are ignored. If a field does not have a corresponding private or public variable with the same name, that field is left unchanged.

OODML The Rowset object's contains an array of Field objects, accessed through its *fields* property. These Field objects have *value* properties that may be programmed like variables.

See Also APPEND AUTOMEM, CLEAR AUTOMEM, REPLACE, STORE AUTOMEM, USE

REPLACE BINARY

Replaces the contents of a binary field with the contents of a binary file.

Syntax REPLACE BINARY <binary field name> FROM <filename>
[TYPE <binary type user number>]

<binary field name> The binary field of the current table that is replaced by the contents of <filename>.

FROM <filename> Specifies the file to copy to the binary field in the current record. If you specify a file without including its extension, *Visual dBASE* assumes a .BMP extension; however, the file may be any type.

TYPE <binary type user number> Specifies a number that can be used to identify the type of binary data being stored. By default, *Visual dBASE* attempts to detect the type of file and assigns the appropriate binary type. Use the BINTYPE() function to retrieve the type number. The range is from 1 to 32K – 1 for user-defined file types and 32K to 64K – 1 for predefined types (although any number may be specified within the allowable range).

| Predefined binary type numbers | Description |
|--------------------------------|-------------------------|
| 1 to 32K – 1 (32,767) | User-defined file types |
| 32K (32,768) | .WAV files |
| 32K + 1 (32,769) | Image files |

Description Use REPLACE BINARY to copy a binary file to the current record's binary field. You can copy one binary file to each binary field of each record in a table.

While memo fields may contain types of information other than text, binary fields are recommended for storing images, sound, or any other binary or BLOB type data.

See class Image for a list of image types that *Visual* dBASE can automatically detect and display.

OODML Use the Field object's *replaceFromFile()* method. The binary type option is not supported.

Example The following event handler displays a dialog to pick an image file, then stores the contents of that file in a binary field, erasing any previous contents.

```
PROCEDURE importImageButton_onClick
    local cFile
    cFile = getFile( "*.bmp", "Image file to import" )
    if "" # cFile
        replace binary IMAG_FIELD from ( cFile )
    endif
```

GETFILE() will return an empty string if no file is selected. In the IF statement, the order of the empty string and the variable cFile is important. If they were the other way around and SET EXACT is OFF, then the IF statement would always be *false*.

The parentheses are used as indirection operators to get the name of the file from the variable. Without them, *Visual* dBASE would attempt to append a file named cFile.

The binary type for an image, 32,769, is automatically assigned.

See Also APPEND MEMO, BINTYPE(), COPY BINARY, REPLACE MEMO, REPLACE MEMO...FROM, REPLACE OLE, RESTORE IMAGE

REPLACE FROM ARRAY

Transfers data stored in an array to the fields of the current table.

Syntax REPLACE FROM ARRAY <array>
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [FIELDS <field list>]
 [REINDEX]

<array> The name of the array that you want to transfer data from.

<scope>**FOR <condition 1>****WHILE <condition 2>** The scope of the command. The default scope is REST. The dimensions and size of <array> also controls which records are updated.**FIELDS <field list>** Restricts data replacement to the fields specified by <field list>.**REINDEX** Specifies that all affected indexes are rebuilt once the REPLACE FROM ARRAY operation finishes. Without REINDEX, *Visual* dBASE updates all open indexes after replacing each record in the scope. When replacing many or all records in a table that has multiple open indexes, REPLACE FROM ARRAY executes faster with the REINDEX option.

Description Use REPLACE FROM ARRAY to transfer values from an array into fields of the current table. REPLACE FROM ARRAY treats the columns in a one-dimensional array like a single record of fields; and treats a two-dimensional array like a table, with the rows (the first dimension) of the array like records, and the columns (the second dimension) like fields.

For a one-dimensional array, REPLACE FROM ARRAY will replace the first record in the command scope that matches the specified condition. If there is no specified scope and condition, the current record is replaced.

For a two-dimensional array, REPLACE FROM ARRAY will attempt to copy each row in the array to a record in the command scope that matches the specified condition until the end-of-scope or all rows of the array have been copied, in which case the record pointer is left at the last record replaced. As with the REPLACE command, be careful if are changing the values of the key fields of the current master index; see REPLACE for details.

If the array has more columns than the table has fields, the additional elements are ignored. Similarly, if a two-dimensional array has more rows than can be copied to the table, the additional rows are ignored.

REPLACE FROM ARRAY does not replace memo (or binary) fields; these fields should not be counted when sizing the array, and cannot be included in the <field list>.

The data types of the array must match those of corresponding fields in the table you are replacing. If the data type of an array element and a corresponding field don't match, an error occurs.

OODML Use a loop to copy the elements of the array into the *value* properties of the Field objects in the rowset's *fields* array, nested in another loop to traverse the rowset if necessary.

See Also APPEND FROM ARRAY, COPY TO ARRAY, REPLACE

REPLACE MEMO

Copies a text file into a memo field.

Syntax REPLACE MEMO <memo field> FROM <filename>
[ADDITIVE]

<memo field> The memo field to replace.

FROM <file name> The name of the text file. The default extension is .TXT.

ADDITIVE Causes the new text to be appended to existing text. REPLACE MEMO without the ADDITIVE option causes *Visual* dBASE to overwrite any text currently in the memo field.

Description Use the REPLACE MEMO command to insert the contents of a text file into a memo field. You may use an alias name and the alias operator (that is, *alias->memofield*) to specify a memo field in the current record of any open table.

REPLACE MEMO is identical to APPEND MEMO, except that REPLACE MEMO defaults to overwriting the current contents of the memo field, and has the option of appending, while APPEND MEMO is the opposite.

While memo fields may contain types of information other than text, binary fields are recommended for storing images, sound, and other user-defined binary type information. Use OLE fields for linking to OLE documents from other Windows applications.

OODML Use the Field object's *replaceFromFile()* method.

Example The following event handler displays a dialog to pick a text file, then stores the contents of that file in a memo field, erasing any previous contents.

```
PROCEDURE importTextButton_onClick
    local cFile
    cFile = getfile( "*.txt", "Text file to import" )
    if "" # cFile
        replace memo MEMO_FIELD from ( cFile )
    endif
```

GETFILE() will return an empty string if no file is selected. In the IF statement, the order of the empty string and the variable cFile is important. If they were the other way around and SET EXACT is OFF, then the IF statement would always be *false*.

The parentheses are used as indirection operators to get the name of the file from the variable. Without them, *Visual* dBASE would attempt to append a file named cFile.

See Also APPEND MEMO, COPY MEMO, REPLACE BINARY, REPLACE MEMO...WITH ARRAY, REPLACE OLE

REPLACE OLE

Inserts an OLE document into an OLE field.

Syntax REPLACE OLE <OLE field name> FROM <filename>
[LINK]

<OLE field name> The field where an OLE document is inserted.

FROM <file name> The file that identifies an OLE document, including its extension. There is no default extension.

LINK LINK stores a pointer to the OLE document. By default, *Visual* dBASE embeds the OLE document itself in the specified memo field.

Description Use REPLACE OLE to insert the contents of an OLE document into an OLE field. You can either embed the actual OLE document in an OLE field (the default) or access the OLE document by linking it to the OLE field.

If you link the OLE document, the OLE field contains only a reference to the OLE document. As long as the OLE document remains in the same location, the OLE field displays the most current version of the document.

If you embed the OLE document, the OLE field contains a copy of the document. There are no links between the field and the OLE document: therefore, any changes to the original version of the OLE document are not reflected in the embedded document.

OODML Use the Field object's *replaceFromFile()* method. The file is embedded; you cannot link it.

See Also CLASS OLE

RLOCK()

Locks the current record or a specified list of records in the current or specified table.

Syntax RLOCK([<list expC>, <alias>] | [<alias>])

<list expC> The list of record numbers to lock, separated by commas. If omitted, the current record is locked.

<alias> The work area to lock.

You don't have to specify record numbers if you want to specify a value for <alias> only. However, if you have specified record numbers, you must specify an <alias>.

Description Use RLOCK() to lock the current record or a list of records in any open table. If you don't pass RLOCK() any arguments, it locks the current record in the

current table. If you pass only *<alias>* to RLOCK(), it locks the current record in the specified table. If RLOCK() is successful in locking *all* the records you specify, it returns *true*; otherwise it returns *false*. You can lock up to 100 records in each table open at your workstation with RLOCK().

You can view and update a record you lock with RLOCK(). Other users can view this record but can't update it. When you lock a record with RLOCK(), it remains locked until you do one of the following:

- Issue UNLOCK
- Close the table

All commands that change table data cause *Visual* dBASE to attempt to execute an automatic record or file lock. If *Visual* dBASE fails to execute an automatic record or file lock, an error occurs. You might want to use RLOCK() to handle a lock failure yourself, instead of letting the error occur.

RLOCK() can't lock the records you specify when any of the following conditions exist:

- Another user has locked, explicitly or automatically, the current record or one of the records in *<list expC>*.
- Another user has locked, explicitly or automatically, the table that contains the records you're trying to lock.

When SET REPROCESS is set to 0 (the default) and RLOCK() can't immediately lock its records, *Visual* dBASE prompts you to attempt the lock again or cancel the attempt. Until you choose to cancel the function, RLOCK() repeatedly attempts to get the record locks. Use SET REPROCESS to eliminate being prompted to cancel the RLOCK() function, or to set the number of locking attempts.

RLOCK() is similar to FLOCK(), except FLOCK() locks an entire table. Use FLOCK(), therefore, when you need to have sole access to an entire table or related tables—for example, when you need to update multiple tables related by a common key—or when you want to update more than 100 records at a time.

When you set a relation in a *parent table* with SET RELATION and then lock a record in that table with RLOCK(), *Visual* dBASE attempts to lock all *child records* in *child tables*. For more information on relating tables, see SET RELATION.

RLOCK() is equivalent to LOCK().

OODML Use the Rowset object's *lockRow()* method.

See Also FLOCK(), SET LOCK, SET RELATION, SET REPROCESS, UNLOCK

ROLLBACK()

Cancels the transaction by undoing all logged changes.

Syntax ROLLBACK([<database name expC>])

<database name expC> The name of the database in which to rollback the transaction.

- If you began the transaction with BEGINTRANS(<database name expC>), you must issue ROLLBACK(<database name expC>). If instead you issue ROLLBACK(), Visual dBASE ignores the ROLLBACK() statement.
- If you began the transaction with BEGINTRANS(), <database name expC> is an optional ROLLBACK() argument. If you include it, it must refer to the same database as the SET DATABASE TO statement that preceded BEGINTRANS().

Description A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling ROLLBACK(). Otherwise, COMMIT() is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

Since new rows have already been written to disk, rows that were added during the transaction are deleted. In the case of DBF tables, the rows are marked as deleted, but are not physically removed from the table. If you want to actually remove them, you can pack the table with PACK. Rows that were just edited are returned to their saved values.

All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.

For more information on transactions, see BEGINTRANS().

OODML Call the *rollback()* method of the Database object.

See Also BEGINTRANS(), COMMIT(), SET EXCLUSIVE

SCAN

Steps through each record in the current table, executing specified statements on each record that meets specified conditions.

Syntax SCAN [<scope>] [FOR <condition 1>] [WHILE <condition 2>]
 [<statements>]
 ENDSCAN

<scope>**FOR <condition 1>****WHILE <condition 2>** The scope of the command. The default scope is ALL.**<statements>** Statements to execute for each record visited.**ENDSCAN** A required keyword that marks the end of the SCAN loop.

Description Use SCAN to process the current table record by record. With no scope options, SCAN starts with the first record in the table in the current index order and visits all the records, stopping at the end-of-file. You may specify a different <scope> and a WHILE condition to control the range of records, and a FOR condition that each record must pass for the <statements> to be executed.

At the end of each loop, dBASE automatically moves the record pointer forward one record in the table before returning to the beginning of the loop; therefore, don't include a SKIP command. You may use the EXIT command to exit out of the loop and the LOOP command to go to the next record, skipping all remaining commands in the loop.

You may nest SCAN loops, except that you cannot nest SCAN loops for the same table.

SCAN works like a DO WHILE .NOT. EOF()...SKIP...ENDDO construct; however, with SCAN you can specify conditions with FOR, WHILE, and <scope>. SCAN also requires fewer lines of code than DO WHILE.

When using SCAN with an indexed table, don't change the value of a field that is (or is part of) the master index key. When you change the value of such a field, *Visual* dBASE repositions the record in the index file, which might cause unintended results. For example, if you change a key field that causes its record to move to the end of the index, that record might have the SCAN...ENDSCAN statements executed on it a second time.

If you change work areas within a SCAN loop, select the work area containing the table being scanned before control passes back to the first statement in the SCAN loop.

OODML This example opens a table named FOO and traverses all the records, copying the value of the character field C1 to a throw-away variable, using a SCAN loop and the OODML equivalent:

```
use F00
scan
  x = C1
endscan
use

local q, r
q = new Query( "select * from F00" )
r = q.rowset
if r.first()
do
  x = r.fields[ "C1" ].value
```

```

        until not r.next()
    endif

```

Of note:

- A “SELECT * FROM” query is equivalent to a plain USE command.
- A reference to the query’s rowset is assigned to another variable as shorthand. It also executes a bit quicker.
- A SCAN—without any scope qualifiers like REST or NEXT—always starts at the beginning of the table, so a call to the *first()* method is needed.
- If *first()* returns false, there’s nothing to do
- A DO...UNTIL loop is used so that the navigation happens after processing the current row. Since the *first()* method returned true to get into the loop, there must be at least one row to process.
- When *next()* returns false, you’ve hit the EOF, which terminates the loop.

See Also DO WHILE, DO...UNTIL, FOR...NEXT, INDEX, LOCATE, SEEK, SKIP

SEEK

Searches for the first record in an indexed table whose key fields matches the specified expression or expression list.

Syntax SEEK <exp> | <exp list>

<exp> The expression to search for in an index for a DBF table.

<exp list> One or more expressions, separated by commas, to search for in a simple or composite key index for non-DBF tables.

Description Visual dBASE can search a table for specific information either by a sequential search of a table or by an indexed search of the table's master index. A sequential search is similar to looking for information in a book by reading the first page, then the second, and so on, until the information is found or all pages have been read. LOCATE uses this method, checking each record until the information is found or the last record has been inspected.

An indexed search is similar to looking up a topic in a book index and turning directly to the appropriate page. Once a table index is created, SEEK can use this index to quickly identify the appropriate record.

SEEK looks for the first match in the index. If a match is found, the record pointer of the associated table is positioned at the record containing the match, and FOUND() returns *true*.

Use SKIP to access other records whose key fields match the index key fields or expression. SKIP advances the record pointer one record; because of the indexed order, other matches immediately follow the first. However, SKIP after SEEK (unlike CONTINUE after LOCATE) doesn't search for a match; it moves

the record pointer one record whether or not it finds a match. You can combine SEEK and LOCATE or SEEK and SCAN (both with the WHILE clause) to do a quick indexed search for the first matching record before looking through or processing all the other matches.

The SET NEAR setting determines whether *Visual* dBASE, after an unsuccessful SEEK, positions the record pointer at the end-of-file or at the record in the indexed table immediately after the position at which the value searched for would have been found. If SET NEAR is OFF (the default) and SEEK is unsuccessful, EOF() returns *true* and FOUND() returns *false*. If SET NEAR is ON and SEEK is unsuccessful, EOF() returns *false* (unless the position at which the sought value would have been found is the last record in the index), and FOUND() returns *false*.

The expression you look for with SEEK must match the key expression or fields of the master index. For example, if the master index key uses UPPER(), the search expression must also be in uppercase.

For tables that support composite key indexes based on multiple fields, specify a value for each field in the composite key, separated by commas.

When you seek a key expression of type character, the rules established by SET EXACT determine if a match exists. If SET EXACT is OFF (the default) only the beginning characters of the key field need to be used for SEEK to find a match. For example, if SET EXACT is OFF, SEEK "S" will find "Sanders", or whatever the first key value is that starts with "S". If SET EXACT is ON, the expression must be identical to the key field for a match to exist.

SEEK and LOCATE each have their own advantages. SEEK conducts the most rapid searches; however, it requires an indexed table and can search only for values of the key expression.

If the information for which you are searching is in an unindexed table or is not contained in the key fields of an index, you can use LOCATE. LOCATE accepts any logical condition, which can specify any fields in the table in any combination. For large tables, however, a sequential search using LOCATE can be slow. In such cases, you might want to use INDEX to create a new index and then use SEEK or SEEK().

The SEEK() function works like SEEK followed by FOUND(), except that SEEK searches in the current work area, while SEEK() can search in the current or a specified work area. However, SEEK() can only search for a single expression; it does not support composite keys based on multiple fields. SEEK() returns *true* or *false* depending on whether the search is successful.

OODML Use the Rowset object's *findKey()* or *findKeyNearest()* methods.

See Also EOF(), FOUND(), INDEX, LOCATE, SEEK(), SET EXACT, SET NEAR

SEEK()

Searches for the first record in an indexed table whose key matches the specified expression.

Syntax `SEEK(<exp> [, <alias>])`

<exp> The key value to search for.

<alias> The work area you want to search

Description `SEEK()` evaluates the expression `<exp>` and attempts to find its value in the master index of the table opened in the current or specified work area. `SEEK()` returns *true* if it finds a match of the key expression in the master index, and *false* if no match is found.

The `SEEK()` function combines the `SEEK` command and the `FOUND()` function, adding the ability to search in any work area. However, `SEEK()` does not support composite key indexes based on multiple fields used by non-DBF tables.

Because an index search is almost always followed by a test to see if the search was successful, when searching DBF tables, use `SEEK()` instead of `SEEK` and `FOUND()`. `FOUND()` will return the result of the last `SEEK()` as well.

`SET NEAR` and `SET EXACT` affect `SEEK()` the same way they affect `SEEK`. See `SEEK` for more details.

OODML Use the Rowset object's `findKey()` or `findKeyNearest()` methods.

Example The following example uses `SEEK()` to prune a customer table by deleting all records that do not have any orders.

```
use CUSTOMERS
use ORDERS in select() order CUST_ID
delete for .not. seek( CUST_ID, "ORDERS" )
```

The Customers table is in the current work area while the `SEEK()` is performed in the Orders table.

See Also `EOF()`, `FOUND()`, `INDEX`, `SEEK`, `SET EXACT`, `SET NEAR`

SELECT

Sets the current work area in which to open or perform operations on a table.

Syntax `SELECT <alias>`

<alias> The work area to select.

Description Use SELECT to choose a work area in which to open a table, or to specify a work area in which a table is already open. Many commands operate on the currently selected work area only, or by default.

To select a table that is already open, its alias name is preferred over the work area number, because tables may be opened in different work areas depending on conditions, and the alias name will always select the right table (or cause an error if the table is not opened), while the work area number may take you to the wrong table.

Each work area supports its own value of FOUND() and an independent record pointer. Changes in the record pointer of the active work area have no effect on the record pointers of any other work areas, unless you set a relation between the work areas with the SET RELATION command.

If the <alias> is in a variable, use the parentheses as indirection operators. For example, if xAlias is a variable containing a work area number or alias name, use SELECT(xAlias). Otherwise, *Visual* dBASE will attempt select a work area with the alias name "xAlias".

OODML There is no concept of the "current" Query object. Use your usual object management techniques to manage Query objects.

See Also ALIAS(), SELECT(), SET RELATION, USE, WORKAREA()

SELECT()

Returns the number of an available work area or the work area number associated with a specified alias.

Syntax SELECT([<alias>])

<alias> The work area to return. (If <alias> is a work area number, there is no need to call this function, because that number is what the function will return.)

Description If you do not specify an alias, SELECT() returns the number of the next available work area, a number between 1 and 225; or zero if all work areas in the current workset are being used. If you specify an alias, SELECT() determines whether the specified alias name is already in use. If it is, SELECT() returns the work area number that's using the alias name; otherwise it returns zero.

Use SELECT() to locate an available work area in which to open a table, or to see if a table is already open so that you don't open it again.

OODML There is no concept of the "current" Query object. Use your usual object management techniques to manage Query objects.

Example It's common practice to use SELECT() to open a table in the next available work area. This way, you don't have to worry about accidentally closing an open table. You then always use the alias name to refer to that table. For example:

```
use CUSTOMER in select() order CUST_NAME
use ORDERS   in select() order CUST_ORD
use LINEITEM in select() order ORD_LINE
select CUSTOMER
```

See Also ALIAS(), DBF(), SELECT, WORKAREA()

SET AUTOSAVE

Determines if dBASE writes data to disk each time a record is changed or added.

Syntax SET AUTOSAVE on | OFF

Description Use SET AUTOSAVE ON to reduce the chances of data loss. When SET AUTOSAVE is ON and you alter or add a record, *Visual* dBASE updates tables and index files on disk when you move the record pointer. When SET AUTOSAVE is OFF, changes are saved to disk as the record buffer is filled.

Since *Visual* dBASE periodically saves table changes to disk, in most situations you don't need to SET AUTOSAVE ON. SET AUTOSAVE OFF lets you process data faster, since *Visual* dBASE writes your changes to disk less often.

OODML AUTOSAVE is always OFF. To force data to be written to disk, call the Rowset object's *flush*() method in the *onSave* event.

See Also CLOSE TABLES, FLUSH

SET DATABASE

Sets the default database from which tables are accessed.

Syntax SET DATABASE TO [<database name>]

<database name> Specifies the name of the database you want to make the current database.

Description SET DATABASE sets the current database, which defines the default location for tables accessed by *Visual* dBASE commands. Using this command, you can select from any databases previously opened with the OPEN DATABASE command. Databases are defined using the BDE Administrator.

When you issue the SET DATABASE TO command without a database, *Visual* dBASE restores operation to accessing tables in the current directory (or in the directory specified by SET PATH) instead of through a database.

OODML Assign the appropriate Database object to the Query object's *database* property.

See Also CLOSE..., DATABASE(), OPEN DATABASE, SET DBTYPE

SET DBTYPE

Sets the default table type to either Paradox or dBASE.

Syntax SET DBTYPE TO [PARADOX | DBASE]

PARADOX | DBASE Sets the default table type to a Paradox (DB) or dBASE (DBF) table. The default is DBASE.

Description SET DBTYPE sets the default type of table used by commands that open or create a table. You can override this selection by specifying a specific extension, that is, .DBF for a dBASE table or .DB for a Paradox table; or by using the TYPE option offered by many commands.

SET DBTYPE TO specified without a DBASE or PARADOX argument returns DBTYPE to its default (dBASE).

OODML No equivalent.

See Also CLOSE..., COPY TABLE, CREATE, DATABASE(), DELETE TABLE, MODIFY STRUCTURE, OPEN DATABASE, RENAME TABLE, SET DATABASE, USE

SET DELETED

Controls whether *Visual* dBASE hides records marked as deleted in a DBF table.

Syntax SET DELETED ON | off

Description Use SET DELETED to include or exclude records marked as deleted in a DBF table. When SET DELETED is OFF, all records appear in a table. When SET DELETED is ON, *Visual* dBASE excludes records that have been marked as deleted, hiding them from most operations.

INDEX, REINDEX, and RECCOUNT() aren't affected by SET DELETED.

If two tables are related with SET RELATION, SET DELETED ON suppresses the display of deleted records in the child table. The related record in the parent table still appears, however, unless the parent record is also deleted.

OODML Soft deletes are not supported.

See Also DELETE, DELETED(), PACK, RECALL, SET(), SET FILTER, SET RELATION

SET EXACT

Establishes the rules used to determine whether two character strings are equal.

Syntax SET EXACT on | OFF

Description Use SET EXACT to choose between a partial string match and an exact string match for certain Xbase DML commands, the Array class *scan()* method, and the = comparison operator. The == comparison operator always behaves like SET EXACT is ON.

When SET EXACT is OFF, partial string matches are performed. For example, SEEK("S") will find "Smith" in an index, and "Smith"="S" evaluates to *true*.

When SET EXACT is ON, the two strings must match exactly, except that trailing blanks are ignored. For example, SEEK("Smith") will find "Smith " in an index and "Smith"="Smith " will evaluate to *true*.

A partial string match can be thought of as a "begins with" check. For example, the SEEK() function searches for an index key value that begins with certain characters, and the = operator checks to see if one string begins with another string.

In language drivers that have *primary* and *secondary weights* for characters (not U.S. language drivers but most others), Visual dBASE compares characters by their primary weights when SET EXACT is OFF and by their secondary weights when SET EXACT is ON. For example, when SET EXACT is OFF, and the current language driver is German, "drücken" and "drucken" are equal.

See Also SET NEAR

SET EXCLUSIVE

Controls whether Visual dBASE opens tables and their associated index and memo files in exclusive or shared mode.

Syntax SET EXCLUSIVE on | OFF

Description When you issue SET EXCLUSIVE ON, subsequent tables you open—and their associated indexes and memos—are in *exclusive mode*, unless you open them with USE...SHARED. When you open a table in exclusive mode, other users can't open, view, or change the file or any of its associated index and memo files. If you try to open a table that another user has opened in exclusive mode, or if you try to open in exclusive mode a table that another user has opened, an error occurs.

Exclusive use of a table is different than a file lock that you would get with FLOCK(). With a file lock, others may have the table open and can read data,

although only one user may have a file lock at any time. With exclusive use, no one else can have the table open.

SET EXCLUSIVE OFF causes subsequent tables you open—and their associated indexes and memos—to be in *shared mode*, unless you open them with USE...EXCLUSIVE. If a table in shared mode is in a shared network directory, other users on the network with access to the directory can open, view, and change the file and any of its associated index and memo files.

If you use SET INDEX and the table is open in exclusive mode, *Visual dBASE* opens the index in exclusive mode. If the table is open in shared mode by way of an overriding USE...SHARED, *Visual dBASE* opens the index in the mode specified by USE.

An index created with INDEX is opened in exclusive mode, regardless of whether the table is opened in shared or exclusive mode and regardless of the SET EXCLUSIVE setting. After creating an index, you can open the index in shared mode with USE...INDEX...SHARED or by issuing SET EXCLUSIVE OFF followed by SET INDEX TO.

The following commands require the exclusive use of a table with either SET EXCLUSIVE ON or USE...EXCLUSIVE:

- CONVERT
- DELETE TAG
- INDEX...TAG
- MODIFY STRUCTURE
- PACK
- REINDEX
- ZAP

OODML EXCLUSIVE is always OFF. When a method like *packTable()* requires exclusive access to a table, the method always attempts to open the table in exclusive mode. If the table is already open in another query, the method will fail.

See Also FLOCK(), INDEX, RLOCK(), SET INDEX, SET LOCK, USE

SET FIELDS

Defines the list of fields a table appears to have.

Syntax SET FIELDS TO
[<field list> | ALL [LIKE <skeleton 1>] [EXCEPT <skeleton 2>]]

SET FIELDS on | OFF

<field list> | ALL [LIKE <skeleton 1> | EXCEPT <skeleton 2>] Adds the specified fields to the list of fields the table appears to have. The fields list may include fields from tables open in all work areas and may also include read-only calculated fields. The following table provides a description of SET FIELDS TO options:

| Option | Description |
|--|--|
| ALL | Adds all fields in the current work area to the field list |
| LIKE < <i>skeleton 1</i> > | Adds all fields in the current work area whose names match < <i>skeleton 1</i> > to the field list |
| EXCEPT < <i>skeleton 2</i> > | Adds all fields in the current work area except those whose names match < <i>skeleton 2</i> > to the field list |
| LIKE < <i>skeleton 1</i> > EXCEPT < <i>skeleton 2</i> > | Adds all fields in the current work area whose names are like < <i>skeleton 1</i> > except those whose names match < <i>skeleton 2</i> > to the field list |

Description

When there is no field list, or SET FIELDS is OFF, operations in a work area that use all fields (by default) use all the fields in the currently selected table. For example, if you LIST a Customer table with 10 fields and 500 records, those 10 fields are displayed.

A field list overrides this default behavior, making the table appear to have the fields you specify. This is usually done to either:

- Restrict the fields to certain fields in the table. For example, you can make the Customer table appear to have only a customer ID and name, hiding the other 8 fields.
- Include fields in other related tables. For example, you could set a relation to a table of sales people, and make the Customer table appear to have the customer ID, customer name, and the name of their account representative.

When a field list is active, it is the field list for *all work areas* in the workset. Because the fields in a field list always contain the full alias and field name, the fields in the field list will always be used, even if those fields are in another, unrelated work area. For example, suppose you create a field list with fields from the Customer table, and then select the Vendor, which has 90 records and is not related into the Customer table. If you then issue LIST, you will see 90 records, because the LIST command works on the current work area, but you will see the fields from the Customer table—the fields from the same record repeated 90 times, because the two tables are not related, and those are the values of the named fields for each record in the Vendor table.

Therefore, when you create a field list, it is usually used only for the work area in which it is created. If the field list contains fields from other work areas, some way of synchronizing the movement of the record pointers in those work areas, usually with SET RELATION, is required.

If there is no field list, SET FIELDS creates the specified field list and activates it. If there is a field list, whether it's active or not, SET FIELDS adds the specified fields to the field list, and activates it. Fields in other work areas that are added to the field list may be referred to by their field name only, without using an alias; those fields now appear to be fields in the current work area. The alias is still allowed, and is necessary if you have two fields with the same name from different tables in the field list.

A field may be added to the field list more than once, although this is not recommended. For example, if you execute SET FIELDS TO ALL twice, you will

see all the fields in the current table twice. Be sure to use CLEAR FIELDS before issuing SET FIELDS if your intent is to create a new field list, not add to an existing field list. To specify a field in other work areas, prefix the field name with the alias name and alias operator (that is, *alias->field*).

You can temporarily disable the field list with SET FIELDS OFF. To reactivate the field list, use SET FIELDS ON. Adding fields with SET FIELDS always reactivates the field list. If you SET FIELDS ON without using the SET FIELDS TO *<field list>* command, no fields are accessible. SET FIELDS TO with no fields has the same effect as CLEAR FIELDS, deactivating and clearing the field list.

Some commands have a FIELDS option, or some way of specifying fields. You may further restrict the fields used with this option, but you cannot reference fields that have been hidden because they have not been included with SET FIELDS.

When a field list is active, fields that are not on the field list cannot be used in expressions. However, some commands ignore the field list, including:

- INDEX and index expressions
- LOCATE
- SET FILTER
- SET RELATION

The fields list specified with SET FIELDS TO can include both table field names and calculated fields. The /R option provides a setting to specify read-only access to table fields, for example:

```
salary/R, hours/R
```

To specify a calculated field, you can specify any valid expression. For example,

```
gross_pay = salary * hours
```

OODML No direct equivalent. When accessing the *fields* array, you may include program logic to include or exclude specific fields.

Example This example takes a table of customers and a table of sales persons and creates a table that contains only the customer name and the name of their sales representative.

```
use CUSTOMER in select()
use SALESPER in select() order SALES_ID
select customer
set relation to SALES_ID into SALESPER
set fields to CUST_NAME = CUSTOMER->NAME, SALES_PERS = SALESPER->NAME
copy to CUSTSALES
```

The field list contains two calculated fields that are used simply to assign new names to the fields from two different tables that happen to have the same name.

Although the COPY command has a FIELDS option, it does not support calculated fields. Therefore SET FIELDS is required for this operation.

See Also CLEAR FIELDS, FLDLIST(), SET(), SET RELATION

SET FILTER

Hides records based on a logical condition.

Syntax SET FILTER TO [*<condition>*]

<condition> The condition that records must meet to be seen.

Description A filter is a mechanism by which you can temporarily hide, or filter out, those records that do not match certain criteria so that you can see only those records that do match. The criteria is expressed as a logical expression, for example,

```
set filter to upper( FIRST_NAME ) == "WALDO"
```

In this case, you would see only those records in the current table whose First_name field was “Waldo” (capitalized in any way).

The filter does not take effect until some sort of record navigation is attempted. For example, any command with an ALL scope will attempt to start at the first record. In this case, the command will start at the first record that matches the filter condition, and process all matching records. A SKIP command would attempt to navigate to the next matching record, and SKIP -1 would attempt to navigate to the previous matching record. GO TOP is often used after SET FILTER to position the record pointer on the first matching record.

Any time you attempt to navigate to a record, the record is evaluated to see if it matches the filter condition. If it does, then the record pointer is allowed to position itself at that record and the record can be seen. If the record does not match the filter condition, the record pointer continues in the direction it was moving to find the next matching record. It will continue to move in that direction until it finds a match or gets to the end-of-file. For example, suppose you issue:

```
skip 4
```

If no filter is active, you would move four records forward. If a filter is active, the records pointer will move forward until it has encountered four records that match the filter condition, and stop at the fourth. That may be the next four records in the table, if they all happen to match, or the next five, or the next 400, or never, if there aren't four records after the current record that match. In that last case, the record pointer will be at the end-of-file.

In other words, when there is no filter active, every record is considered a match. By setting a filter, you filter out all the records that don't match certain criteria.

Note You cannot use the special variables *this* or *form* in the *<condition>*. This is explicitly prohibited because these special variables automatically take on the value of whatever object and form has focus (or fires an event) at any given moment. Therefore, the filter condition will vary and quite likely be invalid.

Generally speaking, you should not use variables in a filter condition at all, because the variables may go out of scope, making the filter condition an invalid expression. To solve these problems, use macro substitution, as shown in the example.

Many commands have scope option that includes FOR and WHILE conditions. These conditions are applied in addition to the filter condition.

SET FILTER applies to the current work area. Each work area may have its own filter condition. To disable the filter condition, issue SET FILTER TO with no options.

OODML Use the rowset object's *beginFilter()* and *applyFilter()* methods.

Example The following example sets a filter based on the state that is chosen from a combobox on a form:

```
PROCEDURE setFilter_onClick
    private cState
    cState = form.stateCombobox.value
    set filter to STATE == "&cState"
```

Note the use of macro substitution inside a literal string. A private variable is used; you cannot use the macro operator on a local variable. For example, if the variable contains the value "CA", then the macro substitution would evaluate to:

```
set filter to STATE == "CA"
```

See Also SET(), SET DELETED, SET KEY

SET HEADINGS

Controls the display of field names in the output of DISPLAY and LIST.

Syntax SET HEADINGS ON | off

Description When SET HEADINGS is ON, the output of DISPLAY and LIST includes a heading identifying the fields of the table(s). Issue SET HEADINGS OFF before issuing DISPLAY or LIST to view the output without field-name headings.

OODML No equivalent.

See Also DISPLAY, LIST

SET INDEX

Opens index files for the current DBF table.

Syntax SET INDEX TO [<filename list> [ORDER [TAG]
 <ndx filename> | <tag name> [OF <mdx filename>]]]

<filename list> Specifies the index files to open, including both .NDX and .MDX indexes. The default extension is .MDX.

ORDER [TAG] <ndx filename> | <tag name> Specifies a master index, which can be an .NDX file or a tag name contained within an .MDX index file. The TAG keyword is for readability only; it has no effect on the command.

OF <mdx filename> Specifies a multiple index file containing <tag name>. The default extension is .MDX.

Description Use SET INDEX to open the specified .NDX and .MDX files in the current work area. Open index files are updated when changes are made to the associated table. Including an index file list when issuing USE...INDEX is equivalent to following the USE command with the SET INDEX command.

If the first index opened with SET INDEX is an .NDX file, that index becomes the master index unless you specify another master index with the ORDER option or the SET ORDER command. If the first index opened with SET INDEX is an .MDX file and you don't specify the ORDER clause, no master index is defined, and records in the table appear in record number or *natural* order. To specify a master index for the current table, specify the ORDER option or use the SET ORDER command.

Before opening the indexes specified with the command, SET INDEX closes all open index files except the production .MDX file, the index file with the same name as the current table. Specifying SET INDEX TO without a list of indexes closes all open .NDX and .MDX files in the current work area, except for the production index file. You can also use the CLOSE INDEX command. All indexes, including the production .MDX file, are closed when you close the table.

The order in which you specify indexes with the SET INDEX command isn't necessarily the same as the order *Visual* dBASE uses for functions like TAG(). Open indexes for a specified work area are ordered as follows:

- 1 All .NDX index files in the order you list them in <filename list>.
- 2 All tags in the production .MDX file, in the order they were created. The tags are listed in order by the DISPLAY STATUS command.
- 3 All tags in other open .MDX files.

The order of the open indexes remains the same until you specify another index order with the USE...INDEX or SET INDEX commands, or you issue an INDEX command.

OODML Assign to the Rowset object's *indexName* property.

See Also CLOSE INDEXES, INDEX, KEY(), MDX(), NDX(), ORDER(), REINDEX, SET ORDER, TAG(), TAGNO(), TAGCOUNT(), USE

SET KEY TO

Constrains the records in a table to those whose key field values falls within a range.

Syntax SET KEY TO
 [<exp1> | <exp list 1> |
 RANGE
 <exp2> [,] | ,<exp3> | <exp2>, <exp3>
 [EXCLUDE] |
 LOW <exp list 2>] [HIGH <exp list 3>
 [EXCLUDE]]
 [IN <alias>]

<exp1> Shows only those records whose key value matches <exp 1>.

<exp list 1> For tables indexed on a composite (multi-field) key index, shows only those records whose key field values match the corresponding values in <exp list 1>, separated by commas.

RANGE <exp2> [,] | ,<exp3> | <exp2>, <exp3>

LOW <exp list 2> HIGH <exp list 3> Shows only those records whose key values fall within a range. Use RANGE for single key values and LOW/HIGH for composite keys. You may use either the RANGE clause or LOW/HIGH, but not both in the same command. The following table summarizes how SET KEY constrains records in the master index:

| Option | Description |
|--|--|
| RANGE <exp2> [,] LOW <exp list 2> | Shows only those records whose key values are greater than or equal to <exp2> / <exp list 2> |
| RANGE , <exp3> HIGH <exp list 3> | Shows only those records whose key values are less than or equal to <exp3> / <exp list 3> |
| RANGE <exp2>, <exp3> LOW <exp list 2> HIGH <exp list 3> | Shows only those records whose key values are greater than or equal to <exp2> / <exp list 2> and less than or equal to <exp3> / <exp list 3> |

EXCLUDE Excludes records whose key values are equal to <exp2> / <exp list 2> or <exp3> / <exp list 3>. If omitted, these records are included in the range.

IN <alias> The work area in which to set the key constraint.

Description SET KEY TO is similar to SET FILTER; SET KEY TO uses the table's current master index and shows only those records whose key value matches a single value or falls within a range of values. This is referred to as a *key constraint*. Because it uses an index, a key constraint is instantaneous, while a filter condition must be evaluated for each record. SET KEY TO with no arguments removes any range of key values previously established for the current table with SET KEY TO.

The key range values must match the key expression of the master index. For example, if the index key is UPPER(Name), specify uppercase letters in the range expressions. In determining whether the specified range expressions match key field expressions, SET KEY TO follows the rules established by SET EXACT. The SET KEY TO range takes effect after you move the record pointer.

When you issue both SET KEY and SET FILTER for the same table, *Visual* dBASE processes only records that are within the SET KEY index range and that also meet the SET FILTER condition.

OODML Use the Rowset object's *setRange()* method.

See Also INDEX, KEY(), MDX(), NDX(), TAG(), SET FILTER

SET LOCK

Determines whether *Visual* dBASE attempts to lock a shared table during execution of certain commands that read the table but don't change its data.

Syntax SET LOCK ON | off

Description Issue SET LOCK OFF to disable automatic file locking for certain commands that only read a table. This lets other users change data in the file while you access it with read-only commands. For example, you might want to issue SET LOCK OFF before using AVERAGE if you don't expect other users to alter the data in the table you're using significantly. Or, you might want to issue SET LOCK OFF before processing a range of records that other users aren't going to update.

The following commands automatically lock tables when SET LOCK is ON and don't lock tables when SET LOCK is OFF:

- AVERAGE
- CALCULATE
- COPY (source file)
- COPY MEMO
- COPY STRUCTURE
- COPY TO ARRAY
- COPY STRUCTURE [EXTENDED] (source file)
- COUNT
- SORT (source file)
- SUM
- TOTAL (source file)

Visual dBASE continues to lock records and tables automatically for commands that let you change data whether SET LOCK is ON or OFF.

OODML This setting is not applicable.

See Also FLOCK(), RLOCK()

SET MEMOWIDTH

Sets the width of memo field display or output.

Syntax SET MEMOWIDTH TO [*<expN>*]

<expN> Specifies a number from 8 to 255 that sets the width of memo field display and output. The default is 50.

Description Use SET MEMOWIDTH to change the column width of memo fields during display and output, and the default column width for the MEMLINES() and MLINE() functions. Memo fields can be displayed using the commands DISPLAY, LIST, ?, or ?. SET MEMOWIDTH doesn't affect the display of a memo field in an Editor control. If the system memory variable `_wrap` is set to *true*, the system memory variables `_lmargin` and `_rmargin` determine the memo width.

The @V (vertical stretch) picture function causes memo fields to be displayed in a vertical column when `_wrap` is *true*. When @V is specified, the `_pcolno` system memory variable is incremented by the @V value. This lets you change the appearance of the printed output of ? or ?? commands by using the @V function. When @V is equal to zero, memo fields wrap within the SET MEMOWIDTH width.

OODML This setting is not applicable.

See Also ?, ??, DISPLAY, LIST, MEMLINES(), MLINE(), SET(), `_lmargin`, `_rmargin`, `_wrap`

SET NEAR

Specifies where to move the record pointer after a SEEK or SEEK() operation fails to find an exact match.

Syntax SET NEAR on | OFF

Description Use SET NEAR to position the record pointer in an indexed table close to a particular key value when a search does not find an exact match. When SET NEAR is ON, the record pointer is set to the record closest to the key expression searched for but not found with SEEK or SEEK(). When SET NEAR is OFF and a search is unsuccessful, the record pointer is positioned at the end-of-file.

The closest record is the the record whose key value follows the value searched for in the index order, toward the end-of-file. When SET DELETED is ON or a filter is set with the SET FILTER command, SET NEAR skips over deleted or filtered-out records in determining the record nearest the key value expression.

The record pointer will be at the end-of-file if search value comes after the key value of the last record in the index order.

With SET NEAR ON, FOUND() and SEEK() return *true* for an exact match or *false* for a near match. With SET NEAR OFF, FOUND() and SEEK() return *false* if no match occurs.

OODML Use either the *findKey()* or *findKeyNearest()* method of the Rowset object.

See Also EOF(), FOUND(), SEEK, SEEK(), SET DELETED, SET FILTER

SET ODOMETER

Specifies how frequently *Visual* dBASE updates and displays record counter information for certain commands in the status bar.

Syntax SET ODOMETER TO [*<expN>*]

<expN> The interval at which *Visual* dBASE updates the record counter. *<expN>* must be at least 1 and is truncated to an integer. If omitted, the default value, 100, is used.

Description Use SET ODOMETER to specify how frequently *Visual* dBASE updates and displays record counter information during the execution of certain commands, such as AVERAGE, CALCULATE, COUNT, DELETE, GENERATE, INDEX, RECALL, SUM, and TOTAL. If the status bar is not enabled, SET ODOMETER has no effect. The status bar is enabled through *_app.statusBar*.

If SET TALK is OFF, dBASE does not display any record counter information in the status bar, regardless of the SET ODOMETER setting. If SET TALK is ON, use SET ODOMETER with a relatively high value to improve performance.

OODML Use the Session object's *onProgress* event.

SET ORDER

Specifies an open index file or tag as the master index of a table.

Syntax SET ORDER TO [[TAG] *<tag name>*] [OF *<mdx>*] [NOSAVE]]

[TAG] <tag name> The name of an index tag in an open .MDX file or the name of an open .NDX file (without the file extension). The TAG keyword is included for readability only; TAG has no affect on the operation of the command.

OF <mdx> The open .MDX file that contains *<tag name>*. Use this option when two open .MDX files have a tag with the same name. The default extension is .MDX.

If you use the *<tag name>* option but don't specify *<mdx>*, *Visual* dBASE searches for the named index in the list of open indexes.

NOSAVE Used to delete a temporary index after the associated table is closed. If you decide after choosing this option that you want to keep the index, open the index again using SET ORDER without the NOSAVE option, before you close the table.

Description Use SET ORDER to change the master index of a table without having to close and reopen indexes. You can choose the master index from the list of .NDX files or .MDX index tags opened with the SET INDEX or USE...INDEX commands.

If you specify SET ORDER without specifying an index, the table appears in primary key order, if the table has a primary key; or unindexed, in record number order.

OODML Assign the tag name to the Rowset object's *indexName* property.

See Also CLOSE INDEXES, INDEX, KEY(), MDX(), NDX(), ORDER(), REINDEX, SET INDEX, TAG(), TAGCOUNT(), TAGNO(), USE

SET REFRESH

Determines how often *Visual* dBASE refreshes the workstation screen with table information from the server.

Syntax SET REFRESH TO *<expN>*

<expN> A time interval expressed in seconds from 0 to 3,600 (1 hour), inclusive. The default is 0, meaning that *Visual* dBASE doesn't update the screen.

Description Use SET REFRESH to set a refresh interval when working with shared tables on a network. Then, when you use BROWSE or EDIT to edit shared tables, your screen refreshes at the interval you set, showing you changes made by other users on the network to the same tables.

If another user has a lock on the file or records you're currently viewing, the file or records won't be refreshed until that user releases the lock.

OODML Use a Timer object to periodically call the Rowset object's *refreshControls()* method.

See Also BROWSE, EDIT, FLOCK(), REFRESH, RLOCK()

SET RELATION

Links two or more open tables with common fields or expressions.

Syntax SET RELATION TO
 [<key exp list 1> INTO <alias 1> [CONSTRAIN]
 [, <key exp list 2> INTO <alias 2> [CONSTRAIN]] ...
 [ADDITIVE]]

<key exp list 1> The key expression or field list that is common to both the current table and a child table and links both tables. The child table must be indexed on the key field and that index must be the master index in use for the child table.

INTO <alias> <alias> specifies the child table linked to the current table.

<key exp list 2> INTO <alias 2> ...] Specifies additional relationships from the current table into other tables.

CONSTRAIN Limits records processed in the child table to those matching the key expression in the parent table.

ADDITIVE Adds the new relation to any existing ones. Without ADDITIVE, SET RELATION clears existing relations before establishing the new relation.

Description Use SET RELATION to establish a link between open tables based on common fields or expressions.

Before setting a relation, open each table in a separate work area. When a relation is set, the table in the current work area is referred to as the *parent* table, and a table linked to the parent table by the specified key is called a *child* table. The child table must be indexed on the fields or expressions that link tables and that index must be the master index in use for the child table.

A relation between tables is usually set through common keys specified by <key exp list>. The relating expression can be any expression derived from the parent table that matches the keys of the child table master index. The keys may be a single field or a set of concatenated fields contained in each table. The fields in each table can have different names but must contain the same type of data. For Paradox and SQL tables, you can specify single or composite index key fields.

SET RELATION clears existing relations before establishing a new one, unless you use the ADDITIVE option. SET RELATION TO without any arguments clears existing relations from the current table without establishing any new relations.

The CONSTRAIN option restricts access in the child table to only those records whose key values match records in a parent table. This is the same as using SET KEY TO on the key field of the child table. As a result, you can't use SET KEY TO and CONSTRAIN at the same time. If a SET KEY TO operation is in effect on the child table when you specify CONSTRAIN with SET RELATION, *Visual dBASE* returns a "SET KEY active in alias" message. If the CONSTRAIN option is in effect when SET KEY TO is specified, *Visual dBASE* returns the error "Relation using CONSTRAIN." You can use SET FILTER with the CONSTRAIN option, if you want to specify additional conditions to qualify records in a child table.

More than one relation can be defined from the same table. Also, more than one relation can be set from the same parent table if you use the ADDITIVE option or if you specify multiple relations with the same SET RELATION command. You can also establish additional relations from a child table, thus defining a chain of relations. Cyclic relations aren't allowed; that is, *Visual dBASE* returns an error if you attempt to define a relation from a child table back into its parent table.

When a relation is set from a parent table to a child table, the relation can be accessed only from the work area that contains the parent table. To access fields of the child table from the current work area, use the alias operator(->) and prefix the name of fields in the child table by its alias name.

If a matching record can't be found in a linked table, the linked table is positioned at the end-of-file, and EOF() in the child alias returns *true* while FOUND() returns *false*. The setting of SET NEAR does not affect positioning of the record pointer in child tables.

When a SET SKIP list is active, the record pointer is advanced in each table, starting with the last work area in the relation chain and moving up the chain toward the parent table.

OODML Use the Rowset object's *masterFields* and *masterRowset* properties, or the Query object's *masterSource* property.

Example The first example links a table of reviews to a table of authors. This is a one-to-one link that can be used to get the name of the lead author (stored in the Authors table) for each review:

```
use REVIEWS                                && Open in current work area
use AUTHORS in select() order AUTH_ID      && Open in next available work area
set relation to LEAD_AUTH into AUTHORS     && Link Reviews to Authors
```

The Reviews table has a Lead_auth field that contains the ID of the lead author for each review. The Authors table identifies each author with an ID field named Auth_id. The Auth_id field is indexed by itself, so the index has the same name.

The following example is a one-to-many link between a customer table and an orders table that shows only those customers that have made orders in the past:

```
use CUSTOMER
use ORDERS in select() order CUST_DATE
set relation to CUST_ID into ORDERS
set filter to found( "ORDERS" )
```

The Customer table has a Cust_id field that contains the customer ID. The same field is a foreign key in the Orders table. The Cust_date index is an expression index created with:

```
index on CUST_ID + dtos( ORDER_DATE ) tag CUST_DATE
```

The SET FILTER command shows only those Customers that have a record in the Orders table.

See Also FOUND(), SELECT, SET SKIP, SKIP

SET REPROCESS

Specifies the number of times *Visual* dBASE tries to lock a file or record before generating an error or returning *false*.

Syntax SET REPROCESS TO <expN>

<expN> A number from -1 to 32,000, inclusive, that is the number of times for *Visual* dBASE to try get a lock. The default is 0; both 0 and -1 have a special meaning, described below.

Description Use SET REPROCESS to specify how many times *Visual* dBASE should try to get a lock before giving up. RLOCK() and FLOCK() return *false* if the lock attempt fails. For automatic locks, failure to get a lock causes an error. SET REPROCESS affects RLOCK(), and FLOCK(), and all commands and functions that automatically attempt to lock a file or records.

SET REPROCESS TO 0 causes *Visual* dBASE to display a dialog that gives you the option of cancelling while it indefinitely attempts to get the lock.

Setting SET REPROCESS to a number greater than 0 causes *Visual* dBASE to retry getting a lock the specified number of times without prompting.

SET REPROCESS TO -1 causes *Visual* dBASE to retry getting a lock indefinitely, without prompting.

OODML Set the Session object's *lockRetryCount* property.

See Also FLOCK(), ON ERROR, ON NETERROR, RETRY, RLOCK(), SET LOCK

SET SAFETY

Determines whether *Visual* dBASE asks for confirmation before overwriting a file or removing records from a table when you issue ZAP.

Syntax SET SAFETY ON | off

Description When SET SAFETY is ON, dBASE prompts for confirmation before overwriting a file or removing records from a table when you issue ZAP. If you want your application to control the interaction between *Visual* dBASE and the user with regard to overwriting files, issue SET SAFETY OFF in your program.

SET SAFETY affects the following commands:

- Commands using a TO FILE option
- COPY
- COPY FILE

- COPY STRUCTURE [EXTENDED]
- CREATE/MODIFY commands
- INDEX
- SAVE
- SET ALTERNATE TO
- SORT
- TOTAL
- UPDATE
- ZAP

Note SET TALK OFF does not suppress SET SAFETY warnings.

OODML SAFETY is always OFF.

See Also SET TALK

SET SKIP

Specifies how to advance the record pointer through records of linked tables.

Syntax SET SKIP TO [*<alias1>* [, *<alias2>*]...]
<alias1> [, *<alias2>*] ... The work areas defined in the relation.

Description SET SKIP works only with tables that have been linked with the SET RELATION command. Used together, the SET RELATION and SET SKIP commands determine the way in which the record pointer moves through parent and child tables.

Use SET SKIP when you set a relation from a parent table containing unique key values to child tables that contain duplicate key values, that is, a one-to-many relationship. SET SKIP causes commands that move the record pointer to move the pointer to every record with matching key values in a child table before moving the record pointer in the parent table. If you define a chain of relations and use SET SKIP to move from one table to the next down the chain, the record pointer moves to every record in the last child table before the pointer moves in its parent table.

You do not need to specify the root (parent) alias in the SET SKIP list. SET SKIP TO with no options cancels any previously defined SET SKIP behavior.

OODML Override the *next()* method of the detail table. For example:

```
function next( nArg )
  if argcount() < 1
    nArg := 1                                // Skip one row forward by default
  endif
  if not rowset::next( nArg )                 // Navigate as far as specified, but
                                              // if end of detail rowset
    this.masterRowset.next( sign( nArg ) )    // Move forward or backward in master
    if nArg < 0                               // If navigating backwards
```

SET UNIQUE

```
        this.last()                                // Go to last matching detail row
    endif
endif
```

Then navigate by calling *next()* in detail rowset—not the master rowset, as you would with SET SKIP.

See Also SET RELATION, SKIP

SET UNIQUE

Determines if unique indexes are always created.

Syntax SET UNIQUE on | OFF

Description When SET UNIQUE is ON, the INDEX command always creates the index as if the UNIQUE option is specified. The UNIQUE option has different meanings for different table types. For DBF tables, it allows records with duplicate key values to be stored in the table, but only shows the first record with that key value.

For DB and SQL tables, it prevents records with duplicate key values from being stored in the table; attempting to do so causes a key violation error. This type of index is referred to as a distinct index. You can create the same kind of index for DBF tables by using the DISTINCT option.

Whenever you reindex an index file, *Visual* dBASE maintains the index in the same way it was created. For more information on unique and distinct indexes, see the INDEX command.

OODML No equivalent.

See Also INDEX, REINDEX, SET(), SET INDEX, SET ORDER, UNIQUE(), USE

SET VIEW

Opens a previously defined query or view file.

Syntax SET VIEW TO <filename>

<filename> The query or view file containing the commands to define the current working environment or view. If you specify a file without including its extension, *Visual* dBASE looks for a .QBE, then a .VUE file.

Description Use SET VIEW to change the working environment of the current workset to one that was previously defined by CREATE QUERY or CREATE VIEW. The working environment includes open tables and index files, all relations, the active fields list, and filter conditions.

OODML Use a DataModule object.

See Also CREATE QUERY, CREATE VIEW

SKIP

Moves the record pointer in the current or specified work area.

Syntax SKIP [*<expN>*] [IN *<alias>*]

<expN> The number of records *Visual* dBASE moves the record pointer forward or backward in the table open in the current or specified work area. If *<expN>* evaluates to a negative number, the record pointer moves backward. SKIP with no *<expN>* argument moves the record pointer forward one record.

IN *<alias>* The work area in which to move the record pointer.

Description Use SKIP to move the record pointer relative to its current position, in the current index order, if any.

If you issue a SKIP command when the record pointer is at the last record in a table, the record pointer moves to the end-of-file; EOF() returns *true*. Issuing any additional SKIP commands (that move forward) causes an error. Similarly, if you issue a SKIP -1 command when the record pointer is at the first record of a file, BOF() returns *true*, and a subsequent negative SKIP command cause an error.

SKIP IN *<alias>* lets you advance the record pointer in another work area without selecting that work area first with the SELECT command.

If you are using SKIP in a loop to visit all the records in a table, consider using a SCAN loop instead.

OODML Call the *next()* method of the desired Rowset object.

See Also BOF(), EOF(), GO, SCAN, SET SKIP

SORT

Copies the current table to a new table, arranging records in the specified order.

Syntax SORT TO *<filename>* [[TYPE] PARADOX | DBASE]
 ON *<field 1>* [/A | /D [/C]]
 [, *<field 2>* [/A | /D [/C]]...]
 [*<scope>*]
 [FOR *<condition 1>*]
 [WHILE *<condition 2>*]
 [ASCENDING | DESCENDING]

<filename> The new table file to copy and sort the current table's records to.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if *<filename>* does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

ON <field 1> Makes *<field 1>* the first field of *<filename>* and sorts *<filename>* records by the values in *<field 1>*, which can be any data type except binary, memo, or OLE.

/A Sorts records in ascending order (A to Z; 1 to 9; past to future (blank dates come after non-blank dates); *false* then *true*). Since this is the default sort order, include /A for readability only.

/D Sorts records in descending order.

/C Removes the distinction between uppercase and lowercase letters. When you specify both A and C, or both D and C, use only one forward slash (for example, /DC).

<field 2> [/A | /D [/C]] ... Sorts on a second field so that the new table is ordered first according to *<field 1>*, then, for identical values of *<field 1>*, according to *<field 2>*. If a third field is specified, records with identical values in *<field 1>* and in *<field 2>* are then sorted according to *<field 3>*. The sorting continues in this way for as many fields as are specified.

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

ASCENDING Sorts all specified fields for which you don't include a sort order in ascending order. Since this is the default, include ASCENDING for readability only.

DESCENDING Sorts all specified fields for which you don't include a sort order in descending order.

Description The SORT command creates a new table in which the records in the current table are positioned in the order of the specified key fields.

When you use SORT, creates a temporary index file. During the sorting process, your disk must have space for this temporary index file and the new table file.

SORT differs from INDEX in that it creates a new table rather than provide an index to the original table. Although using SORT is generally not as efficient as using an index to organize tables, you might want to use SORT for the following applications:

- To archive an outdated table and store it in a sorted order
- To create a table that is a sorted subset of an original table
- To maintain a small table that needs to be sorted in only one order

- To create an ordered table where record numbers are sequential and contiguous

OODML No equivalent.

Example Suppose you have a large table and you want to delete any duplicate records, records that have the same value in 6 important fields. The easiest way to remove duplicate records in general is to index the table so that all duplicate records are next to each other. Unfortunately, some of the 6 important fields are large; the resulting the index key would be larger than the allowed limit, 100 characters. So you sort the table instead to a temporary table instead.

```
use THETABLE
sort to SORTTEMP on FIELD1, FIELD2, FIELD3, FIELD4, FIELD5, FIELD6
use SORTTEMP
set fields to KEY = FIELD1 + FIELD2 + FIELD3 + FIELD4 + FIELD5 + FIELD6
local cKey
do while .not. eof()
    cKey = KEY
    skip
    delete while KEY == cKey
enddo
clear fields
```

SET FIELDS is used to create a temporary calculated field to make it easier to compare the important field values for each record.

See Also INDEX

STORE AUTOMEM

Stores the contents of all the current record's fields to a set of memory variables.

Syntax STORE AUTOMEM

Description STORE AUTOMEM copies every field of the current record to a set of matching automem variables. Each memory variable has the same name, length, and data type as one of the fields. *Visual* dBASE creates these memory variables if they don't already exist.

Automem variables let you temporarily store the data from table records, manipulate the data as memory variables rather than as field values, and then return the data to the table (using REPLACE AUTOMEM or APPEND AUTOMEM).

STORE AUTOMEM is one of three commands that create automem variables. The other two, USE <filename> AUTOMEM and CLEAR AUTOMEM, initialize blank automem variables for the fields of the current table.

When referring to the value of automem variables you need to prefix the name of an automem variable with M-> to distinguish the variable from the

corresponding fields, which have the same name. The M-> prefix is not needed during variable assignment; the STORE command and the = and := operators do not work on Xbase fields.

OODML The Rowset object's contains an array of Field objects, accessed through its *fields* property. These Field objects have *value* properties that may be programmed like variables.

See Also CLEAR AUTOMEM, REPLACE, USE

SUM

Computes a total for specified numeric fields in the current table.

Syntax SUM [<exp list>
 [<scope>
 [FOR <condition 1>
 [WHILE <condition 2>
 [TO <memvar list> | TO ARRAY <array>]

<exp list> The numeric fields, or expressions involving numeric fields, to sum.

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

TO <memvar list> | TO ARRAY <array> Initializes and stores averages to the variables (or properties) of <memvar list> or stores averages to the existing array <array>. If you specify an array, each field average is stored to elements in the order in which you specify the fields in <exp list>. If you don't specify <exp list>, each field average is stored in field order. <array> can be a single- or multidimensional array; the array elements are accessed via their element numbers, not their subscripts.

Description The SUM command computes the sum of numeric expressions and stores the results in specified variables or array elements. If you store the values in variables, the number of variables must be exactly the same as the number of fields or expressions averaged. If you store the values in an array, the array must already exist, and the array must contain at least as many elements as the number of averaged expressions.

If SET TALK is ON, SUM also displays its results in the results pane of the Command window. The SET DECIMALS setting determines the number of decimal places that SUM displays. Numeric fields in blank records are evaluated as zero. To exclude blank records, use the ISBLANK() function in defining a FOR condition. EMPTY() excludes records in which a specified expression is either 0 or blank.

SUM is similar to TOTAL, which operates on an indexed or sorted table to create a second table containing the sums of the numeric and float fields of records grouped on a key expression.

OODML Loop through the rowset to calculate the sum.

See Also AVERAGE, CALCULATE, COUNT, TOTAL

TAG()

Returns the name of an open index.

Syntax TAG([<.mdx filename expC>] [<index number expN> [,<alias>]])

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.

<index position expN> the numeric position of the index tag in the specified .MDX file, or the position of the index in the list of open indexes.

<alias> The work area you want to check.

Note Unlike most functions, the first parameter is optional. If you omit the first parameter, the remaining parameters shift forward one; the second parameter becomes the first parameter, and so on.

Description TAG() returns the name of the specified index, either:

- The tag name of an index in an .MDX file, or
- The name of an .NDX file, without the file extension.

The index must be referenced by number. If you do not specify an .MDX file to check, index numbering in the list of open indexes is complicated if you have open .NDX indexes or you have open non-production .MDX files. For more information on index numbering, see SET INDEX.

If you do not specify an index tag, TAG() returns the name of the current master index tag, or an empty string if there is no master index.

If the specified .MDX file or index tag does not exist, TAG() returns an empty string.

OODML No equivalent.

Example See MDX()

See Also DBF(), DISPLAY STATUS, KEY(), MDX(), NDX(), ORDER(), SET INDEX, SET ORDER, TAGCOUNT(), TAGNO(), USE

TAGCOUNT()

Returns the number of active indexes in a specified work area or .MDX index file.

Syntax TAGCOUNT([<.mdx filename> [,<alias>]])

<.mdx filename expC> The .MDX file you want to check. The .MDX must be opened in the specified work area.

<alias> The work area you want to check.

Description TAGCOUNT() returns the total number of open indexes or the number of index tag names in a specified .MDX file. TAGCOUNT() returns 0 if there are no indexes or index tags open for the current or specified work area, or if the .MDX index file specified with <.mdx filename expC> does not exist. If you do not specify an .MDX file name, TAGCOUNT() returns the total number of indexes in the specified work area: the number of open .NDX files, plus the total number of tags in all open .MDX files. If you do not specify an alias, TAGCOUNT() returns the total number of indexes in the current work area.

OODML No equivalent.

See Also DBF(), DISPLAY STATUS, KEY(), MDX(), NDX(), ORDER(), SET INDEX, SET ORDER, TAG(), TAGNO(), USE, WORKAREA()

TAGNO()

Returns the index number of the specified index.

Syntax TAGNO([<tag name expC> [,<.mdx filename expC> [,<alias>]]])

<tag name expC> The name of the index tag that you want to return the position of. If you don't specify a tag name, TAGNO() returns the position of the current master index.

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.

<alias> The work area you want to check.

Description TAGNO() returns a number that indicates the position of the specified index name in the list of open indexes in the current or specified work area. The order of indexes is determined by the order in which they were opened with the USE or SET INDEX commands.

If you don't specify a tag name, TAGNO() returns the number of the master index. If you don't specify an .MDX file name, TAGNO() searches the list of open index files in the specified work area, including .NDX files. If you don't

specify an alias, TAGNO() operates on the list of open indexes in the current work area.

TAGNO() returns zero if the specified index tag or .MDX file does not exist.

Use TAGNO() to get the index number of an index when you know the tag name for functions like DESCENDING(), FOR(), KEY(), and UNIQUE().

OODML No equivalent.

See Also DBF(), DESCENDING(), DISPLAY STATUS, FOR(), KEY(), MDX(), NDX(), ORDER(), SET INDEX, SET ORDER, TAG(), TAGCOUNT(), UNIQUE(), USE

TARGET()

Returns the name of a table linked with the SET RELATION command.

Syntax TARGET(<expN> [,<alias>])

<expN> The number of the relation that you want to check.

<alias> The work area you want to check.

Description TARGET() returns a string containing the name of the child tables that are linked to a parent table by the SET RELATION command. You must specify the number of the relation; if the table in the current or specified work area is linked to only one table, that <expN> is the number 1. TARGET() returns an empty string ("") if no relation is set in the <expN> position.

Use TARGET() to save the link tables of all SET RELATION settings for later use when restoring relations. To save the link expression, use the RELATION() function.

OODML No equivalent. The *masterSource* and *masterRowset* properties contain references to the parent query or rowset; TARGET() returns the names of the child tables.

See Also CREATE, CREATE VIEW...FROM ENVIRONMENT, DISPLAY STATUS, RELATION(), SET(), SET RELATION, SET VIEW

TOTAL

Creates a table that stores totals for specified numeric fields of records grouped by common key values.

Syntax TOTAL ON <key field> TO <filename> [[TYPE] PARADOX | DBASE]
 [<scope>]
 [FOR <condition 1>]
 [WHILE <condition 2>]
 [FIELDS <field list>]

<key field> The name of the field on which the current table has been indexed or sorted.

TO <filename> The table to create.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if <filename> does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

<scope>

FOR <condition 1>

WHILE <condition 2> The scope of the command. The default scope is ALL.

FIELDS <field list> Specifies which numeric and float fields to total. If you don't include FIELDS, *Visual* dBASE totals all numeric and float fields.

Description

Use TOTAL to total the value of numeric fields in a table and create a second table to store the results. The numeric fields in the table storing the results contain totals for all records that have the same key field in the original table.

The current table must be either indexed or sorted on the key field. All records with the same key field become a single record in the table storing the result totals. All numeric fields appearing in the fields list contain totals. All other fields contain data from the first record of the set of records with identical keys.

To limit the fields that are created in the new file, or to group on more than one key field, use SET FIELDS as shown in the example.

TOTAL is similar to SUM, except that SUM operates on an indexed or unindexed table, returning a sum for all records of each numeric field. SUM doesn't create another table, but stores the results to memory variables or an array.

OODML No equivalent.

Example

Suppose you're totaling licensee fee revenue for a county, and you want totals for each city, for each different fee category. First you create an index on the city and fee category:

```
index on CITY + FEE_CAT tag CITY_FEE
```

Then you use SET FIELDS to create a calculated key field based on the two fields on which you can TOTAL:

```
set fields to CITY_FEE = CITY + FEE_CAT
set fields to CITY, FEE_CAT, LIC_PAID
total on CITY_FEE to CITY_PAD
```

Even though the composite field, which will appear in the result table, has the city and fee category, the city and fee category fields are included in the field list so that they will appear in the result table as separate fields. The field containing the license fee pad is also included in the field list, otherwise there would be nothing to total.

See Also AVERAGE, CALCULATE, COUNT, SUM

UNIQUE()

Indicates if a specified index ignores duplicate records.

Syntax UNIQUE([<.mdx filename expC>,<index position expN> [,<alias>]])

<.mdx filename expC> The .MDX file that contains the index tag you want to check. The .MDX must be opened in the specified work area. If omitted, all open indexes, including the production .MDX file, are searched.

<index position expN> the numeric position of the index tag in the specified .MDX file, or the position of the index in the list of open indexes.

<alias> The work area you want to check.

Note Unlike most functions, the first parameter is optional. If you omit the first parameter, the remaining parameters shift forward one; the second parameter becomes the first parameter, and so on.

Description UNIQUE() returns *true* if the index tag specified by the <index position expN> parameter was created with the UNIQUE keyword or with SET UNIQUE ON; otherwise, it returns *false*.

The index must be referenced by number. If you do not specify an .MDX file to check, index numbering in the list of open indexes is complicated if you have open .NDX indexes or you have open non-production .MDX files. For more information on index numbering, see SET INDEX. Either way, it is often easier to reference an index tag by name by using the TAGNO() function to get the corresponding position number.

If you do not specify an index tag, UNIQUE() checks the current master index tag and returns *false* if there is no master index.

If the specified .MDX file or index tag does not exist, UNIQUE() returns *false*.

OODML No equivalent.

Example See MDX()

See Also DESCENDING(), FOR(), INDEX, KEY(), MDX(), NDX(), ORDER(), SET UNIQUE, TAG(), TAGCOUNT(), TAGNO(), WORKAREA()

UNLOCK

Releases all explicit locks.

Syntax UNLOCK [ALL | IN <alias>]

ALL Releases all explicit locks in all work areas in the current workset.

IN <alias> Releases all explicit locks in the specified work area.

Description Use UNLOCK to unlock file locks you obtained with FLOCK(), or to unlock record locks you obtained with RLOCK() or LOCK(). Issue UNLOCK at the same workstation as the one at which you issued the FLOCK(), RLOCK(), and LOCK() functions. UNLOCK can't release locks obtained through other workstations, and does not release automatic file and record locks.

When you set a relation from parent table to child tables with SET RELATION and then unlock the parent table or records in the parent table with UNLOCK, Visual dBASE also unlocks child tables or records. For more information on relating tables, see SET RELATION.

OODML Use the Rowset object's *unlock()* method.

See Also FLOCK(), RLOCK(), SET RELATION

UPDATE

Replaces data in the specified fields of the current table with data from another table.

Syntax UPDATE ON <key field> FROM <alias>
 REPLACE <field 1> WITH <exp 1>
 [, <field 2> WITH <exp 2>...]
 [RANDOM]
 [REINDEX]

<key field> The key field that is common to both the current table and the table containing the updated information.

FROM <alias> The work area that provides updates to the current table.

REPLACE <field 1> The field in the current table to be updated with data from the table specified by FROM <alias>.

WITH <exp 1> The expression to store in field <field 1>. Use the FROM table's alias name and the alias operator (that is, *alias->field*) to refer to field values in the FROM table.

[, <field n> WITH <exp n> ...] Specifies additional fields to be updated.

RANDOM Specifies the FROM table is neither indexed nor sorted. (The current table must be indexed on the key field common to both tables.)

REINDEX Rebuilds open indexes after all records have been updated. Without REINDEX, Visual dBASE updates all open indexes after updating each record. When the current table has multiple open indexes or contains many records, UPDATE executes faster with the REINDEX option.

Description The UPDATE command uses data from a specified table to replace field values in the current table. It makes the changes by matching records in the two files based on a single key field.

The current table must be indexed on the field in the key field. Unless the RANDOM option is used, the table in the specified work area should also be indexed or sorted on the same field. The key fields must have identical names in the two tables.

UPDATE works by traversing the FROM table, finding the matching record in the current table (the current table must be indexed or sorted so that the match can be found quickly), and executing the REPLACE clause. If there is no match for a record in the FROM table, it is ignored. If there are multiple records in the FROM table that match a single record in the current table, all the replacements will be applied. For a simple REPLACE clause, only the last one will appear to have taken effect.

SET EXACT affects the matching, so if you are using a language driver with both primary and secondary weights (not U.S. language drivers but most others) you should have SET EXACT ON.

OODML Use the *update()* method of an UpdateSet object.

Example Suppose you have a list of students and you receive an update file containing their new grade point averages. You can use the UPDATE command to update your list of students:

```
use STUDENTS order STU_ID
use ? alias UPDATES
update on STU_ID from UPDATES replace GPA with UPDATES->GPA RANDOM
```

The ? option in the USE command displays a dialog box from which you can pick the new file. The file is always opened with the alias UPDATES.

See Also APPEND FROM, REPLACE, SELECT, SET RELATION

USE

Opens the specified table and its associated index and memo files, if any.

Syntax USE
 [<filename1> [[TYPE] PARADOX | DBASE]
 [IN <alias>]
 [INDEX <filename2> [, <filename3> ...]]
 [ORDER [TAG] <.ndx filename> |
 <tag name> [OF <.mdx filename>]]
 [AGAIN]
 [ALIAS <alias name>]
 [AUTOMEM]
 [EXCLUSIVE | SHARED]

[NOSAVE]
[NOUPDATE]]

<filename 1> The table you want to open.

[TYPE] PARADOX | DBASE Specifies the type of table you want to create, if <filename> does not include an extension. This option overrides the current SET DBTYPE setting. The TYPE keyword is included for readability only; it has no effect on the operation of the command.

IN <alias> The work area in which to open the table. You can specify the work area that is being used by another table, in which case that other table is closed first.

INDEX <filename2> [, <filename3> ...] Applicable to DBF indexes only. (Indexes on other table types are specified by the ORDER clause.) Opens up to 100 individual index files for the specified table, which can include single (.NDX) and multiple index file (.MDX) names and wildcards.

ORDER [TAG] <tag name> Makes the <tag name> index file the master index.

If you don't include the ORDER clause and the first file name after INDEX is a single index .NDX file, the single index file is the master index. If you don't include ORDER and the first file name after INDEX is a multiple index .MDX file, the table is in natural order. If the table has a primary key index, it is used; otherwise the table is unordered.

OF <.mdx filename> The .MDX file that includes <tag name>. Without OF <filename>, Visual dBASE searches for <tag name> in the table's production .MDX file, the .MDX file with the same root name as the table.

ORDER [TAG] <.ndx filename> Makes the single index file, <.ndx filename>, the master index. The .NDX file must be specified in the INDEX clause. Use the name of the index without the file extension.

AGAIN Opens a table and its related index files in the current or specified work area, leaving the table open in one or more other work areas. This keyword is superfluous and included for compatibility. Visual dBASE always opens table with AGAIN.

ALIAS <alias name> An alternate alias name to assign to the table.

AUTOMEM Initializes a memory variable for each field of the specified table (not including memo, binary, or OLE types). The memory variables are assigned the same names and types as the fields.

EXCLUSIVE | SHARED EXCLUSIVE opens the table so that no other users can open the table until you close it; SHARED allows other users access while the table is opened. This option overrides the current setting of SET EXCLUSIVE.

NOSAVE Used to open a table as a temporary table. When you close a table opened with NOSAVE, it is erased along with its associated index and memo files. If you inadvertently open a table with the NOSAVE option, use COPY to save the data.

NOUPDATE Prevents users from altering, deleting, or recalling any records in the table.

Description The USE command opens an existing table and its associated files, including index and memo files. You need to open a table before you can access any data stored in the table.

USE with no options closes the open table and its associated files in the current work area. USE IN *<alias>*, with no other options, does the same in the specified work area. CLOSE TABLES closes tables in all work areas.

You can open a table in any work area. It is common practice to USE IN SELECT() to open the table in the first available work area. If a table is already opened in the specified work area, that table is closed before the specified table is opened.

USE...INDEX specifies index files that are opened and maintained for a particular table. For a DBF table, its production .MDX is automatically opened and does not need to be listed.

The ORDER option specifies the master index from the list of indexes opened with the INDEX option and the production .MDX index. USE...INDEX is identical to USE followed by SET INDEX. See the SET INDEX and SET ORDER commands for an explanation of the open index order and specifying a master index.

You can include .NDX as well as .MDX index file names with the INDEX option. If a table has an .NDX and an .MDX index file with the same name, *Visual* dBASE opens indexes listed in the .MDX index file. In that case, to open the .NDX file you would need to specify its full name, including its extension.

When opening a table, you can name the work area by including the ALIAS option in the USE command line. ALIAS names follow the same rules as file names. Aliases are used when referring to a table from another work area. If you do not specify an *<alias name>* the table name (without the extension) is used, unless that name is invalid, because:

- That alias name is already in use by another open table, perhaps because the table is already open in another work area, or
- The table name is not a valid alias name because it is a single letter from A to J or M, which are all reserved alias names, or some other reason.

If the table name is not a valid alias, a valid default alias is generated.

The AUTOMEM option creates blank automem variables for the table, as if the CLEAR AUTOMEM command was executed immediately after opening the table.

Use the NOSAVE option of USE to open a table as a temporary file. *Visual* dBASE automatically erases the table, along with its associated memo and index files, when you close the table.

To open a table read-only, which prevents intentional or accidental changes, use the NOUPDATE option.

WORKAREA()

OODML Use a Query object with "SELECT * FROM <table>" as the *sql* property.

Example The following opens the Flight table in the Fleet database with a specific index order:

```
use :FLEET:FLIGHT order :FROM ID:
```

Note the use of the colon delimiters to both specify a table in a database and an index tag name that has spaces in it.

See Also ALIAS(), CLOSE TABLES, SELECT, SELECT(), SET INDEX, SET ORDER

WORKAREA()

Returns a number representing the currently selected work area.

Syntax WORKAREA()

Description The WORKAREA() function returns the number of the currently selected work area. Use WORKAREA() in a program to save the current work area number and then later restore that work area using the SELECT command.

Using the work area name returned by ALIAS() is generally preferred, but WORKAREA() will work better if there's a possibility that no table is in use in the current work area.

OODML There is no concept of the "current" Query object. Use your usual object management techniques to manage Query objects.

See Also ALIAS(), DBF(), SELECT, SELECT()

ZAP

Removes all records from the current table.

Syntax ZAP

Description ZAP is the fastest way to delete all records from a table. DELETE ALL, followed by PACK, also deletes all records from a table. Using ZAP requires a table be opened exclusively.

When SET SAFETY is ON and you issue ZAP, *Visual dBASE* displays a warning message asking you to confirm the operation before removing records.

OODML Use the Database object's *emptyTable()* method.

See Also DELETE, PACK, SET SAFETY

Local SQL

The Borland Database Engine (BDE) enables access to database tables through the industry-standard SQL language. Different table formats, for example InterBase® and Oracle, use different dialects of SQL. Local SQL (sometimes called “client-based SQL”) is a subset of ANSI-92 SQL for accessing DB (Paradox) and DBF (dBASE) tables and fields (called “columns” in SQL).

Although it is called “local” SQL, the DB and DBF tables may reside on a remote network file server.

For information on the SQL dialect for other table formats, consult your SQL server documentation.

SQL statements are divided into two categories:

- Data definition language
These statements are used for creating, altering, and dropping tables, and for creating and dropping indexes.
- Data manipulation language
These statements are used for selecting, inserting, updating, and deleting table data.

In the examples, an SQL statement may be displayed on multiple lines for readability. But SQL is not line-oriented. When an SQL statement is specified in a string, as it is in a Query object’s *sql* property, the entire SQL statement is specified in a single line. However, if you include a multi-line SQL statement in a program file, you must add semicolons to the end of each line (except the last) to act as line continuation characters; otherwise, the statement will not compile correctly.

SQL is not case-sensitive. The convention for SQL keywords is all uppercase, which is used in this chapter. SQL statements in the rest of the *Language Reference* may use either uppercase or lowercase.

Naming conventions

This section describes the naming conventions for tables and columns in local SQL.

Tables

Local SQL supports full file and path specifications for table names. Table names with a path, spaces, or other special characters in their names must be enclosed in single or double quotation marks. You may use forward slashes instead of backslashes. For example,

```
SELECT * FROM PARTS.DB           // Simple name with extension; no quotes required
SELECT * FROM "AIRCRAFT PARTS.DB" // Name has space; quotes needed
SELECT * FROM "C:\SAMPLE\PARTS.DB" // Filename with path
SELECT * FROM "C:/SAMPLE/PARTS.DB" // Forward slash instead of backslash
```

Local SQL also supports BDE aliases for table names. For example,

```
SELECT * FROM :IBAPPS:KBCAT
```

If you omit the file extension for a local table name, the table is assumed to be the table type specified the current setting of SET DBTYPE.

Finally, local SQL permits table names to duplicate SQL keywords as long as those table names are enclosed in single or double quotation marks. For example,

```
SELECT PASSID FROM "PASSWORD"
```

Columns

Local SQL supports multi-word column names and column names that duplicate SQL keywords as long as those column names are

- Enclosed in single or double quotation marks
- Prefaced with an SQL table name or table correlation name

For example, the following column name is two words:

```
SELECT E."Emp Id" FROM EMPLOYEE E
```

In the next example, the column name duplicates the SQL DATE keyword:

```
SELECT DATELOG."DATE" FROM DATELOG
```

Operators

Local SQL supports the following operators:

Table 13.1 Local SQL operators

| Type | Operator | Type | Operator |
|------------|-------------|----------------------|----------|
| Arithmetic | + | Logical | AND |
| | - | | OR |
| | * | | NOT |
| | / | | |
| Comparison | < | String concatenation | |
| | > | | |
| | = | | |
| | <> | | |
| | >= | | |
| | <= | | |
| | IS NULL | | |
| | IS NOT NULL | | |

Note The equality operator is a single equals sign; double equals are not allowed.

Reserved words

The following is an alphabetical list of the 215 words reserved by local SQL:

Table 13.2 List of local SQL reserved words

| | | | |
|--------------------|-----------|-------------|------------|
| ACTIVE | ADD | ALL | AFTER |
| ALTER | AND | ANY | AS |
| ASC | ASCENDING | AT | AUTO |
| AUTOINC | AVG | BASE_NAME | BEFORE |
| BEGIN | BETWEEN | BLOB | BOOLEAN |
| BOTH | BY | BYTES | CACHE |
| CAST | CHAR | CHARACTER | CHECK |
| CHECK_POINT_LENGTH | COLLATE | COLUMN | COMMIT |
| COMMITTED | COMPUTED | CONDITIONAL | CONSTRAINT |
| CONTAINING | COUNT | CREATE | CSTRING |
| CURRENT | CURSOR | DATABASE | DATE |
| DAY | DEBUG | DEC | DECIMAL |
| DECLARE | DEFAULT | DELETE | DESC |
| DESCENDING | DISTINCT | DO | DOMAIN |
| DOUBLE | DROP | ELSE | END |
| ENTRY_POINT | ESCAPE | EXCEPTION | EXECUTE |
| EXISTS | EXIT | EXTERNAL | EXTRACT |
| FILE | FILTER | FLOAT | FOR |
| FOREIGN | FROM | FULL | FUNCTION |

Table 13.2 List of local SQL reserved words

| | | | |
|-----------------|------------------------|-----------------|-------------|
| GDSCODE | GENERATOR | GEN_ID | GRANT |
| GROUP | GROUP_COMMIT_WAIT_TIME | HAVING | HOURL |
| IF | IN | INT | INACTIVE |
| INDEX | INNER | INPUT_TYPE | INSERT |
| INTEGER | INTO | IS | ISOLATION |
| JOIN | KEY | LONG | LENGTH |
| LOGFILE | LOWER | LEADING | LEFT |
| LEVEL | LIKE | LOG_BUFFER_SIZE | MANUAL |
| MAX | MAXIMUM_SEGMENT | MERGE | MESSAGE |
| MIN | MINUTE | MODULE_NAME | MONEY |
| MONTH | NAMES | NATIONAL | NATURAL |
| NCHAR | NO | NOT | NULL |
| NUM_LOG_BUFFERS | NUMERIC | OF | ON |
| ONLY | OPTION | OR | ORDER |
| OUTER | OUTPUT_TYPE | OVERFLOW | PAGE_SIZE |
| PAGE | PAGES | PARAMETER | PASSWORD |
| PLAN | POSITION | POST_EVENT | PRECISION |
| PROCEDURE | PROTECTED | PRIMARY | PRIVILEGES |
| RAW_PARTITIONS | RDB\$DB_KEY | READ | REAL |
| RECORD_VERSION | REFERENCES | RESERV | RESERVING |
| RETAIN | RETURNING_VALUES | RETURNS | REVOKE |
| RIGHT | ROLLBACK | SECOND | SEGMENT |
| SELECT | SET | SHARED | SHADOW |
| SCHEMA | SINGULAR | SIZE | SMALLINT |
| SNAPSHOT | SOME | SORT | SQLCODE |
| STABILITY | STARTING | STARTS | STATISTICS |
| SUB_TYPE | SUBSTRING | SUM | SUSPEND |
| TABLE | THEN | TIME | TIMESTAMP |
| TIMEZONE_HOUR | TIMEZONE_MINUTE | TO | TRAILING |
| TRANSACTION | TRIGGER | TRIM | UNCOMMITTED |
| UNION | UNIQUE | UPDATE | UPPER |
| USER | VALUE | VALUES | VARCHAR |
| VARIABLE | VARYING | VIEW | WAIT |
| WHEN | WHERE | WHILE | WITH |
| WORK | WRITE | YEAR | |

Data definition

Local SQL supports data definition language (DDL) for creating, altering, and dropping tables, and for creating and dropping indexes.

Local SQL does not permit the substitution of parameters for values in DDL statements.

The following DDL statements are supported:

- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE INDEX
- DROP INDEX

Data manipulation

This section describes functions available to data manipulation language (DML) statements in local SQL. It covers

- Parameter substitutions in DML statements
- Aggregate functions
- String functions
- Date function
- Updatable queries

With some restrictions, local SQL supports the following statements for data manipulation:

- SELECT, for retrieving existing data
- INSERT, for adding new data to a table
- UPDATE, for modifying existing data
- DELETE, for removing existing data from a table

Parameter substitutions in DML statements

Parameters can be used in DML statements in place of values. Parameters must always be preceded by a colon (:). For example,

```
SELECT LAST_NAME, FIRST_NAME
FROM "CUSTOMER.DB"
WHERE LAST_NAME > :parm1 AND FIRST_NAME < :parm2
```

Assigning an SQL statement with parameters in a Query or StoredProc object automatically creates the corresponding elements in the object's *params* array. You then store values to substitute in that array.

Aggregate functions

The following ANSI-standard SQL aggregate functions are available to local SQL for use with data retrieval:

- SUM(), for totaling all numeric values in a column
- AVG(), for averaging all non-NULL numeric values in a column

- MIN(), for determining the minimum value in a column
- MAX(), for determining the maximum value in a column
- COUNT(), for counting the number of values in a column that match specified criteria

Complex aggregate expressions are supported, such as

```
SUM( Field * 10 )
SUM( Field ) * 10
SUM( Field1 + Field2 )
```

String functions

Local SQL supports the following ANSI-standard SQL string manipulation functions for retrieval, insertion, and updating:

- UPPER(), to force a string to uppercase:

```
UPPER(<expC>)
```

- LOWER(), to force a string to lowercase:

```
LOWER(<expC>)
```

- TRIM(), to remove repetitions of a specified character from the left, right, or both sides of a string:

```
TRIM( BOTH | LEADING | TRAILING
      <char> FROM <expC> )
```

- SUBSTRING() to create a substring from a string:

```
SUBSTRING(<expC> FROM <start expN> FOR <length expN>)
```

You may use the LIKE predicate for pattern matching in the WHERE clause:

```
WHERE <expC> LIKE <pattern expC> [ESCAPE <char>]
```

In *<pattern expC>*, the % (percent) character stands for zero or more wildcard characters, and the _ (underscore) stands for a single wildcard character. To include either special character as an actual pattern character, specify an ESCAPE character and precede the wildcard character with that escape character.

Enclose literal strings in single quotes. To specify a single quote in a literal string, use double single quotes.

Example The following query returns the contents of the Title column, removing any double quotation marks around the text:

```
SELECT TRIM( BOTH '"' FROM TITLE ) FROM BOOKS
```

The following query returns all rows where the first three letters of the City column are “San”, capitalized in any way:

```
SELECT * FROM CUSTOMER WHERE UPPER( SUBSTRING( CITY FROM 1 FOR 3 )) = 'SAN'
```

The following query returns all rows where the letters “n’t” appear in the Title (e.g. “Can’t”, “Won’t”, “Ain’t”, “Don’t”):

```
SELECT * FROM BOOKS WHERE TITLE LIKE '%n't%'
```

Date function

Local SQL supports the `EXTRACT()` function for isolating a single numeric field from a date/time field on retrieval using the following syntax:

```
EXTRACT (<extract field> FROM <field name>)
```

where *<extract_field>* can be one of: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND.

In local SQL, `EXTRACT()` does not support the `TIMEZONE_HOUR` or `TIMEZONE_MINUTE` clauses.

Example The following statement extracts the year value from a DATE field:

```
SELECT EXTRACT(YEAR FROM HIRE_DATE)
FROM EMPLOYEE
```

Updatable queries

These restrictions apply to updates:

- Linking fields cannot be updated
- Index switching will cause an error

Restrictions on live queries

Single-table queries are updatable provided that

- There are no JOINS, UNIONS, INTERSECTs, or MINUS operations.
- There is no DISTINCT key word in the SELECT. (This restriction may be relaxed if all the fields of a unique index are projected.)
- Everything in the SELECT clause is a simple column reference or a calculated field; no aggregation is allowed.
- The table referenced in the FROM clause is either an updatable base table or an updatable view.
- There is no GROUP BY or HAVING clause.
- There are no subqueries that reference the table in the FROM clause and no correlated subqueries.
- Any ORDER BY clause can be satisfied with an index (a simple single-field index for DBF tables).

Restrictions on live joins

Live joins may be used only if

- All joins are left-to-right outer joins.
- All join are equi-joins.
- All join conditions are satisfied by indexes.
- Output ordering is not defined.
- The query contains no elements listed above that would prevent single-table updatability.

Constraints

You can constrain any updatable query by setting the Query object's *constrained* property to *true* before activating the query. This causes the query to behave more like an SQL-server-based query. New or modified rows that do not match the conditions of the query will disappear from the result set, although the data is saved.

Statements supported

The rest of this chapter describes the DDL and DML statements supported by local SQL.

ALTER TABLE

Adds or drops (deletes) one or more columns (fields) from a table.

Syntax

```
ALTER TABLE table
    ADD <column name> <data type> |
    DROP <column name>
    [, ADD <column name> <data type> |
    DROP <column name> ...]
```

Description

Use ALTER TABLE to modify the structure of an existing table. ALTER TABLE with the ADD clause adds the column <column name> of the type <data type> to <table name>. Use the DROP clause to remove the existing column <column name> from <table>.

Warning

Data stored in a dropped column is lost without warning, regardless of the SET SAFETY setting.

Multiple columns may be added and/or dropped in a single ALTER TABLE command.

Use ALTER TABLE as a means of modifying the structure of a table without using the Table Designer.

Examples

The following statement adds a column:

```
ALTER TABLE "employee.dbf" ADD BUILDING_NO SMALLINT
```

The next statement drops two columns:

```
ALTER TABLE "employee.db" DROP LAST_NAME, DROP FIRST_NAME
```

The following statement drops two columns and adds one:

```
ALTER TABLE "employee.dbf" DROP LAST_NAME, DROP FIRST_NAME, ADD FULL_NAME CHAR(30)
```

CREATE INDEX

Creates a new index on a table.

Syntax CREATE INDEX <index name> ON <table name> (<column name> [, <column name>...])

Description Use CREATE INDEX to create a new index <index name>, in ascending order, based on the values in one or more columns <column name> of <table name>. Expressions cannot be used to create an index, only columns.

When working with DBF tables, the index can only be created for a single column. The new index is created as a new index tag in the production index. A production index is created if it does not exist. Using CREATE INDEX is the only way to create indexes for DBF tables in SQL.

CREATE INDEX can create only secondary indexes for Paradox tables. Primary Paradox indexes can be created only by specifying a PRIMARY KEY constraint when creating a new table with CREATE TABLE. The secondary indexes are created as case-insensitive and maintained, when possible.

CREATE INDEX is equivalent to the INDEX ON <field list> TAG <tag name> syntax in the dBASE language.

Examples The following statement creates an index on a DBF table:

```
CREATE INDEX NAMEX ON employee.dbf (LAST_NAME)
```

The following statement adds an index called ZIP on the ZIP_POSTAL column of the CUSTOMER table:

```
CREATE INDEX ZIP ON CUSTOMER (ZIP_POSTAL)
```

CREATE TABLE

Creates a new table.

Syntax CREATE TABLE <table name> (<column name> <data type> [, <column name> <data type>...] [, PRIMARY KEY(<field name>)])

Description Create a Paradox or dBASE table using local SQL by specifying the file extension when naming the table:

- DB for Paradox tables
- DBF for dBASE tables

If you omit the file extension for a local table name, the table created is the table type specified in the Default Driver setting in the System page of the BDE Administrator.

CREATE TABLE has the following limitations:

- Column definitions based on domains are not supported.
- Constraints are limited to PRIMARY KEY. For DBF7 tables, only single-field primary keys are supported through the CREATE TABLE command. (Use the Table Designer or the Xbase INDEX command to create complex primary keys.) Primary keys are not supported for earlier versions of DBF.

At least one *<column name>* *<data type>* must be defined. The column definition list must be enclosed in parentheses.

CREATE TABLE is an alternate way of creating a table without using the Table Designer, the Database object's *copyTable()* method, or an UpdateSet object.

Data type mappings for CREATE TABLE

The following table lists SQL syntax for data types used with CREATE TABLE, and describes how those types are mapped to Paradox (DB) and dBASE (DBF) types by BDE:

| SQL syntax | DB | DBF 7 | DBF 5 |
|--------------|----------------|----------------|----------------|
| SMALLINT | Short | Long | Numeric (6,10) |
| INTEGER | Long Integer | Long | Numeric (20,4) |
| DECIMAL(x,y) | BCD | Numeric (x,y) | N/A |
| NUMERIC(x,y) | Number | Numeric (x,y) | Numeric (x,y) |
| FLOAT(x,y) | Number | Double | Float (x,y) |
| CHARACTER(n) | Alpha | Character (n) | Character (n) |
| VARCHAR(n) | Alpha | Character (n) | Character (n) |
| DATE | Date | Date | Date |
| BOOLEAN | Logical | Logical | Logical |
| BLOB(n,1) | Memo | Memo | Memo |
| BLOB(n,2) | Binary | Binary | Binary |
| BLOB(n,3) | Formatted memo | N/A | N/A |
| BLOB(n,4) | OLE | OLE | OLE |
| BLOB(n,5) | Graphic | N/A | N/A |
| TIME | Time | N/A | N/A |
| TIMESTAMP | Timestamp | Timestamp | N/A |
| MONEY | Money | Numeric (20,4) | Numeric (20,4) |
| AUTOINC | Autoincrement | Autoincrement | N/A |
| BYTES(n) | Bytes | N/A | N/A |

x = precision (default: specific to driver)

y = scale (default: 0)

n = length in bytes (default: 0)

1-5 = BLOB subtype (default: 1)

Examples The following example creates a DBF table called SALES with the following structure:

Table 13.3 SALES.DBF structure

| Field name | Field type | Field length | Decimal places |
|------------|------------|--------------|----------------|
| SALESID | Character | 6 | |
| CUSTOMERID | Character | 10 | |
| ORDERDATE | Date | 8 | |
| ORDERNMBR | Numeric | 7 | 0 |
| ORDERAMT | Numeric | 9 | 2 |
| DELIVERED | Logical | 1 | |

```
CREATE TABLE SALES (
  SALESID      CHAR(6),
  CUSTOMERID   CHAR(10),
  ORDERDATE    DATE,
  ORDERNMBR    NUMERIC(7,0),
  ORDERAMT     NUMERIC(9,2),
  DELIVERED    BOOLEAN)
```

The following statement creates a Paradox table with a PRIMARY KEY constraint on the LAST_NAME and FIRST_NAME columns:

```
CREATE TABLE "employee.db" (
  LAST_NAME    CHAR(20),
  FIRST_NAME   CHAR(15),
  SALARY       NUMERIC(10,2),
  DEPT_NO      SMALLINT,
  PRIMARY KEY(LAST_NAME, FIRST_NAME) )
```

DELETE

Deletes rows (records) from a table.

Syntax DELETE FROM <table name> [WHERE <search condition>]

Description Use DELETE to delete rows, or records, from <table name>. Without the WHERE clause, all the rows in the table are deleted. Use the WHERE clause to specify a <search condition>. Only records matching the <search condition> are deleted.

The Local SQL DELETE command is similar to the Xbase DELETE command; DELETE FROM with no WHERE clause is like the Xbase DELETE ALL.

In DBF tables, DELETE only **marks** rows as deleted; it does not remove them from the table. They may be recalled using the Xbase RECALL command. To remove the deleted records, use the Xbase PACK command. In Paradox tables, the rows are actually deleted, and are not recallable.

Example The following example deletes all the rows in a DBF table called CUSTOMER and results in a table with zero rows:

```
DELETE FROM CUSTOMER.DBF
```

The following example marks all the rows in a DBF table called CUSTOMER for deletion, but does not actually delete the rows from the table:

```
DELETE FROM CUSTOMER.DBF WHERE CUSTOMER_N > 0
```

The following example marks all the rows where the CITY field is equal to "Freeport" for deletion in a DBF table called CUSTOMER:

```
DELETE FROM CUSTOMER.DBF WHERE CITY = "Freeport"
```

The following example deletes all the rows where the CITY field is equal to "Freeport" in a Paradox table called CUSTOMER:

```
DELETE FROM CUSTOMER.DB WHERE CITY = "Freeport"
```

DROP INDEX

Drops (deletes) an existing index from a table.

Syntax DROP INDEX <table_name>.<index_name> | PRIMARY

Description Use DROP INDEX to drop, or delete, the index <index name> from <table name>. For DBF tables <index name> must be the name of a tag in the production index. The PRIMARY keyword is used to delete a primary Paradox index. For example, the following statement drops the primary index on EMPLOYEE.DB:

```
DROP INDEX "employee.db".PRIMARY
```

To drop any dBASE index, or to drop secondary Paradox indexes, provide the index name. For example, this statement drops a secondary index on a Paradox table:

```
DROP INDEX "employee.db".NAMEX
```

Example The following statement drops the index tag NAME from the production index of a dBASE table called EMPLOYEE:

```
DROP INDEX EMPLOYEE.NAME
```

DROP TABLE

Drops (deletes) a table.

Syntax DROP TABLE <table name>

Description Use DROP TABLE to delete the table <table name> from disk. The associated production index file and memo file, if any, are also deleted.

Examples The following statement drops a table named EMPLOYEE:


```
DROP TABLE EMPLOYEE
```

INSERT

Adds new rows (records) to a table.

Syntax `INSERT INTO <table name>`
 `[(<column list>)] VALUES (<value list>) |`
 `SELECT <command>`

Insertion from one table to another through a subquery is not allowed.

Description Use INSERT to add rows, or records, to a table. There are two forms of this command. In the first form, you use <value list> to specify individual column values that are to be inserted for the new row. The values to be inserted must match in number, order, and type with the columns specified in <column list>, if <column list> is specified. Columns in the new row for which no value is given are left blank. If no <column list> is given, the order of the columns as they appear in the table is assumed. Without a <column list> a value must be provided for each column in the <value list>.

In the second form, the SELECT clause is executed just like a SELECT command. The row or rows returned by the SELECT are inserted into <table name>. The columns of the rows returned by the SELECT are matched up with the columns listed in <column list>. Therefore, the columns returned by SELECT must match in number, order, and type with the columns specified in <column list>, if <column list> is specified. If no <column list> is given, the number, order, and type of the columns returned by the SELECT must match the number, order, and type of the columns in <table name>.

Examples The following statement adds a row to a table, assigning values to two columns:

```
INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID) VALUES (52, "DGP11")
```

The next statement specifies values to insert into a table with a SELECT statement:

```
INSERT INTO PROJECTS
SELECT * FROM NEW_PROJECTS
WHERE NEW_PROJECTS.START_DATE > '06/06/94'
```

SELECT

Retrieves data from one or more tables.

Syntax `SELECT [DISTINCT] <column list>`
 `FROM <table reference>`
 `[WHERE <search condition>]`
 `[ORDER BY <order list>]`
 `[GROUP BY <group list>]`
 `[HAVING <having condition>]`

SELECT

```
[UNION <select expr>]  
[SAVE TO <table>]
```

Description Use SELECT to retrieve data from a table or set of tables based on some criteria.

A SELECT that retrieves data from multiple tables is called a join.

<column list> is a comma-delimited list of columns in the table(s) you want to retrieve. The columns are retrieved in the order given in the list. If two or more tables used by SELECT use the same field names, distinguish the tables by using the table name and a dot (.). For example, if you're SELECTing from the CUSTOMER table and the PRODUCT table, and they both have a field called NAME, enter the fields as CUSTOMER.NAME and PRODUCT.NAME in <column list>. To retrieve all the columns from <table list>, use an asterisk (*) for <column list>. To eliminate rows containing duplicate values within the same column, precede the <column list> with the keyword DISTINCT.

For example, the following statement retrieves data from two columns:

```
SELECT PART_NO, PART_NAME  
FROM PARTS
```

You may include calculated fields in the <column list>, optionally using the AS option to name them. For example:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE  
FROM CUSTOMER
```

A SELECT statement that contains a join must have a WHERE clause in which at least one field from each table is involved in an equality check.

FROM clause

The FROM clause specifies the table or tables from which to retrieve data. <table reference> can be a single table, a comma-delimited list of tables, or can be an inner or outer join as specified in the SQL-92 standard. For example, the following statement specifies a single table:

```
SELECT PART_NO FROM PARTS
```

The next statement specifies a left outer join for table_reference:

```
SELECT * FROM PARTS LEFT OUTER JOIN INVENTORY  
ON PARTS.PART_NO = INVENTORY.PART_NO
```

WHERE clause

The optional WHERE clause reduces the number of rows returned by a query to those that match the criteria specified in <search condition>. For example, the following statement retrieves only those rows with PART_NO greater than 543:

```
SELECT * FROM PARTS  
WHERE PART_NO > 543
```

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, LIKE, ANY, ALL, and EXISTS are supported.

The IN predicate is followed by a list of values in parentheses. For example, the next statement retrieves only those rows where a part number matches an item in the IN predicate list:

```
SELECT * FROM PARTS
WHERE PART_NO IN (543, 544, 546, 547)
```

ORDER BY clause

The ORDER BY clause specifies the order of retrieved rows, using the keywords ASC (the default) and DESC for ascending and descending, respectively. For example, the following query retrieves a list of all parts listed in alphabetical order by part name:

```
SELECT * FROM PARTS
ORDER BY PART_NAME ASC
```

The next query retrieves all part information ordered in descending numeric order by part number:

```
SELECT * FROM PARTS
ORDER BY PART_NO DESC
```

Calculated fields can be ordered by correlation name or ordinal position. For example, the following query orders rows by FULL_NAME, a calculated field:

```
SELECT LAST_NAME || ', ' || FIRST_NAME AS FULL_NAME, PHONE
FROM CUSTOMER
ORDER BY FULL_NAME
```

Projection of all grouping or ordering columns is not required.

GROUP BY clause

The GROUP BY clause specifies how retrieved rows are grouped for aggregate functions. For example,

```
SELECT PART_NO, SUM(QUANTITY) AS PQTY
FROM PARTS
GROUP BY PART_NO
```

Aggregates in the SELECT clause must have a GROUP BY clause if a projected field is used, as shown in the example above.

HAVING clause

The HAVING clause specifies conditions records must meet to be included in the return from a query. It is a conditional expression used in conjunction with the GROUP BY clause. Groups that do not meet the expression in the HAVING clause are omitted from the result set.

Subqueries are supported in the HAVING clause. A subquery works like a search condition to restrict the number of rows returned by the outer, or parent, query. See WHERE Clause.

In addition to scalar comparison operators (=, <, > ...) additional predicates using IN, LIKE, ANY, ALL, and EXISTS are supported.

UNION clause

The UNION clause combines the results of two or more SELECT statements to produce a single table.

Heterogeneous joins

Local SQL supports joins of tables in different database formats; such a join is called a heterogeneous join.

When you specify a table name after selecting a local alias,

- For local tables, specify either the alias or the path.
- For remote tables, specify the alias.

The following statement retrieves data from a Paradox table and a dBASE table:

```
SELECT DISTINCT C.CUST_NO, C.STATE, O.ORDER_NO
FROM CUSTOMER.DB C, ORDER.DBF O
WHERE C.CUST_NO = O.CUST_NO
```

You can also use BDE aliases in conjunction with table names.

SAVE TO clause

The SAVE TO clause saves the data gathered by the SELECT into another table, instead of returning the result set. Use this option to copy part or all of a table into another table, or to save the result of a join or aggregate to another table. For example, the following statement averages student scores by grade and stores the result to another table:

```
SELECT GRADE, AVG(SCORE)
FROM STUDENTS
GROUP BY GRADE
SAVE TO SCORES
```

Examples The following is a basic query that selects an entire table:

```
SELECT * FROM BIOLIFE
```

The following examples show simple SELECTs:

```
SELECT NAME, PHONE FROM CUSTOMER WHERE STATE_PROV = "CA"
SELECT CUSTOMER_NO FROM CUSTOMER WHERE LAST_NAME = "Johnson"
SELECT PART_NO, SUM(QUANTITY) AS PQTY FROM PARTS GROUP BY PART_NO
```

The following example illustrates the ORDER BY with a DESCENDING clause:

```
SELECT DISTINCT CUSTOMER_NO
FROM "C:/DATA/CUSTOMER"
ORDER BY CUSTOMER_NO DESCENDING
```

The following example illustrates how the SELECT statement is supported as an equivalent to a JOIN:

```
SELECT DISTINCT P.PART_NO, P.QUANTITY, G.CITY
FROM PARTS P, GOODS G
WHERE P.PART_NO = G.PART_NO
AND P.QUANTITY > 20
```

```
ORDER BY P.QUANTITY, G.CITY, P.PART_NO
```

Sub-select queries are supported. The following example illustrates this syntax:

```
SELECT P.PART_NO
FROM PARTS P
WHERE P.QUANTITY IN
      (SELECT I.QUANTITY
       FROM INVENTORY I
       WHERE I.PART_NO = 'AA9393')
```

The following example shows a join in which fields from each table are involved in some type of equality check and require a WHERE clause:

```
SELECT DISTINCT PARTS.PART_NO, PARTS.QUANTITY, GOODS.CITY
FROM PARTS, GOODS
WHERE PARTS.PART_NO = GOODS.PART_NO AND PARTS.QUANTITY > 20
ORDER BY PARTS.QUANTITY, GOODS.CITY, PARTS.PART_NO
```

The following example shows the use of the DESCENDING keyword in the ORDER BY clause. Note that in this case you must also specify DISTINCT.

```
SELECT DISTINCT CUSTOMER_NO
FROM CUSTOMER
ORDER BY CUSTOMER_NO DESCENDING
```

UPDATE

Adds or changes values in existing columns in existing rows of a table.

Syntax UPDATE <table name>
 SET <column name> = <expression> [, <column name> = <expression>...]
 WHERE <search condition>

Description Use UPDATE to update (change) values within existing columns in existing rows of a table. The column specified by <column name> is updated with the value of <expression> in all rows that match the <search criteria> of the WHERE clause. If the WHERE clause is omitted, the column is updated in all rows in the table. Multiple columns may be updated in a single UPDATE command. A given column of a table may only appear once to the left of an equal sign (=) in the SET clause.

Example The following command updates that YTD sales to zero for each customer that was contacted in the previous calendar year:

```
UPDATE CUSTOMER SET YTD_SALES = 0 WHERE FIRST_CONT < '01/01/95'
```


Data access objects

Data access objects provide access to database tables and are used to link tables to the user interface.

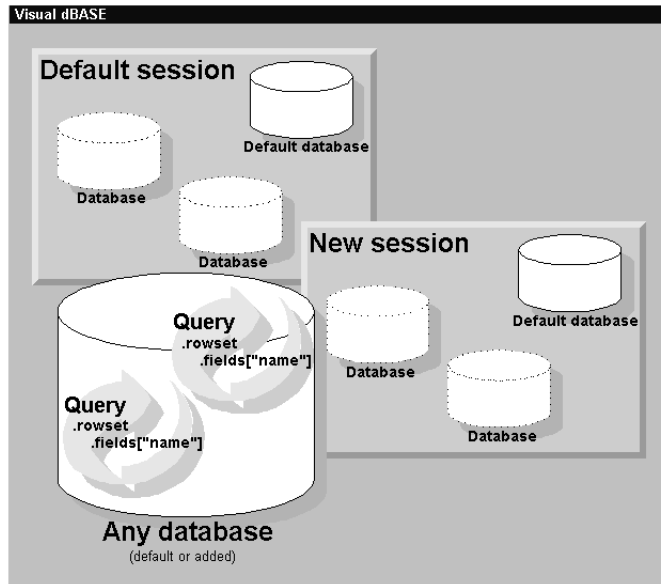
The Borland Database Engine (BDE) considers the DBF (dBASE/FoxPro) and DB (Paradox) tables types as Standard tables. The BDE can access any Standard table directly through its path and file name, without having to use a BDE alias.

All other table types, including InterBase, Oracle, Microsoft/Sybase SQL Server, Informix, and any ODBC connection, require the creation of a BDE alias through the BDE Administrator. You may also create a BDE alias to access Standard tables. In that case, the alias specifies the directory in which the tables exist; the database consists of the Standard tables in that directory, and you may not open any others from another directory.

All tables, whether or not they require a BDE alias, are accessed through SQL and the data access objects.

Understanding the data access object hierarchy

To understand the implications of using a BDE alias, you need to understand the class hierarchy of the data access objects, as shown in Figure 14.1.

Figure 14.1 Data access objects: class hierarchy diagram

At the top of the hierarchy is *Visual dBASE* itself. Next is the *Session* class. A session represents a separate user task, and is required primarily for DBF and DB table security. *Visual dBASE* supports up to 2048 simultaneous sessions. When *Visual dBASE* first starts, it already has a default session. Unless your application needs to log in as more than one person simultaneously, there is usually no need to create your own session objects.

Each session contains one or more *Database* objects. A session always contains a default Database object, one that has no BDE alias and is intended to directly access Standard tables. You must create new Database objects to use tables through a BDE alias. Once you set the BDE alias, activate the Database object, and log in if necessary, you have access to that database's tables. You may also log transactions or buffer updates to each database to allow you to rollback, abandon, or post changes as desired.

Accessing tables

The *Query* object acts primarily as a container for an SQL statement and the set of rows, or *rowset*, that results from it. A rowset represents all or part of a single table or group of related tables. There is only one rowset per query, but you may have more than one query, and therefore more than one rowset, per database. A rowset maintains the current record or row, and therefore contains the typical navigation, buffering, and filtering methods.

The SQL statement may also contain parameters, which are represented in the Query object's *params* array.

Finally, a rowset also contains a *fields* property, which is an array of field objects that contain information about the fields and the values of the fields for the current row. There are events that allow you to morph the values so that the values stored in the table are different than the values displayed. Each field object can also be linked to a visual component through the component's *dataLink* property to form a link between the user interface and the table. When the two objects are linked in this way, they are said to be *dataLinked*.

Putting the data access objects together

If you're using Standard tables only, at the minimum you create a query, which gets assigned to the default database in the default session, set the SQL statement and make the query active. If the query is successful, it generates a rowset, and you can access the data through the *fields* array.

When accessing tables through a BDE alias, you will need to create a new database, create the query, assign the database to the query, then set the SQL and make the query active.

If you use the Form, Report, or Data Module designers, you design these relationships visually and code is generated. For more information, see the *Developer's Guide*.

Using stored procedures

The object hierarchy for using stored procedures in an SQL-server database is very similar to the one used for accessing tables. The difference is that a *StoredProc* object is used instead of a Query object. Above the *StoredProc* object, the Database and Session objects do the same thing. If the stored procedure returns a rowset, the *StoredProc* object contains a rowset, just like a Query object.

A *StoredProc* object also has a *params* array, but instead of simple values to substitute into an SQL statement in a Query object, the *params* array of a *StoredProc* object contains *Parameter* objects. Each object describes both the type of parameter—input, output, or result—and the value of that parameter.

Before running the stored procedure, input values are set. After the stored procedure runs, output and result values can be read from the *params* array, or data can be accessed through its rowset.

class Database

A session's built-in database or a BDE database alias, which gives access to tables.

Syntax [*<oRef>* =] new Database()

<oRef> A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Database object.

Properties The following tables list the properties and methods of the Database class. (No events are associated with this class.)

| Property | Default | Description |
|-----------------------|-----------------|--|
| <i>active</i> | false | Whether the database is open and active or closed |
| <i>cacheUpdates</i> | false | Whether to cache changes locally for batch posting later |
| <i>className</i> | DATABASE | Identifies the object as an instance of the Database class (Property discussed in Chapter 5, "Core language.") |
| <i>databaseName</i> | Empty string | BDE alias, or empty string for built-in database |
| <i>driverName</i> | Empty string | Type (table format or server) of database |
| <i>handle</i> | | BDE database handle |
| <i>isolationLevel</i> | Read committed | Isolation level of transaction |
| <i>loginString</i> | Empty string | User name and password to automatically try when opening database |
| <i>parent</i> | null | Container form or report (Property discussed in Chapter 5, "Core language.") |
| <i>session</i> | Default session | Session to which a database is assigned |
| <i>share</i> | All | How to share the database connection |

| Method | Parameters | Description |
|-------------------------|--|---|
| <i>abandonUpdates()</i> | | Discards all cached changes |
| <i>applyUpdates()</i> | | Attempts to post cached changes |
| <i>beginTrans()</i> | | Begins transaction; starts logging changes |
| <i>close()</i> | | Closes the database connection (called implicitly when <i>active</i> is set to <i>false</i>) |
| <i>commit()</i> | | Commits changes made during transaction; ends transaction |
| <i>copyTable()</i> | <source name expC>, <destination name expC> | Makes a copy of a table in the same database |
| <i>createIndex()</i> | <table name expC>, <index name expC>, <key> | Creates an index in the table |
| <i>dropIndex()</i> | <table name expC>, <index name expC> | Deletes index from table |
| <i>dropTable()</i> | <table name expC> | Deletes table from database |
| <i>emptyTable()</i> | <table name expC> | Deletes all records from a table |
| <i>executeSQL()</i> | <expC> | Pass-through SQL statement |
| <i>getSchema()</i> | "DATABASES" "TABLES" "PROCEDURES" "VIEWS" | Retrieves information about a database |
| <i>open()</i> | | Opens the database connection (called implicitly when <i>active</i> is set to <i>true</i>) |
| <i>packTable()</i> | <table name expC> | Removes deleted records from DBF or DB table and reconsolidates disk usage |
| <i>reindex()</i> | <table name expC> | Rebuilds indexes for DBF or DB table |

| Method | Parameters | Description |
|----------------------|-------------------------------------|---|
| <i>renameTable()</i> | <old name expC>, <new name expC> | Renames table in database |
| <i>rollback()</i> | | Undoes changes made during transaction; ends transaction |
| <i>tableExists()</i> | <table name expC> | Whether or not specified table exists in database or on disk |

Description

All sessions, including the default session you get when you start *Visual dBASE*, contain a default database, which can access the Standard table types, DBF (dBASE) and DB (Paradox) tables, without requiring a BDE alias. Whenever you create a Query object, it is initially assigned to the default database in the default session. If you want to use Standard tables in the default session you don't have to do anything with that Query object's *database* or *session* properties. If you want to use a Standard table in another session, for example to use DBF or DB table security, assign that session to the Query object's *session* property, which causes that session's default database to be assigned to that Query object. Default databases are always active; their *active* property has no effect.

You may also set up a BDE alias to access Standard tables. By referring to your Standard tables through a database alias, you can move the tables to a different drive or directory without having to change any paths in your code. All you would have to do is change the path specification for that alias in the BDE Administrator. When using a BDE alias with Standard tables, you are restricted from opening a table in a different directory.

For all non-Standard table types, you will need to set up a BDE alias for the database if you haven't done so already. After creating a new Database object, you may assign it to another session if desired; otherwise it is assigned to the default session. Then you need to do the following:

- Assign the BDE alias to the *databaseName* property.
- If you need to log in to that database, either set the *loginString* property if you already know the user name and password; or let the login dialog appear.
- Set the *active* property to *true*. This attempts to open the named database. If it's successful, you now have access to the tables in the database.

Each database, including any default databases, is able to independently support either transaction logging or cached updates. Transaction logging allows changes to be made to tables as usual, but keeps track of those changes. Those changes can then be undone through a *rollback()*, or OK'd with a *commit()*. In contrast, cached updates are not written to the table as they happen, but are cached locally instead. You can then either abandon all the updates or attempt to apply them as a group. If any of the changes fail to post—for a variety of reasons, like locked records or hardware failures—any changes that did take are immediately undone, and the updates remain cached. You can then attempt to solve the problem and reapply the update, or abandon the changes. You may also want to use cached updates to reduce network traffic.

Each non-Standard database is responsible for its own transaction processing, up to whatever isolation level it supports. For Standard tables opened through the default database, if you want simultaneous multiple transactions, you need to create multiple sessions, because each database object can support only one active transaction or update cache, and there is only one default database per session.

All Database objects opened by the Navigator are listed in the *databases* array property of the *_app* object. The default database of the default session is *_app.databases[1]*.

A Database object also encapsulates a number of table maintenance methods. These methods occur in the context of the specified Database object. For example, the *copyTable()* method makes a copy of a table in the same database. To use these methods on Standard tables, call the methods through the default database of the default session; for example,

```
_app.databases[ 1 ].copyTable( "Stuff", "CopyOfStuff" )
```

Example Suppose you have an Access database named PIBMUG.MDB. You create an alias named PIBMUG in the BDE Administrator. To open that database, execute the following code:

```
d = new Database()
d.databaseName = "PIBMUG"
d.active = true
```

The second example logs into a database named PERSONNEL in a new session with a preset user name and password:

```
s1 = new Session()
d1 = new Database()
d1.databaseName = "PERSONNEL"
d1.session = s1
d1.loginString = "visitor/jobsavail"
d1.active = true
```

See also class Query, class Rowset, class Session

class DataModule

An empty container in which to store data access objects.

Syntax These objects are usually instantiated through a DataModRef object.

Properties The following table lists the properties of the DataModule class. (No events or methods are associated with this class.)

| Property | Default | Description |
|------------------|------------|--|
| <i>className</i> | DATAMODULE | Identifies the object as an instance of the DataModule class (Property discussed in Chapter 5, “Core language.”) |
| <i>rowset</i> | | The primary rowset of the data module |

Description Use data modules to maintain multiple data access objects and the relationships between them. Data modules bear some similarity to forms, except that they contain data access objects only. Session, Database, Query, and StoredProc objects are contained inside a DataModule object; the class is created visually with the Data Module designer. They are represented by source code in files with a .DMD extension. You can create custom data modules (in .CDM files) and subclass them.

The relationships between the objects—in particular any *masterSource*, *masterRowset*, or *masterFields* properties—in addition to other properties and event handlers can be set for all the objects in the data module. A primary rowset is assigned in the data module’s *rowset* property, just like in a form. Other than that and *className*, the DataModule object has no properties; it is intended to be a simple container.

To use the data module, create a DataModRef object in the form or report. For more information, see class DataModRef.

Example The following data module implements the classic teacher-classes-students database. In addition to those three tables, there is a fourth linking table called Attend for the many-to-many link between classes and students.

```
class TeacherClassesStudentsDataModule of DATAMODULE

    this.TEACHER1 = new QUERY()
    this.TEACHER1.parent = this
    with (this.TEACHER1)
        left = 2
        top = 1
        sql = 'select * from "TEACHER.DBF"'
        active = true
    endwith

    with (this.TEACHER1.rowset)
        indexName = "FULL_NAME"
    endwith

    this.CLASSES1 = new QUERY()
    this.CLASSES1.parent = this
    with (this.CLASSES1)
        left = 8
        top = 3
        sql = 'select * from "CLASSES.DBF"'
        active = true
```

```

endwith

with (this.CLASSES1.rowset)
    indexName = "TEACH_NAME"
    masterRowset = parent.parent.teacher1.rowset
    masterFields = "TEACH_ID"
endwith

this.ATTEND1 = new QUERY()
this.ATTEND1.parent = this
with (this.ATTEND1)
    left = 14
    top = 5
    sql = "@ATTEND STUDENT.SQL"
    params["class_id"] = ""
    masterSource = form.classes1.rowset
    active = true
endwith

this.STUDENT1 = new QUERY()
this.STUDENT1.parent = this
with (this.STUDENT1)
    left = 20
    top = 7
    sql = 'select * from "STUDENT.DBF"'
    active = true
endwith

with (this.STUDENT1.rowset)
    indexName = "STU_ID"
    masterRowset = parent.parent.attend1.rowset
    masterFields = "STU_ID"
endwith

this.rowset = this.TEACHER1.rowset

endclass

```

The Teacher table is ordered by the Full_name index. It is related into a table of classes through the *classes1.rowset.masterRowset* property. The Classes table is ordered on the Teach_name tag, a composite index of the Teach_id field (to match the *masterFields*) and the class name.

The *classes1* query acts as the *masterSource* for the *attend1* query. The Attend table has only two fields, Class_id and Stu_id. This table can be used to link classes and students in either direction. For this query, the goal is to create a set of students that attended the class in student name order. The Class_id field from the *classes1* query is the parameter in the parameterized SQL statement stored in "Attend student.SQL" file:

```

SELECT Student.LAST_NAME, Student.FIRST_NAME, Student.STU_ID
FROM "ATTEND.DBF" Attend
    INNER JOIN "STUDENT.DBF" Student
        ON (Attend.STU_ID = Student.STU_ID)
WHERE Attend.CLASS_ID = :class_id
ORDER BY Student.LAST_NAME, Student.FIRST_NAME

```

This SQL SELECT performs an inner join (matching rows only) between the Attend and Student table to get the students' names so that it can sort on them. (Local SQL requires that the ORDER BY fields be in the result set.) The "class_id" in the WHERE clause is substituted with the value of the Class_id field in the *masterSource* query (*classes1*).

Finally, to actually display the student information, the *student1* query's rowset specifies *attend1.rowset* as its *masterRowset*; a one-to-one link. The *indexName* is set to match. This link makes the student information editable. You could get similar results by using fewer queries with more joins, but then the result would be read-only.

See also class DataModRef

class DataModRef

A reference to a DataModule object.

Syntax [*<oRef>* =] new DataModRef()

<oRef> A variable or property—typically of a Form or Report object—in which to store a reference to the newly created DataModRef object.

Properties The following table lists the properties of the DataModRef class. (No events or methods are associated with this class.)

| Property | Default | Description |
|---------------------|------------|--|
| <i>active</i> | false | Whether the referenced data module is active |
| <i>className</i> | DATAMODREF | Identifies the object as an instance of the DataModule class (Property discussed in Chapter 5, "Core language.") |
| <i>dataModClass</i> | | The class name of the data module |
| <i>filename</i> | | The name of the file containing the data module class |
| <i>parent</i> | null | Container form or report (Property discussed in Chapter 5, "Core language.") |
| <i>ref</i> | null | A reference to the data module object |
| <i>share</i> | None | How to share the data module |

Description A DataModRef object is used to access data modules. The *filename* property is set to the .DMD file that contains the data module class definition. The *dataModClass* property is set to the class name of the desired data module. Then the *active* property is set to *true* to activate the data module.

If the *share* property is All instead of the None, any existing instance of the desired data module class is used. Otherwise a new instance is created. A reference to the data module is assigned to the *ref* property.

When a DataModRef object is activated in the Form designer, the DataModule object's *rowset* property is assigned to the form's *rowset* property. Therefore you can access the form's primary rowset, and all other rowsets relative to it, in the same way, whether you're using a data module or not. To reference the queries in the data module from the form, you have to go through two additional levels of objects. For example, instead of:

```
form.query1.rowset
```

you would have to use:

```
form.dataModRef1.ref.query1.rowset
```

However, if *query1.rowset* was the primary rowset of the data module, you would still use:

```
form.rowset
```

anyway, and in *query1.rowset*'s event handlers, you would still use:

```
this.parent.parent.query2.rowset
```

to access *query2.rowset* whether you're using a data module or not, because the two Query objects are in the same relative position in the object containership hierarchy.

Example The following code excerpt from a form class uses the data module shown in the example for class DataModule, which is stored in the file "teacher classes students.DMD"

```
this.DATAMODREF1 = new DATAMODREF()
this.DATAMODREF1.parent = this
with (this.DATAMODREF1)
    filename = "teacher classes students.dmd"
    dataModClass = "TeacherClassesStudentsDataModule"
    share = 0
    active = true
endwith
```

See also class DataModule

class DbError

An object that describes a BDE or server error.

Syntax These objects are created automatically by *Visual* dBASE when a DbException occurs.

Properties The following table lists the properties of the DbError class. (No events or methods are associated with this class.)

| Property | Default | Description |
|-------------------|--------------|---|
| <i>className</i> | DBERROR | Identifies the object as an instance of the DbError class (Property discussed in Chapter 5, "Core language.") |
| <i>code</i> | | BDE error number |
| <i>context</i> | | Field name, table name, and so on, that caused error |
| <i>message</i> | Empty string | Text to describe the error |
| <i>nativeCode</i> | | Server error code |

Description When an error using a data access object occurs, a DbException is generated. Its *errors* property points to an array of DbError objects.

Each DbError object describes a BDE or SQL server error. If *nativeCode* is zero, the error is a BDE error. If *nativeCode* is non-zero, the error is a server error. The *message* property describes the error.

Example See class DbException.

See also class DbException, class Exception

class DbException

An object that describes a data access exception. DbException subclasses the Exception class.

Syntax These objects are created automatically by *Visual* dBASE when an exception occurs.

Properties The following table lists the properties of the DbException class. DbException objects also contain those properties inherited from the Exception class. (No events or methods are associated with the DbException class.)

| Property | Default | Description |
|------------------|-------------|---|
| <i>className</i> | DBEXCEPTION | Identifies the object as an instance of the DbException class (Property discussed in Chapter 5, "Core language.") |
| <i>errors</i> | | Array of DbError objects |

Description The DbException class is a subclass of the Exception class. It is generated when an error using a data access object occurs. In addition to the *Visual* dBASE error code and message, it provides access to BDE and SQL server error codes and messages.

Example The following statements attempt to apply cached updates. If there is an error and the code is compiled with the debug flag on, the errors are displayed in the result pane of the Command window.

```
try
    form.database1.applyUpdates()
catch ( DbException e )
    msgbox( "Cached updates failed to post", "Fatal error", 16 )
    #ifdef DEBUG
        local n
        for n = 1 to e.errors.size
            with e.errors[ n ]
                ? nativeCode, message
            endwhile
        endfor
    #endif
endtry
```

See also class DbError, class Exception

class DbfField

A field from a DBF (dBASE) table. DbfField subclasses the Field class.

Syntax These objects are created automatically by the rowset.

Properties The following table lists the properties of the DbfField class. DbfField objects also contain those properties inherited from the Field class. (No events or methods are associated with the DbfField class.)

| Property | Default | Description |
|----------------------|----------|--|
| <i>className</i> | DBFFIELD | Identifies the object as an instance of the DbfField class (Property discussed in Chapter 5, “Core language.”) |
| <i>decimalLength</i> | 0 | Number of decimal places if the field is a numeric field |
| <i>default</i> | | Default value for field (DBF7 only) |
| <i>maximum</i> | | Maximum allowed value for field (DBF7 only) |
| <i>minimum</i> | | Minimum allowed value for field (DBF7 only) |
| <i>readOnly</i> | false | Specifies whether the field has read-only access |
| <i>required</i> | false | Whether the field must be filled in (DBF7 only) |

Description The DbfField class is a subclass of the Field class. It represents a field from a DBF (dBASE) table, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

See also class Field, class PdxField, class Rowset, class SqlField

class Field

A base class object that represents a field from a table and can be used as a calculated field.

Syntax [*<oRef>* =] new Field()

<oRef> A variable or property in which to store the reference to the newly created Field object for use as a calculated field.

Properties The following tables list the properties, events, and methods of the Field class.

| Property | Default | Description |
|---------------------|--------------|---|
| <i>className</i> | FIELD | Identifies the object as an instance of the Field class (Property discussed in Chapter 5, "Core language.") |
| <i>fieldName</i> | | Name of the field the Field object represents, or the assigned calculated field name |
| <i>length</i> | | Maximum length |
| <i>lookupRowset</i> | | Reference to lookup table for field |
| <i>lookupSQL</i> | | SQL SELECT statement for field lookup values |
| <i>parent</i> | null | <i>fields</i> array that contains the object (Property discussed in Chapter 5, "Core language.") |
| <i>type</i> | Character | The field's data type |
| <i>value</i> | Empty string | Represents current value of field in row buffer |

| Event | Parameters | Description |
|-----------------------|--------------------------|--|
| <i>beforeGetValue</i> | | When <i>value</i> property is to be read; return value is used as <i>value</i> |
| <i>canChange</i> | <i><new value></i> | When attempting to change <i>value</i> property; return value allows or disallows change |
| <i>onChange</i> | | After <i>value</i> property is successfully changed |
| <i>onGotValue</i> | | After <i>value</i> is read |

| Method | Parameters | Description |
|---------------------------|--|--|
| <i>copyToFile()</i> | <i><filename expC></i> | Copies data from BLOB field to external file |
| <i>replaceFromFile()</i> | <i><filename expC></i> <i>[, <append expL>]</i> | Copies data from external file to BLOB field |

Description The Field class acts as the base class for the DbfField (dBASE), PdxField (Paradox), and SqlField (everything else) classes. It contains the properties common to all field types. Each subclass contains the properties specific to that table type. You also create calculated fields with a Field object.

Each rowset has a *fields* property, which points to an array. Each element of that array is an object of one of the subclasses of the Field class, depending on the

table type or types contained in the rowset. Each field object corresponds to one of the fields returned by the query or stored procedure that created the rowset.

While the *fieldName*, *length*, and *type* properties describe the field and are the same from row to row, the *value* property is the link to the field's value in the table. The *value* property's value reflects the current value of that field for the current row in the row buffer; assigning a value to the *value* property assigns that value to the row buffer. The buffer is not written to disk unless the rowset's *save()* method is explicitly called or there is an implicit save, which is usually caused by navigation in the rowset. You can abandon any changes you make to the row buffer by calling the rowset's *abandon()* method.

You may assign a Field object to the *dataLink* property of a control on a form. This makes the control data-aware, and causes it to display the current value of the Field object's *value* property; if changes are made to the control, the new value is written to the Field object's *value* property.

Calculated fields Use a calculated field to generate a value based on one or more fields, or some other calculation. For example, in a line item table with both the quantity ordered and price per item, you can calculate the total price for that line item. There would be no need to actually store that total in the table, which wastes space.

Because a calculated field is treated like a field in most respects, you can do things like *dataLink* it to a control on a form, show it in a grid, or use it in a report. Because a calculated field does not actually represent a field in a table, writing to its *value* property directly or changing its value through a *dataLinked* control never causes a change in a table.

To create a calculated field, create a new Field object and assign it a *fieldName*, then *add()* it to the *fields* array of a Rowset object.

Note You must assign the *fieldName* before adding the field to the *fields* array.

Because a rowset is not valid until its query opens, you must make the query active before you add the Field object. The query's *onOpen* event, which fires after the query is activated, is a good place to create the calculated field. To set the value of a calculated field, you can do one of two things

- Assign a code-reference, either a codeblock or function pointer, to the Field object's *beforeGetValue* event. The return value of the code becomes the Field object's value.
- Assign a value to the Field object's *value* property directly as needed, like in the rowset's *onNavigate* event.

Example The following example creates a calculated field, using the Field's *beforeGetValue* event to calculate the total price from the quantity and price per item for each line item:

```
q = new Query()
q.sql := "select * from LINEITEM"
q.active := true

c = new Field()
```

```

c.fieldName := "Total"
q.rowset.fields.add( c )
c.beforeGetValue := (||this.parent["Quantity"].value * this.parent["PricePer"].value)

```

Because *this* refers to the Field object itself, *this.parent* refers to the *fields* array, through which you can access the other field objects.

See also class DbfField, class PdxField, class Rowset, class SqlField

class LockField

A _DBASELOCK field in a DBF table.

Syntax These objects are created automatically by the rowset.

Properties The following table lists the properties of the LockField class. (No events or methods are associated with this class.)

| Property | Default | Description |
|------------------|------------|---|
| <i>className</i> | LOCKFIELD | Identifies the object as an instance of the LockField class (Property discussed in Chapter 5, "Core language.") |
| <i>fieldName</i> | _DBASELOCK | Name of the field the LockField object represents (read-only) |
| <i>lock</i> | | Date and time of last row lock |
| <i>parent</i> | null | <i>fields</i> array that contains the object (Property discussed in Chapter 5, "Core language.") |
| <i>update</i> | | Date and time of last row update |
| <i>user</i> | | Name of user that last locked or updated the row |

Description A LockField object is used to represent the _DBASELOCK field in a DBF table that has been CONVERTed. By examining the properties of a LockField object, you may determine the nature of the last row lock or update.

When a row is locked, either explicitly or automatically, the time, date, and login name of the user placing the lock are stored in the _DBASELOCK field of that row. When a file is locked, this same information is stored in the _DBASELOCK field of the first physical record in the table.

If a DBF table has a _DBASELOCK field, the LockField object is always the last field in the *fields* array, and is referenced by its field name, "_DBASELOCK".

All the properties of a LockField object are read-only.

See also class Field, CONVERT

class Parameter

A parameter for a stored procedure.

Syntax These objects are created automatically by the stored procedure.

Properties The following table lists the properties of the Parameter class. (No events or methods are associated with this class.)

| Property | Default | Description |
|------------------|-----------|---|
| <i>className</i> | PARAMETER | Identifies the object as an instance of the Parameter class (Property discussed in Chapter 5, “Core language.”) |
| <i>type</i> | Input | The parameter type (0=Input, 1=Output, 2=InputOutput, 3=Result) |
| <i>value</i> | | The value of the parameter |

Description Parameter objects represent parameters to stored procedures. Each element of the *params* array of a StoredProc object is a Parameter object. The Parameter objects are automatically created when the *procedureName* property is set, either by getting the parameter names for that stored procedure from the SQL server or by using parameter names specified directly in the *procedureName* property.

A parameter may be one of four types, as indicated by its *type* property:

- **Input:** an input value for the stored procedure. The *value* must be set before the stored procedure is called.
- **Output:** an output value from the stored procedure. The *value* must be set to the correct data type before the stored procedure is called; any dummy value may be used. Calling the stored procedure sets the *value* property to the output value.
- **InputOutput:** both input and output. The *value* must be set before the stored procedure is called. Calling the stored procedure updates the *value* property with the output value.
- **Result:** the result value of the stored procedure. In this case, the stored procedure acts like a function, returning a single result value, instead of updating parameters that are passed to it. Otherwise, the *value* is treated like an output value. The name of the Result parameter is always “Result”.

A Parameter object may be assigned as the *dataLink* of a component in a form. Changes to the component are reflected in the *value* property of the Parameter object, and updates to the *value* property of the Parameter object are displayed in the component.

Example The following statements call a stored procedure that returns an output parameter. The result is displayed in the result pane of the Command window.

```
d = new Database()
d.databaseName = "IBLOCAL"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "DEPT_BUDGET"
p.params[ "DNO" ].value = "670"           // Set input parameter
```

```
p.active = true
? p.params[ "TOT" ].value           // Display output
```

The following statements call a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT      0
#define PARAMETER_TYPE_OUTPUT    1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT    3

d = new Database()
d.databaseName = "WIDGETS"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "PROJECT_SALES( :month, :units )"
p.params[ "month" ].type = PARAMETER_TYPE_INPUT
p.params[ "month" ].value = 6
p.params[ "units" ].type = PARAMETER_TYPE_OUTPUT
p.params[ "units" ].value = 0           // Output will be numeric
p.active = true
? p.params[ "TOT" ].value           // Display output
```

See also class StoredProc

class PdxField

A field from a DB (Paradox) table. PdxField subclasses the Field class.

Syntax These objects are created automatically by the rowset.

Properties The following table lists the properties of the PdxField class. PdxField objects also contain those properties inherited from the Field class. (No events or methods are associated with the PdxField class.)

| Property | Default | Description |
|--------------------|--------------|--|
| <i>className</i> | PDXFIELD | Identifies the object as an instance of the PdxField class (Property discussed in Chapter 5, "Core language.") |
| <i>default</i> | | Default value for field |
| <i>lookupTable</i> | Empty string | Table to use for lookup value |
| <i>lookupType</i> | Empty string | Type of lookup |
| <i>maximum</i> | | Maximum allowed value for field |
| <i>minimum</i> | | Minimum allowed value for field |
| <i>picture</i> | Empty string | Formatting template |

| Property | Default | Description |
|-----------------|---------|--|
| <i>required</i> | false | Whether the field must be filled in |
| <i>readOnly</i> | false | Whether the field has read-only access |

Description This class is called PdxField—not “DbField”—to avoid confusion and simple typographical errors between it and the DbField class.

The PdxField class is a subclass of the Field class. It represents a field from a DB (Paradox) table, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

See also class DbField, class Field, class Rowset, class SqlField

class Query

A representation of an SQL statement that describes a query and contains the resulting rowset.

Syntax [*<ORef>* =] new Query()

<ORef> A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Query object.

Properties The following tables list the properties, events, and methods of the Query class.

| Property | Default | Description |
|---------------------|--------------|---|
| <i>active</i> | false | Whether the query is open and active or closed |
| <i>className</i> | QUERY | Identifies the object as an instance of the Query class (Property discussed in Chapter 5, “Core language.”) |
| <i>constrained</i> | false | Whether the WHERE clause of the SQL SELECT statement will be enforced when attempting to update Standard tables |
| <i>database</i> | null | Database to which the query is assigned |
| <i>handle</i> | | BDE statement handle |
| <i>masterSource</i> | null | Query that acts as master query and provides parameter values |
| <i>params</i> | AssocArray | Associative array that contains parameter names and values for the SQL statement |
| <i>parent</i> | null | Container form or report (Property discussed in Chapter 5, “Core language.”) |
| <i>requestLive</i> | true | Whether you want a writable rowset |
| <i>rowset</i> | null | Results of the query |
| <i>session</i> | null | Session to which the query is assigned |
| <i>sql</i> | Empty string | SQL statement that describes the query |

| Property | Default | Description |
|-----------------------|-----------|--|
| <i>unidirectional</i> | false | Whether to assume forward-only navigation to increase performance on SQL-based servers |
| <i>updateWhere</i> | AllFields | Enum to determine which fields to use in constructing the WHERE clause of an SQL UPDATE statement, used for posting changes to SQL-based servers |

| Event | Parameters | Description |
|-----------------|------------|--|
| <i>canClose</i> | | When attempting to close query; return value allows or disallows closure |
| <i>canOpen</i> | | When attempting to open query; return value allows or disallows opening |
| <i>onClose</i> | | After query closes |
| <i>onOpen</i> | | After query first opens |

| Method | Parameters | Description |
|--------------------|------------|---|
| <i>execute()</i> | | Executes query (called implicitly when <i>active</i> property is set to <i>true</i>) |
| <i>prepare()</i> | | Prepares SQL statement |
| <i>query()</i> | | Rebinds and executes SQL statement |
| <i>unprepare()</i> | | Cleans up when query is deactivated (called implicitly when <i>active</i> property is set to <i>false</i>) |

Description The Query object is where you specify which fields you want from which rows in which tables and the order in which you want to see them, through an SQL SELECT statement stored in the query's *sql* property. The results are accessed through the query's *rowset* property. To use a stored procedure that results in a rowset, use a StoredProc object instead.

Whenever you create a query object, it is initially assigned to the default database in the default session. If you want to use Standard tables in the default session you don't have to do anything with that query's *database* or *session* properties. If you want to use a Standard table in another session, assign that session to the query's *session* property, which causes that session's default database to be assigned to that query.

For non-Standard tables, you will need to set up a BDE alias for the database if you haven't done so already. After creating a new Database object, you may assign it to another session if desired; otherwise it is assigned to the default session. Once the Database object is active, you can assign it to the query's *database* property. If the database is assigned to another session, you need to assign that session to the query's *session* property first.

After the newly created query is assigned to the desired database, an SQL SELECT statement describing the data you want is assigned to the query's *sql* property.

If the SQL statement contains parameters, the Query object's *params* array is automatically populated with the corresponding elements. The value of each

array element must be set before the query is activated. A Query with parameters can be used as a detail query in a master-detail relationship through the *masterSource* property.

Setting the Query object's *active* property to *true* opens the query and executes the SQL statement stored in the *sql* property. If the SQL statement fails, for example the statement is misspelled or the named table is missing, an error is generated and the *active* property remains *false*. If the SQL statement executes but does not generate any rows, the *active* property is *true* and the *endOfSet* property of the query's *rowset* is *true*. Otherwise the *endOfSet* property is *false*, and the rowset contains the resulting rows.

Setting the *active* property to *false* closes the query, writing any buffered changes.

Example The first example opens a table named VACATION.DBF:

```
q= new Query()
q.sql = "select * from VACATION"
q.active = true
```

The second example opens a table named REQS in a database named PERSONNEL in a new session with a preset user name and password:

```
s1 = new Session()
d1 = new Database()
d1.databaseName = "PERSONNEL"
d1.session = s1
d1.loginString = "visitor/jobsavail"
d1.active = true
q1 = new Query()
q1.session = s1
q1.database = d1
q1.sql = "select * from REQS"
q1.active = true
```

The third example uses an SQL statement with parameters. Note that the parameter name is case-sensitive; the name in the *params* array must match the name in the SQL statement:

```
q1 = new Query()
q1.sql = "select * from CUSTOMER where STATE = :state"
q1.params[ "state" ] = "VA"
q1.active = true
```

See also class Database, class Rowset, class Session

class Rowset

The data that results from an SQL statement in a Query object.

Syntax These objects are created automatically by the query.

Properties The following tables list the properties, events, and methods of the Rowset class.

| Property | Default | Description |
|-----------------------|-----------------------|--|
| <i>autoEdit</i> | true | Whether the rowset automatically switches to Edit mode when a change is made in a <i>dataLinked</i> component. |
| <i>className</i> | ROWSET | Identifies the object as an instance of the Rowset class (Property discussed in Chapter 5, “Core language.”) |
| <i>endOfSet</i> | | Whether the row cursor is at either end of the set |
| <i>fields</i> | | Array of field objects in row |
| <i>filter</i> | Empty string | Filter SQL expression |
| <i>filterOptions</i> | Match length and case | Enum designating how the filter expression should be applied |
| <i>handle</i> | | BDE cursor handle |
| <i>indexName</i> | Empty string | Active index tag |
| <i>live</i> | true | Whether the data can be modified |
| <i>locateOptions</i> | Match length and case | Enum designating how the locate expression should be applied |
| <i>masterFields</i> | Empty string | Field list for master-detail link |
| <i>masterRowset</i> | null | Reference to master Rowset object |
| <i>modified</i> | false | Whether the row has changed |
| <i>notifyControls</i> | true | Whether to automatically update <i>dataLinked</i> controls |
| <i>parent</i> | null | Query object that contains the Rowset object (Property discussed in Chapter 5, “Core language.”) |
| <i>state</i> | Closed | Enum that describes the mode the rowset is in |

| Event | Parameters | Description |
|--------------------|------------|---|
| <i>canAbandon</i> | | When <i>abandon()</i> is called; return value allows or disallows abandoning of row |
| <i>canAppend</i> | | When <i>beginAppend()</i> is called; return value allows or disallows start of append |
| <i>canDelete</i> | | When <i>delete()</i> is called; return value allows or disallows deletion |
| <i>canEdit</i> | | When <i>beginEdit()</i> is called; return value allows or disallows switch to Edit mode |
| <i>canGetRow</i> | | When attempting to read row; return value acts as an additional filter |
| <i>canNavigate</i> | | When attempting row navigation; return value allows or disallows navigation |
| <i>canSave</i> | | When <i>save()</i> is called; return value allows or disallows saving of row |
| <i>onAbandon</i> | | After successful <i>abandon()</i> |

| Event | Parameters | Description |
|-------------------|--|---------------------------------------|
| <i>onAppend</i> | | After successful <i>beginAppend()</i> |
| <i>onDelete</i> | | After successful <i>delete()</i> |
| <i>onEdit</i> | | After successful <i>beginEdit()</i> |
| <i>onNavigate</i> | <i><method expN></i> , <i><rows expN></i> | After rowset navigation |
| <i>onSave</i> | | After successful <i>save()</i> |

| Method | Parameters | Description |
|--------------------------|---|--|
| <i>abandon()</i> | | Abandons pending changes to current row |
| <i>applyFilter()</i> | | Applies filter set during rowset's Filter mode |
| <i>applyLocate()</i> | [<i><locate expC></i>] | Finds first row that matches specified criteria |
| <i>atFirst()</i> | | Returns <i>true</i> if current row is first row in rowset |
| <i>atLast()</i> | | Returns <i>true</i> if current row is last row in rowset |
| <i>beginAppend()</i> | | Starts append of new row |
| <i>beginEdit()</i> | | Puts rowset in Edit mode, allowing changes to fields |
| <i>beginFilter()</i> | | Puts rowset in Filter mode, allowing entry of filter criteria |
| <i>beginLocate()</i> | | Puts rowset in Locate mode, allowing entry of search criteria |
| <i>bookmark()</i> | | Returns bookmark for current row |
| <i>bookmarksEqual()</i> | <i><bookmark 1></i> [, <i><bookmark 2></i>] | Compares two bookmarks or one bookmark with current row to see if they refer to same row |
| <i>clearFilter()</i> | | Disables filter created by <i>applyFilter()</i> and clears <i>filter</i> property |
| <i>clearRange()</i> | | Disables constraint created by <i>setRange()</i> |
| <i>count()</i> | | Returns number of rows in rowset, honoring filters |
| <i>delete()</i> | | Deletes current row |
| <i>findKey()</i> | <i><key exp></i> | Finds the row with the exact matching key value |
| <i>findKeyNearest()</i> | <i><key exp></i> | Finds the row with the nearest matching key value |
| <i>first()</i> | | Moves row cursor to first row in set |
| <i>flush()</i> | | Commits the rowset buffer to disk |
| <i>goto()</i> | <i><bookmark></i> | Moves row cursor to specified row |
| <i>last()</i> | | Moves row cursor to last row in set |
| <i>locateNext()</i> | [<i><rows expN></i>] | Finds other rows that match search criteria |
| <i>lockRow()</i> | | Locks current row |
| <i>lockSet()</i> | | Locks entire set |
| <i>next()</i> | [<i><rows expN></i>] | Navigates to adjacent rows |
| <i>refresh()</i> | | Refreshes entire rowset |
| <i>refreshControls()</i> | | Refreshes <i>dataLinked</i> controls |

| Method | Parameters | Description |
|---------------------|--|---|
| <i>refreshRow()</i> | | Refreshes current row only |
| <i>rowCount()</i> | | Returns logical row count if known |
| <i>rowNo()</i> | | Returns logical row number if known |
| <i>save()</i> | | Saves current row |
| <i>setRange()</i> | <key exp> or <startKey exp> null ,<endKey exp> null | Constrains the rowset to those rows whose key field values falls within a range |
| <i>unlock()</i> | | Releases locks set by <i>lockRow()</i> and <i>lockSet()</i> |

Description A Rowset object represents a set of rows that results from a query. It maintains a cursor that points to one of the rows in the set, which is considered the current row, and a buffer to manage the contents of that row. The row cursor may also point outside the set, either before the first row or after the last row, in which case it is considered to be at the end-of-set. Each row contains fields from one or more tables. These fields are represented by an array of Field objects that is represented by the rowset's *fields* property. For a simple query like the following, which selects all the fields from a single table with no conditions, the rowset represents all the data in the table:

```
select * from CUSTOMER
```

As the cursor moves from row to row, you can access the fields in that row.

A Query object always has a *rowset* property, but that rowset is not open and usable and does not contain any fields until the query has been successfully activated. Setting the Query object's *active* property to *true* opens the query and executes the SQL statement stored in the *sql* property. If the SQL statement fails, for example the statement is misspelled or the named table is missing, an error is generated and the *active* property remains *false*. If the SQL statement executes but does not generate any rows, the *active* property is *true* and the *endOfSet* property of the query's *rowset* is *true*. Otherwise the *endOfSet* property is *false*, and the rowset contains the resulting rows.

Once the rowset has been opened, you can do any of the following:

- Navigate the rowset; that is, move the row cursor
- Filter and search for rows
- Add, modify, and delete rows
- Explicitly lock individual rows or the entire set
- Get information about the rowset, including row cursor's current position

The individual Field objects in a rowset's *fields* array property may be *dataLinked* to controls on a form. As the row cursor is navigated from row to row, the controls will be updated with the current row's values, unless the rowset's *notifyControls* property is set to *false*. Changing the values shown in the controls will change the *value* property of the *dataLinked* Field objects. You may also directly modify the *value* property of the Field objects. All of the values are maintained in the row buffer.

Rowset objects support master-detail linking. Navigation and updates in the master rowset change the set of rows in the detail rowset. The detail rowset is controlled by changing the key range of an existing index in the detail rowset. The *masterRowset* and *masterFields* properties are set in the detail rowset. This allows a single master rowset to control any number of detail rowsets.

When a query opens, its rowset is in Browse mode. By default, a rowset's *autoEdit* property is *true*, which means that its fields are changeable through *dataLinked* controls. Typing a destructive key in a *dataLinked* control automatically attempts to switch the rowset into Edit mode. By setting *autoEdit* to *false*, the rowset is read-only, and the *beginEdit()* method must be called to switch to Edit mode and allow editing. *autoEdit* has no effect on assignments to the *value* of a field; they are always allowed.

The rowset's *modified* property indicates whether any changes have been made to the current row. Changes made to the row buffer are not written until the *save()* method is explicitly called or there is an implicit save, which can be caused by one of the following:

- Navigation in the rowset: calling *next()*, *first()*, *last()*, *goto()*, *findKey()*, *findKeyNearest()*, or *applyLocate(<SQL exp>)*; or assigning to *filter*
- A state or mode change in the rowset: calling *beginAppend()*, *beginFilter()*, or *beginLocate()*
- The rowset's query is closed.

In all cases, even with an explicit *save()* call, no attempt is made to save data if the rowset's *modified* property is *false*. This architecture lets you define row-validation code once in the *canSave* event handler that is called whenever it is needed and only when it is needed.

In addition to normal data access through Browse and Edit modes, the rowset supports three other modes: Append, Filter, and Locate, which are initiated by *beginAppend()*, *beginFilter()*, and *beginLocate()* respectively. At the beginning of all three modes, the row buffer is disassociated from whatever row it was buffering and cleared. This allows the entry of field values typed into *dataLinked* controls or assigned directly to the *value* property. In Append mode, these new values are saved as a new row if the row buffer is written. In Filter mode, executing an *applyFilter()* causes the non-blank field values to be used as criteria for filtering rows, showing only those that match. In Locate mode, calling *applyLocate()* causes the non-blank field values to be used as criteria to search for matching rows. In all three modes, using the field values cancels that mode. Also, calling the *abandon()* method causes the rowset to revert back to Browse mode without using the values.

You can easily implement filter-by-form and locate-by-form features with the Filter and Locate modes. Instead of using Filter mode, you can assign an SQL expression directly to the rowset's *filter* property. The rowset's *canGetRow* event will filter rows based on any dBASE code, not just an SQL expression, and can be used instead of or in addition to Filter mode and the *filter* property. You can also use *applyLocate()* without starting Locate mode first by passing an SQL expression to find the first row for which the expression is *true*.

Any row-selection criteria—from the WHERE clause of the query's SQL SELECT statement, the key range enforced by a master-detail link, or a filter—is actively enforced. *applyLocate()* will not find a row that does not match the criteria. When appending a new row or changing an existing row, if the fields in the row are written such that the row no longer matches the selection criteria, that row becomes out-of-set, and the row cursor moves to the next row, or to the end-of-set if there are no more matching rows. To see the out-of-set row, you must remove or modify the selection criteria to allow that row.

Row and set locking support varies among different table types. The Standard (DBF and DB) tables fully support locking, as do some SQL servers. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

Any attempt to change the data in a row, like typing a letter in a *dataLinked* Entryfield control, causes an automatic row lock to be attempted. If that row is already locked, the lock is retried up to the number of times specified by the session's *lockRetryCount* property; if after those attempts the lock is unsuccessful, the change does not take. If the automatic lock is successful, the lock remains until navigation off the locked row occurs or the row is saved or abandoned; then the lock is automatically removed.

Example The following code gives everyone an extra day of vacation:

```
q= new Query()
q.sql = "select * from EMPLOYEE"
q.active = true
do while not q.rowset.endOfSet
    q.rowset.fields[ "VacHours" ].value += 8
    q.rowset.next()
enddo
```

See also class Database, class Field, class Query, class Session

class Session

An object that manages simultaneous database access.

Syntax [*<oRef>* =] new Session()

<oRef> A variable or property—typically of a Form or Report object—in which to store a reference to the newly created Session object.

Properties The following tables list the properties, events, and methods of the Session class.

| Property | Default | Description |
|--------------------------|---------|---|
| <i>className</i> | SESSION | Identifies the object as an instance of the Session class (Property discussed in Chapter 5, “Core language.”) |
| <i>handle</i> | | BDE session handle |
| <i>lockRetryCount</i> | 0 | Number of times to retry a failed lock attempt |
| <i>lockRetryInterval</i> | 0 | Number of seconds to wait between each lock attempt |
| <i>parent</i> | null | Container form or report (Property discussed in Chapter 5, “Core language.”) |

| Event | Parameters | Description |
|-------------------|-----------------------------------|--|
| <i>onProgress</i> | <percent expN>, <message expC> | Periodically during data processing operations |

| Method | Parameters | Description |
|----------------------|--|---|
| <i>access()</i> | | Returns the user’s access level for the session |
| <i>addPassword()</i> | <password expC> | Adds a password to the password table for access to encrypted DB (Paradox) tables |
| <i>login()</i> | <group expC>, <user expC>, <password expC> | Logs the specified user into the session to access encrypted DBF (dBASE) tables |
| <i>user()</i> | | Returns the user’s login name for the session |

Description A session represents a separate user task, and is required primarily for DBF and DB table security. *Visual* dBASE supports up to 2048 simultaneous sessions. When *Visual* dBASE first starts, it already has a default session.

DBF and DB table security is session-based. (SQL-table security is database-based.) When you create a new Session object, it copies the security settings of the default session. Therefore, if you have the user log in when *Visual* dBASE starts, all the new sessions you create to handle multiple tasks will have the access level.

Unlike the Database and Query objects, a Session object does not have an *active* property. Sessions are always active. To close a session, you must destroy it by releasing all references to it.

Example This example assigns a query that accesses the encrypted PAYROLL.DBF table to a new Session object. When the query is activated, a password dialog will be automatically displayed by *Visual* dBASE to access the table.

```
s1 = new Session()
q1 = new Query()
q1.session = s1
q1.sql = "select * from PAYROLL"
q1.active = true
```


See also class Database, class Query, class Rowset

class SqlField

A field from an SQL-server-based table. SqlField subclasses the Field class.

Syntax These objects are created automatically by the rowset.

Properties The following table lists the properties of the SqlField class. SqlField objects also contain those inherited from the Field class. (No events or methods are associated with the SqlField class.)

| Property | Default | Description |
|------------------|----------|--|
| <i>className</i> | SQLFIELD | Identifies the object as an instance of the SqlField class (Property discussed in Chapter 5, “Core language.”) |
| <i>precision</i> | | The number of digits of precision |
| <i>scale</i> | | How the number is scaled |

Description The SqlField class is a subclass of the Field class. It represents a field from an SQL-server-based table, including any ODBC connection, and contains properties that are specific to fields of that table type. Otherwise it is considered to be a Field object.

See also class DbfField, class Field, class PdxField, class Rowset

class StoredProc

A representation of a stored procedure call.

Syntax [*<oRef>* =] new StoredProc()

<oRef> A variable or property—typically of a Form or Report object—in which to store a reference to the newly created StoredProc object.

Properties The following tables list the properties, events, and methods of the StoredProc class.

| Property | Default | Description |
|------------------|------------|--|
| <i>active</i> | false | Whether the stored procedure is open and active or closed |
| <i>className</i> | STOREDPROC | Identifies the object as an instance of the StoredProc class (Property discussed in Chapter 5, “Core language.”) |
| <i>database</i> | null | Database to which the stored procedure is assigned |
| <i>handle</i> | | BDE statement handle |

| Property | Default | Description |
|----------------------|--------------|---|
| <i>params</i> | AssocArray | Associative array that contains Parameter objects for the stored procedure call |
| <i>parent</i> | null | Container form or report (Property discussed in Chapter 5, "Core language.") |
| <i>procedureName</i> | Empty string | Name of the stored procedure |
| <i>rowset</i> | null | Results of the stored procedure call |
| <i>session</i> | null | Session to which the stored procedure is assigned |

| Event | Parameters | Description |
|-----------------|------------|---|
| <i>canClose</i> | | When attempting to close stored procedure; return value allows or disallows closure |
| <i>canOpen</i> | | When attempting to open stored procedure; return value allows or disallows opening |
| <i>onClose</i> | | After stored procedure closes |
| <i>onOpen</i> | | After stored procedure first opens |

| Method | Parameters | Description |
|--------------------|------------|--|
| <i>execute()</i> | | Executes stored procedure (called implicitly when <i>active</i> property is set to <i>true</i>) |
| <i>prepare()</i> | | Prepares stored procedure call |
| <i>query()</i> | | Rebinds and executes stored procedure |
| <i>unprepare()</i> | | Cleans up when stored procedure is deactivated (called implicitly when <i>active</i> property is set to <i>false</i>) |

Description Use a StoredProc object to call a stored procedure in a database. Most stored procedures take one or more parameters as input and may return one or more values as output. Parameters are passed to and from the stored procedure through the StoredProc object's *params* property, which points to an associative array of Parameter Objects.

Some stored procedures return a rowset. In that case, the StoredProc object is similar to a Query object; but instead of executing an SQL statement that describes the data to retrieve, you name a stored procedure, pass parameters to it, and execute it. The resulting rowset is accessed through the StoredProc object's *rowset* property, just like in a Query object.

Because stored procedures are SQL-server-based, you must create and activate a Database object and assign that object to the StoredProc object's *database* property. Standard tables do not support stored procedures.

Next, the *procedureName* property must be set to the name of the stored procedure. For most SQL servers, the BDE can get the names and types of the parameters for the stored procedure. On some servers, no information is available; in that case you must include the parameter names in the *procedureName* property as well.

Getting or specifying the names of the parameters automatically creates the corresponding elements in the StoredProc object's *params* array. Each element is a Parameter object. Again, for some servers, information on the parameter types is available. For those servers, the *type* properties are automatically filled in and the *value* properties are initialized. For other servers, you must supply the missing *type* information and initialize the *value* to the correct type.

To call the stored procedure, set its *active* property to *true*. If the stored procedure does not generate a rowset, the *active* property is reset to *false* after the stored procedure executes and returns its results, if any. This facilitates calling the stored procedure again if desired, after reading the results from the *params* array.

If the stored procedure generates a rowset, the *active* property remains true, and the resulting rowset acts just like a rowset generated by a Query object.

You can *dataLink* components in a form to fields in a rowset, or to the Parameter objects in the *params* array.

Example The following statements call a stored procedure that returns an output parameter. The result is displayed in the result pane of the Command window.

```
d = new Database()
d.databaseName = "IBLOCAL"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "DEPT_BUDGET"
p.params[ "DNO" ].value = "670"
p.active = true
? p.params[ "TOT" ].value           // Display output
```

The following statements call a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT      0
#define PARAMETER_TYPE_OUTPUT    1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT    3

d = new Database()
d.databaseName = "WIDGETS"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "PROJECT_SALES( :month, :units )"
p.params[ "month" ].type = PARAMETER_TYPE_INPUT
p.params[ "month" ].value = 6
p.params[ "units" ].type = PARAMETER_TYPE_OUTPUT
p.params[ "units" ].value = 0           // Output will be numeric
p.active = true
? p.params[ "TOT" ].value           // Display output
```

See also class Parameter, class Query

class UpdateSet

An object that updates one table with data from another.

Syntax [*<oRef>* =] new UpdateSet()

<oRef> A variable or property in which to store a reference to the newly created UpdateSet object.

Properties The following tables list the properties and methods of the UpdateSet class. (No events are associated with this class.)

| Property | Default | Description |
|------------------------------|-----------|---|
| <i>changedTableName</i> | | Table to collect copies of original values of changed rows |
| <i>className</i> | UPDATESET | Identifies the object as an instance of the UpdateSet class (Property discussed in Chapter 5, “Core language.”) |
| <i>destination</i> | | Rowset object or table name that is updated or created |
| <i>indexName</i> | | Name of index to use |
| <i>keyViolationTableName</i> | | Table to collect rows with duplicate primary keys |
| <i>problemTableName</i> | | Table that collects problem rows |
| <i>source</i> | | Rowset object or table name that contains updates |

| Method | Parameters | Description |
|------------------------|------------|---|
| <i>append()</i> | | Adds new rows |
| <i>appendUpdate()</i> | | Updates existing rows and adds new rows |
| <i>copy()</i> | | Creates destination table |
| <i>delete()</i> | | Deletes rows in destination that match rows in source |
| <i>update()</i> | | Updates existing rows |

Description The UpdateSet object is used to update data from one rowset to another, or to copy or convert data from one format to another, either in the same database or across databases.

To update a DBF table with *appendUpdate()*, *delete()*, or *update()*, the *indexName* property of the UpdateSet object must be set to a valid index. To update a DB table with the same operations, the DB table’s primary key is used by default, or you can assign a secondary index to the *indexName* property.

The *source* and *destination* can be either a character string containing the name of a table, or an object reference to a rowset. If the source is a rowset, the data used in the update can be filtered.

For Standard table names, specify the name of the table and the extension (DBF or DB). For all other tables, place the database name (the BDE alias) in colons before the table name; that is, in this form:

```
:alias:table
```

The named database must be open when the *UpdateSet*() method is executed.

Example The following example copies the result set from a query on an SQL-based-server to a local DBF file:

```
d = new Database()
d.databaseName = "SOMESQL"
d.active = true
q = new Query()
q.database = d
q.sql = "select * from SOMETABLE where THIS = 'that' order by ID"
q.active = true

u = new UpdateSet()
u.source = q.rowset
u.destination = "RESULTS.DBF"
u.copy()
```

This example copies all the rows from the same SQL-based-server table to a local DBF file without using a Query object:

```
d = new Database()
d.databaseName = "SOMESQL"
d.active = true

u = new UpdateSet()
u.source = ":SOMESQL:SOMETABLE"
u.destination = "SOMEDUP.DBF"
u.copy()
```

See also class Database, class Rowset

abandon()

Abandons any pending changes to the current row.

Syntax <oRef>.abandon()
<oRef> The rowset whose current row buffer you want to abandon.

Property of Rowset

Description Changes made to a row, either through *dataLinked* controls or by assigning values to the *value* property of fields, are not written to disk until the rowset's *save()* method is explicitly called or there is an implicit save, which is usually caused by navigation in the rowset. You can discard any pending changes to

the rowset with the *abandon()* method. This is usually done in response to the user's request.

You can check the *modified* property first to see if there have been any changes made to the row. Calling *abandon()* when there's nothing to abandon has no ill effects (although the *canAbandon* and *onAbandon* events are still fired).

You may also want to discard unwritten changes when a query is closed, the opposite of the default behavior. If you are relying on the query's event handlers to do this instead of abandoning and closing the query through code, you must call *abandon()* during the query's *canClose* event and return *true* from the *canClose* event handler; calling *abandon()* during the *onClose* event will have no effect, since the *onClose* event fires after the query has already closed, and any changes have been written.

When using *abandon()* to discard changes to an existing row, all fields are returned to their original values and any *dataLinked* controls are automatically restored. If the row was automatically locked when editing began, it is unlocked.

You may also use *abandon()* to discard a new row created by the *beginAppend()* method, in which case the new row is discarded, and the row that was current at the time *beginAppend()* was called is restored. This is not considered navigation, so the rowset's *onNavigate* does not fire. If you have a *onNavigate* event handler, call it from the *onAbandon* event. *abandon()* also cancels a rowset's Filter or Locate mode in the same manner.

The order of events when calling *abandon()* is as follows:

- 1 If the rowset has a *canAbandon* event handler, it is called. If not, it's as if *canAbandon* returns *true*.
- 2 If the *canAbandon* event handler returns *false*, nothing else happens and *abandon()* returns *false*.
- 3 If the *canAbandon* event handler returns *true*:
 - 1 The current row buffer/state is abandoned, restoring the rowset to its previous row/state.
 - 2 The *onAbandon* event fires.
 - 3 *abandon()* returns *true*.

While *abandon()* discards unwritten changes to the current row, there are two mutually exclusive ways of abandoning changes to more than one row in more than one table in a database, which you can use instead of or in addition to single-row buffering. Calling *beginTrans()* starts transaction logging which logs all changes and allows you to undo them by calling *rollback()* if necessary. The alternative is to set the database's *cacheUpdates* property to *true* so that changes are written to a local cache but not written to disk, and then call *abandonUpdates()* to discard all the changes if needed.

Example The following *onClick* event handler for an Abandon button calls the *abandon()* method for the form's primary rowset:

```
function abandonButton_onClick()
    form.rowset.abandon()
```

See also *abandonUpdates()*, *beginAppend()*, *beginTrans()*, *cacheUpdates*, *canAbandon*, *canClose*, *endOfSet*, *modified*, *onAbandon*, *rollback()*, *save()*

abandonUpdates()

Abandons all cached updates in the database.

Syntax `<oRef>.abandonUpdates()`

<oRef> The database whose cached changes you want to abandon.

Property of Database

Description *abandonUpdates()* discards all changes to a database that have been cached. Unlike *applyUpdates()*, it cannot fail. See *cacheUpdates* for more information on caching updates.

Changes to the current row that have not been written are still in the row buffer, and have not been cached. To abandon changes made to the row buffer, call the rowset's *abandon()* method.

Example Suppose you have a form that's used for redeeming prizes for points accumulated for dining at the corporate cafeteria. As each prize is chosen, the choice is written to the prize redemption table, using cached updates. The points aren't actually spent until you press the Redeem button, and you can cancel all the choices that have been made and start over by pressing the Start Over button. The following is the *onClick* event handler for the Start Over button.

```
function startOverButton_onClick()
    form.rowset.parent.database.abandonUpdates() // Discard cached updates
    form.rowset.abandon()                       // and current choice
```

See also *abandon()*, *beginTrans()*, *cacheUpdates*, *rollback()*

access()

Returns the access level of the current session for DBF table security.

Syntax `<oRef>.access()`

<oRef> The session you want to test.

Property of Session

Description DBF table security is session-based. All queries assigned to the same session in their *session* property have the same access level.

`access()` returns a number from 0 to 8. 8 is the lowest level of access, 1 is the highest level of access, and 0 is returned if the session is not using DBF security.

Example The following method overrides the form's `open()` method. It checks the user's current access level and hides a button to display the administration form if the access level isn't high enough. An overriding method is used instead of the `onOpen` event because `onOpen` fires after the form has already opened on-screen, which means that the administration button might appear briefly before it vanishes.

```
function open()
    local nAccess
    nAccess = form.rowset.parent.session.access()
    if nAccess == 0 or nAccess > 4
        form.adminButton.visible = false
    endif
    return super::open()
```

The session is referenced via the form's primary rowset. The rowset's *parent* is the query object, which is assigned to a session. The access level is stored in a variable for convenience because it needs to be checked twice: first to make sure security is enabled, and second to check the access level itself.

See also `addPassword()`, `login()`, `user()`

active

Specifies whether an object is open and active or closed.

Property of Database, DataModRef, Query, StoredProc

Description When created, a new session's default database is active since it does not require any setup. Other Database, DataModRef, Query, and StoredProc objects do require setup, so their *active* property defaults to *false*. Once they have been set up, set their *active* property to *true* to open the object and make it active.

When a Query or StoredProc object's *active* property is set to *true*, its *canOpen* event is called. If there is no *canOpen* event handler, or the event handler returns *true*, the object is activated. In a Query object, the SQL statement in its *sql* property is executed; in a StoredProc object, the stored procedure named in its *procedureName* property is called. Then the object's *onOpen* event is fired.

To close the object, set its *active* property to *false*. Closing an object closes all objects below it in the class hierarchy. Attempting to close a Query or StoredProc object calls its *canClose* event. If there is no *canClose* event handler, or the event handler returns *true*, the object is closed. Closing a Database object closes all its Query and StoredProc objects. After the objects are closed, all the Query and StoredProc objects' *onClose* events are fired.

Activating and deactivating an object implicitly calls a number of advanced methods. You may override or completely redefine these methods for custom

data classes; in typical usage, don't touch them. When you set *active* to *true*, a Database object's *open*() method is called; activating a query or stored procedure calls *prepare*(), then *execute*(). When you set *active* to *false*, a Database object's *close*() method is called; deactivating a query or stored procedure calls its *unprepare*() method. These methods are called as part of the activation or deactivation of the object, before the *onOpen* or *onClose* event.

Closing a query or a StoredProc object that generated a rowset attempts to write any changes to its rowset's current row buffer, and to apply all cached updates or commit all logged changes. To circumvent this, you must call the *abandon*(), *abandonUpdates*(), and/or *rollback*() before the object's *onClose* event—for example, during the *canClose* event or before setting the *active* property to *false*—because *onClose* fires after the object has already closed.

Once an object has been closed, you may change its properties if desired and reopen it by setting its *active* property back to *true*.

See also *abandon*(), *abandonUpdates*(), *canClose*, *canOpen*, *onClose*, *onOpen*, *rollback*()

addPassword()

Adds a password to the session's password list for DB table security.

Syntax <oRef>.addPassword(<expC>)

<oRef> The session you want to receive the password.

<expC> The password string.

Property of Session

Description DB table security is based on password lists. If you know a password, you have access to all the files that use that password. There is no matching between a user name and password. The access level for each file may be different for the same password.

Password lists are session-based. Once a password has been added to a session, it will continue to be tried for all encrypted tables. All queries assigned to the same session in their *session* property use the same password list. If you attempt to open an encrypted table and there is no valid password that gives access to that table in the list, you will be prompted for the password. Responding with a password adds it to the list.

The *addPassword*() method allows you add passwords directly to the session's password list. You can do this if you want to add a default password, so that users won't be prompted, or if you're writing your own custom login form, and need to add the password to the session.

`append ()`

Example The following *onClick* event handler for the login button on a custom login form adds the password typed into the *password1* component, a custom entryfield that obscures text as it is typed, and runs the main form:

```
function loginButton_onClick()  
    form.rowset.parent.session.addPassword( form.password1.value )  
    do MAIN.WFM
```

See also *login()*

append()

Adds rows from one rowset or table to another.

Syntax `<oRef>.append()`

<oRef> The UpdateSet object that describes the append.

Property of UpdateSet

Description Use *append()* to add rows from a source rowset or table to an existing destination rowset or table. If there is no primary key in the destination, the rows from the source are always added. If there is a primary key in the destination, rows with keys that already exist in the destination will be copied to the table specified by the UpdateSet object's *keyViolationTableName* property instead.

To update rows with the same primary key in the destination, use the *appendUpdate()* method. To move data to a new table instead of an existing table or rowset, use the *copy()* method.

Example The following code accumulates records from the Daily table in an archive. The Archive table is occasionally moved to tape, so the code uses the *append()* or *copy()* method, depending on whether the Archive table already exists. The Daily table is stored in a database that supports the CURRENT_DATE SQL function.

```
d = new Database()  
d.databaseName = "TRAFFIC"  
d.loginString = "backup/murphy"  
d.active = true  
q = new Query()  
q.database = d  
q.sql = "select * from DAILY where POSTED = CURRENT_DATE"  
q.active = true  
  
u = new UpdateSet()  
u.source = q.rowset  
u.destination = "ARCHIVE.DBF"  
if _app.databases[ 1 ].tableExists( "ARCHIVE.DBF" )  
    u.append()  
else
```

```

        u.copy()
    endif

```

See also *appendUpdate(), copy(), destination, keyViolationTableName, source*

appendUpdate()

Updates one rowset or table from another by updating existing rows and adding new rows.

Syntax *<oRef>.appendUpdate()*

<oRef> The UpdateSet object that describes the update.

Property of UpdateSet

Description Use *appendUpdate()* to update a rowset, allowing new rows to be added. You must specify the UpdateSet object's *indexName* property which will be used to match the records. The index must exist for the destination rowset. The original values of all changed records will be copied to the table specified by the updateSet's *changedTableName* property.

To update existing rows only, use the *update()* method instead. To always add new rows, use the *append()* method.

See also *append(), destination, changedTableName, source, update()*

applyFilter()

Applies the filter that was set during a rowset's Filter mode.

Syntax *<oRef>.applyFilter()*

<oRef> The rowset whose filter criteria you want to apply.

Property of Rowset

Description Rowset objects support a Filter mode in which values can be assigned to Field objects and then used to filter the rows in a rowset to show only those rows with matching values. *beginFilter()* puts the rowset in Filter mode and *applyFilter()* applies the filter values. *clearFilter()* cancels the filter. Because *dataLinked* controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a filter-by-form feature in your application.

When *applyFilter()* is called, the row cursor is repositioned to the first matching row in the set, or to the end-of-set if there are no matches. The rowset's *filter*

`applyLocate ()`

property is updated to contain the resulting SQL expression used for the filter. *applyFilter()* returns *true* or *false* to indicate if a match was found.

To filter rows with a condition without using Filter mode, set the rowset's *filter* property directly. See the *filter* property for more information on how filters are applied to data. To filter rows with dBASE code instead of or in addition to an SQL expression, use the *canGetRow* event.

Example The following two event handlers demonstrate the basic filter-by-form functionality. First, before switching to Filter mode, get the bookmark for the current row:

```
function beginFilterButton_onClick()
    form.bookmark = form.rowset.bookmark()
    form.rowset.beginFilter()
    // Good place to visually indicate form is now Filter mode
```

Then when the user attempts to apply the filter criteria, you can try to go back to the row they were on:

```
function applyFilterButton_onClick()
    form.rowset.notifyControls := false // Suppress display of all this navigation
    if not form.rowset.applyFilter()
        form.rowset.clearFilter() // No matches, get rid of filter
        try
            form.rowset.goto( form.bookmark )
        catch ( Exception e ) // Couldn't go to saved row (maybe deleted)
            form.rowset.first() // Go to first row in rowset instead
        endtry
        msgbox( "No matches found", "Filter", 48 )
    endif
    // Undo any visual indications of Filter mode here
    form.rowset.notifyControls := true
    form.rowset.refreshControls() // Refresh controls on form
```

See also *beginFilter()*, *canGetRow*, *clearFilter()*, *endOfSet*, *filter*, *filterOptions*, *value*

applyLocate()

Finds the first row that matches specified criteria.

Syntax `<oRef>.applyLocate([<SQL condition expC>])`

<oRef> The rowset you want to search for the specified criteria.

<SQL condition expC> An SQL condition expression.

Property of Rowset

Description Rowset objects support a Locate mode in which values can be assigned to Field objects and then used to find rows in a rowset that contains matching values. *beginLocate()* puts the rowset in Locate mode and *applyLocate()* finds the first

matching row. *locateNext()* finds other matching rows. Because *dataLinked* controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a search-by-form feature in your application.

applyLocate() moves the row cursor to the first row that matches the criteria set during the rowset's Locate mode.

You may also use *applyLocate()* without calling *beginLocate()* first to put the rowset in Locate mode: call *applyLocate()* with a parameter string that contains an SQL condition expression. Doing so finds the first row that matches the condition. (Calling *applyLocate()* with a parameter when the rowset is in Locate mode discards any field values entered during Locate mode and uses the specified condition expression only to find a match.)

Calling *applyLocate()* with a parameter will attempt an implicit save if the rowset is not in Locate mode and the rowset's *modified* property is *true*. If the implicit save fails, because the *canSave* returns *false* or any other reason, the search is not attempted.

If a search is attempted, *applyLocate()* returns *true* or *false* to indicate if a match is found. *onNavigate* always fires after a search attempt, either on the first matching row, or the current row if the search failed.

applyLocate() will use available indexes to find a match more quickly. When searching on the current index specified by the rowset's *indexName* property, you may find the *findKey()* and *findKeyNearest()* methods more convenient and direct.

Example The following statement finds the first row where the City field matches the value typed into a Entryfield component in a form. Note the use of single quotation marks to delimit the value of the Entryfield component.

```
form.rowset.applyLocate( "CITY = '" + form.cityText.value + "'" )
```

See also *beginLocate()*, *endOfSet*, *findKey()*, *findKeyNearest()*, *locateNext()*, *locateOptions*, *value*

applyUpdates()

Attempts to apply all cached updates in the database.

Syntax <oRef>.applyUpdates()

<oRef> The database whose cached updates you want to apply.

Property of Database

Description *applyUpdates()* attempts to apply all changes to a database that have been cached and returns *true* or *false* to indicate success or failure. If it succeeds, all cached updates are cleared; if it fails, the updates remain cached. Since

`atFirst()`

`applyUpdates()` uses a transaction while attempting to apply the changes and you cannot nest transactions in a database, cached updates and transaction logging with `beginTrans()` are mutually exclusive. See `cacheUpdates` for more information on caching updates.

Changes to the current row that have not been written are still in the row buffer, and have not been cached. To apply changes made to the row buffer, call the rowset's `save()` method before you call `applyUpdates()`.

Example Suppose you have a form that's used for redeeming prizes for points accumulated for dining at the corporate cafeteria. As each prize is chosen, the choice is written to the prize redemption table, using cached updates. The points aren't actually spent until you press the Redeem button, and you can cancel all the choices that have been made and start over by pressing the Start Over button. The following is the `onClick` event handler for the Redeem button.

```
function redeemButton_onClick()  
    if form.rowset.save()                // Save current row  
        form.rowset.parent.database.applyUpdates() // Apply cached updates  
    endif
```

See also `abandonUpdates()`, `beginTrans()`, `cacheUpdates`, `save()`

atFirst()

Returns *true* if the row cursor is at the first row in the rowset.

Syntax `<oRef>.atFirst()`

<oRef> The rowset whose position you want to check.

Property of Rowset

Description Use `atFirst()` to determine if the row cursor is at the first row in the rowset. When `atFirst()` returns *true*, the row cursor is at the first row. In most cases, `atFirst()` is an inexpensive operation. The current row is usually compared with a bookmark of the first row made when the query is first opened. However, `atFirst()` may be time-consuming for certain data drivers.

A common use of `atFirst()` is to conditionally disable backward navigation controls. If you know you are on the first row, you can't go backward, and you reflect this visually with a disabled control.

The end-of-set is different from the first row, so `endOfSet` cannot be *true* if `atFirst()` returns *true*. `endOfSet` is *true* if the row cursor is before the first row in the rowset (or after the last row).

Example The following `onNavigate` event handler sets the *enabled* properties of the navigation buttons on a form, based on the return values of `atFirst()` and `atLast()`.

```

function Rowset_onNavigate
  if this.endOfSet
    return          // Do nothing if end-of-set
  endif
  local lBackward, lForward
  lBackward = not this.atFirst()
  lForward  = not this.atLast()
  with this.parent.parent
    buttonFirst.enabled := lBackward
    buttonPrev.enabled  := lBackward
    buttonNext.enabled  := lForward
    buttonLast.enabled  := lForward
  endwhile
endwith

```

The event handler does nothing if the rowset is at the end-of-set, expecting that the row cursor will be moved in the reverse direction of the navigation. If the navigation attempt was forward, the row cursor would be moved back to the last row, and if the navigation attempt was backward, the row cursor would be moved forward to the first row. In this way, the rowset is never on the end-of-set. This technique cannot be used for rowsets where there may not be any matching rows.

See also *atLast(), endOfSet, rowNo()*

atLast()

Returns *true* if the row cursor is at the last row in the rowset.

Syntax *<oRef>.atLast()*

<oRef> The rowset whose position you want to check.

Property of Rowset

Description Use *atLast()* to determine if the row cursor is at the last row in the rowset. When *atLast()* returns *true*, the row cursor is at the last row. *atLast()* may be an expensive operation. For example, if you have not navigated to the last row in a rowset returned by an SQL server, such a navigation would have to be attempted to determine if you are at the last row, which could be time-consuming for large rowsets.

A common use of *atLast()* is to conditionally disable forward navigation controls. If you know you are on the last row, you can't go forward, and you reflect this visually with a disabled control.

The end-of-set is different from the last row, so *endOfSet* cannot be *true* if *atLast()* returns *true*; *endOfSet* is *true* if the row cursor is after the last row in the rowset (or before the first row).

Example See *atFirst()*.

See also *atFirst()*, *endOfSet*, *rowNo()*

autoEdit

Specifies whether the rowset automatically switches to Edit mode when changes are made through *dataLinked* components.

Property of Rowset

Description When a query (or stored procedure) is activated, its rowset opens in Browse mode. If a rowset's *autoEdit* property is *true* (the default), typing a destructive keystroke in a *dataLinked* component automatically attempts to switch the rowset into Edit mode by implicitly calling *beginEdit()*. If you set *autoEdit* to *false*, data displayed in a form is read-only, and you must explicitly call *beginEdit()* to switch to Edit mode.

autoEdit has no effect on assignments to the *value* of a field; the first assignment to a row always calls *beginEdit()* implicitly to secure a row lock.

See also *beginEdit()*, *state*

beforeGetValue

Event fired when reading a field's *value* property, which returns its apparent value.

Parameters none

Property of Field (including DbfField, PdxField, SqlField)

Description By using a field's *beforeGetValue* event, you can make its *value* property appear to be anything you want. For example, in a table you can store codes, but when looking at the data, you see descriptions. This effect is called *field morphing*. The *beforeGetValue* event is also the primary way to set up a calculated field.

A field's *beforeGetValue* event handler must return a value. That value is used as the *value* property. During the *beforeGetValue* event handler, the field's *value* property represents its true value, as stored in the row buffer, which is read from the table.

Be sure to include checks for blank values—which will occur when a *beginAppend()* starts—and the end-of-set. Any attempt to access the field values when the rowset is at the end-of-set will cause an error. Return a *null* instead.

beforeGetValue is fired when reading a field's *value* property explicitly and when read to update a *dataLinked* control. It does not fire when accessed internally for SpeedFilters, index expressions, or master-detail links, or when calling *copyToFile()*.

To reverse the process, use the field's *canChange* event.

Example In this example, a table of messages stores a message section number, but in the form, the section name is displayed in a ComboBox component. To display the section name, the section number is located in the table of section numbers that is opened in the query *sections1*. Note the tests for the end-of-set and *beginAppend()*

```
function messages1_section_beforeGetValue()
  if this.parent.parent.endOfSet
    // When navigating to end-of-set
    return null
  elseif this.value == null
    // For beginAppend()
    return ""
  else
    // Normal lookup, with value in case lookup fails
    local r
    r = this.parent.parent.parent.parent.sections1.rowset
    return iif( r.applyLocate( "Section #" = ' + this.value ),,
               r.fields[ "Name" ].value, "Closed section" )
  endif
```

In the event handler, *this* refers to the field. *this.parent.parent* refers to the rowset that contains the field (the first *parent* is the *fields* array). The form that contains the query that contains the rowset is *this.parent.parent.parent.parent*, from which you can reference the other queries on the form.

An SQL expression to perform the section number lookup is passed to *applyLocate()*. Visual dBASE automatically converts the numeric field value to a string when concatenating. If a match is found, the value of the corresponding Name field is returned; otherwise, a generic string is returned.

See also *canChange, onGotValue, value*

beginAppend()

Starts append of a new row.

Syntax *<oRef>.beginAppend()*

<oRef> The rowset you want to put in Append mode.

Property of Rowset

Description *beginAppend()* clears the row buffer and puts the rowset in Append mode, allowing the creation of a new row, via data entry through *dataLinked* controls, by directly assigning values to the *value* property of fields, or a combination of both. The row buffer is not written to disk until the rowset's *save()* method is explicitly called or there is an implicit save, which is usually caused by navigation in the rowset. At that point, a save attempt is made only if the

`beginAppend ()`

rowset's *modified* property is *true*; this is intended to prevent blank rows from being added. Calling *beginAppend()* again to add another row will also cause an implicit save first, if the row has been modified.

The integrity of the data in the row, for example making sure that all required fields are filled in, should be checked in the rowset's *canSave* event. The *abandon()* method will discard the new row, leaving no trace of the attempt.

The rowset's *canAppend* event is fired when *beginAppend()* is called. If there is a *canAppend* event handler, it must return *true* or the *beginAppend()* will not proceed.

The *onAppend* event is fired after the row buffer is cleared, allowing you to preset default values for any fields. After you preset values, set the *modified* property to *false*, so that the values in the fields immediately after the *onAppend* event are considered as the baseline for whether the row has been changed and needs to be saved.

The order of events when calling *beginAppend()* is as follows:

- 1 If the rowset has a *canAppend* event handler, it is called. If not, it's as if *canAppend* returns *true*.
- 2 If the *canAppend* event handler returns *false*, nothing else happens and *beginAppend()* returns *false*.
- 3 If the *canAppend* event handler returns *true*, the rowset's *modified* property is checked.
- 4 If *modified* is *true*:
 - 1 The rowset's *canSave* event is fired. If there is no *canSave* event, it's as if *canSave* returns *true*.
 - 2 If *canSave* returns *false*, nothing else happens and *beginAppend()* returns *false*.
 - 3 If *canSave* returns *true*, Visual dBASE tries to save the row. If the row is not saved, perhaps because it fails some database engine-level validation, a *DbException* occurs—*beginAppend()* does not return.
 - 4 If the row is saved, the *modified* property is set to *false*, and the *onSave* event is fired.
- 5 After the current row is saved (if necessary):
 - 1 The rowset is switched to Append mode.
 - 2 The *onAppend* event fires.
 - 3 *beginAppend()* returns *true*.

An exception occurs when calling *beginAppend()* if the rowset's *live* property is *false*, or if the user has insufficient rights to add rows.

Example The following event handler is used to add new rows. It carries over the values of some fields from the current row.

```
function addButton_onClick()
    local cCity, cZip, cInsp
    // Make copies of field values to carry over
    cCity = form.rowset.fields[ "City"      ].value
    cZip  = form.rowset.fields[ "Zip"       ].value
    cInsp = form.rowset.fields[ "Inspector" ].value
```

```
// Add new row
if form.rowset.beginAppend()
    form.rowset.fields[ "City"      ].value := cCity
    form.rowset.fields[ "Zip"       ].value := cZip
    form.rowset.fields[ "Inspector" ].value := cInsp
    form.rowset.modified := false    // Clear flag to set baseline for change
endif
```

The field values are copied only if *beginAppend*() succeeds. It could fail if the current row has been modified, but contains invalid data. In that case, you would not want to overwrite the current field values and clear the *modified* flag.

See also *abandon*(), *canAppend*, *canClose*, *canNavigate*, *modified*, *onAppend*, *state*, *save*()
beginAppend() is also a method of the Form class (page 15-64)

beginEdit()

Makes contents of a row editable.

Syntax <oRef>.beginEdit()
<oRef> The rowset you want to put in Edit mode.

Property of Rowset

Description By default, a rowset's *autoEdit* property is *true*, which means that data is immediately editable. The rowset implicitly calls *beginEdit*() when a destructive keystroke is typed in a *dataLinked* component. But you can more strictly control how editing occurs by setting *autoEdit* to *false* and explicitly calling *beginEdit*() as needed.

As usual, the row buffer is not written until the rowset's *save*() method is explicitly called or there is an implicit save, which is usually caused by navigation in the rowset. The integrity of the data in the row, for example making sure that there are no invalid entries in any fields, should be checked in the rowset's *canSave* event. The *abandon*() method will discard any changes to the row. After saving or abandoning any changes, the rowset goes back to Browse mode.

The rowset's *canEdit* event is fired when *beginEdit*() is called. If there is a *canEdit* event handler, it must return *true* or the *beginEdit*() will not proceed. The *onEdit* event is fired after switching to Edit mode.

The order of events when calling *beginEdit*() is as follows, even if the rowset is already in Edit mode:

- 1 If the rowset has a *canEdit* event handler, it is called. If not, it's as if *canEdit* returns *true*.
- 2 If the *canEdit* event handler returns *false*, nothing else happens and *beginEdit*() returns *false*.
- 3 If the *canEdit* event handler returns *true*:

`beginFilter ()`

- 1 The rowset attempts to switch to Edit mode by getting an automatic row lock. If the lock cannot be secured, the mode switch fails and *beginEdit()* returns *false*.
- 2 If the lock is secured, the *onEdit* event fires.
- 3 *beginEdit()* returns *true*.

An exception occurs if the rowset's *live* property is *false*, or if the user has insufficient rights to edit rows, and they call *beginEdit()*.

See also *abandon()*, *autoEdit*, *canEdit*, *canClose*, *canNavigate*, *modified*, *onEdit*, *save()*, *state*

beginFilter()

Puts a rowset in Filter mode, allowing the entry of filter criteria.

Syntax `<oRef>.beginFilter()`

<oRef> The rowset you want to put in Filter mode.

Property of Rowset

Description Rowset objects support a Filter mode in which values can be assigned to Field objects and then used to filter the rows in a rowset to show only those rows with matching values. *beginFilter()* puts the rowset in Filter mode and *applyFilter()* applies the filter values. *clearFilter()* cancels the filter. Because *dataLinked* controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a filter-by-form feature in your application.

When *beginFilter()* is called, the row buffer is cleared. Values that are set either through *dataLinked* controls or by assigning values to *value* properties are used for matching. Fields whose *value* property is left blank are not considered. To cancel Filter mode, call the *abandon()* method.

If navigation is attempted while in Filter mode, Filter mode is canceled and the navigation occurs, relative to the position of the row cursor at the time *beginFilter()* was called.

To filter rows with a condition without using Filter mode, set the rowset's *filter* property. See the *filter* property for more information on how filters are applied to data. To filter rows with dBASE code instead of or in addition to Filter mode, use the *canGetRow* event.

See also *abandon()*, *applyFilter()*, *clearFilter()*, *filter*, *filterOptions*, *state*, *value*

beginLocate()

Puts a rowset in Locate mode, allowing the entry of search criteria.

Syntax <oRef>.beginLocate()

<oRef> The rowset you want to put in Locate mode.

Property of Rowset

Description Rowset objects support a Locate mode in which values can be assigned to Field objects and then used to find rows in a rowset that contain matching values. *beginLocate()* puts the rowset in Locate mode and *applyLocate()* finds the first matching row. *locateNext()* finds other matching rows. Because *dataLinked* controls on forms write to the *value* properties of Field objects, a call to those three methods are all you need to implement a search-by-form feature in your application.

When *beginLocate()* is called, the row buffer is cleared. Values that are set either through *dataLinked* controls or by assigning values to *value* properties are used for matching. Fields whose *value* property is left blank are not considered. To cancel Locate mode, call the *abandon()* method.

If navigation is attempted while in Locate mode, Locate mode is canceled and the navigation occurs, relative to the position of the row cursor at the time *beginLocate()* was called.

See also *abandon()*, *applyLocate()*, *locateNext()*, *locateOptions*, *state*, *value*

beginTrans()

Begins transaction logging.

Syntax <oRef>.beginTrans()

<oRef> The database in which you want to start transaction logging.

Property of Database

Description Separate changes that must be applied together are considered to be a transaction. For example, transferring money from one account to another means debiting one account and crediting another. If for whatever reason one of those two changes cannot be done, the whole transaction is considered a failure and any change that was made must be undone.

Transaction logging records all the changes made to all the tables in a database. If no errors are encountered while making the individual changes in the transaction, the transaction log is cleared with the *commit()* method and the transaction is done. If an error is encountered, all changes made so far are undone by calling the *rollback()* method.

Transaction logging differs from caching updates in that changes are actually written to the disk. This means that others who are accessing the database can see your changes. In contrast, with cached updates your changes are written all

```
beginTrans ( )
```

at once later, when and if you decide to post the changes. For example, if you're reserving seats on an airplane, you want to post a reservation as soon as possible. If the customer changes their mind, you can undo the reservation with a rollback. With cached updates, the seat might be taken by someone else between the time the data entry for the reservation begins and the time it is actually posted.

All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.

For SQL-server databases, the Database object's *isolationLevel* property determines the isolation level of the transaction.

A Database object may have only one transaction active at one time; you cannot nest transactions.

Example The following demonstrates a simple batch transaction, transferring money from one account to another. The account table is indexed on the account number.

```
function transferButton_onClick
try
    form.rowset.parent.database.beginTrans()
    if form.transferAmt.value <= 0
        throw new ApplicationException( "Transfer amount must be greater than zero" )
    endif
    do ; until form.rowset.lockSet() // Wait for set lock to prevent deadlock
    if form.rowset.find( form.transferFrom.value )
        form.rowset.fields[ "Balance" ].value -= form.transferAmt.value
    else
        throw new ApplicationException( "Transfer From account not found" )
    endif
    if form.rowset.find( form.transferTo.value )
        form.rowset.fields[ "Balance" ].value += form.transferAmt.value
    else
        throw new ApplicationException( "Transfer To account not found" )
    endif
    form.rowset.save()
    form.rowset.parent.database.commit()
catch ( ApplicationException e )
    form.rowset.parent.database.rollback()
    msgbox( e.message, "Transfer failed!", 48 )
catch ( Exception e )
    form.rowset.parent.database.rollback()
    logException( e )
    #ifdef DEBUG
        throw e
    #else
        fatalError( e )
    #endif
endtry
```

As the example demonstrates, there are quite a few things that can go wrong with the transaction. These are separated into application logic errors, and all

other unexpected errors. A custom class, `AppException`, is used for the application errors. It provides an easy way to `THROW` an exception with an error message.

The debit to the first row is saved when calling `findKey()` to find the second row. The credit to the second row must be saved before committing the transaction. If an application logic exception occurs, whatever is done so far (if anything) is rolled back. Then a dialog box is displayed, detailing the error. The user can try again. For all other exceptions, a function is called to log the exception for debugging purposes. Then the exception is either `THROWn` so that the standard error dialog appears, if the application is compiled with the debug flag on; or another function is called that details the error for the user before telling them they have to terminate the application.

`logException()` and `fatalError()` are custom functions that are left as an exercise for the developer. Here is the code for the `AppException` class:

```
class AppException( cMsg, nCode ) of Exception
  if argcount() < 2
    nCode := -1
    if argcount() < 1
      cMsg := "Application logic exception"
    endif
  endif
  this.code := nCode
  this.message := cMsg
endclass
```

See also `cacheUpdates`, `commit()`, `isolationLevel`, `rollback()`

bookmark()

Returns the current position in a rowset.

Syntax `<oRef>.bookmark()`

<oRef> The rowset whose current position you want to return.

Property of Rowset

Description A bookmark represents a position in a rowset. `bookmark()` returns the current position in the rowset. The bookmark may be stored in a variable or property so that you can go back to that position later with the `goto()` method.

A bookmark is guaranteed to be valid only as long as the rowset stays open. The bookmark uses the current index represented by the `indexName` property, if any. The same physical row in the table returns different bookmarks when different indexes are in effect. When you `goto()` a bookmark, the index that was in effect when the bookmark was returned is automatically activated.

Example See the example for *applyFilter()* for an example of using *bookmark()* to store the current row in case specified filter condition finds no matches.

See also *bookmarksEqual()*, *goto()*

bookmarksEqual()

Checks if a given bookmark matches the current row, or if two bookmarks refer to the same row.

Syntax *<oRef>.bookmarksEqual(<bookmark1> [, <bookmark2>])*

<oRef> The rowset in which to check the bookmark(s).

<bookmark1> The bookmark to check against the current row in the rowset, if only one bookmark is specified; or the first of two bookmarks to compare.

<bookmark2> The second of two bookmarks to compare.

Property of Rowset

Description Use *bookmarksEqual()* to check a bookmark against the current row, without having to first use *bookmark()* to get a bookmark for the current row. If the bookmark refers to the current row, *bookmarksEqual()* returns *true*; if not it returns *false*. You may also use *bookmarksEqual()* to compare two bookmarks to see if they refer to the same row; the equality operators (= and ==) may also be used to compare two bookmarks.

The bookmark uses the current index represented by the *indexName* property, if any. The same physical row in the table returns different bookmarks when different indexes are in effect. When checking a bookmark against the current row, the rowset must be in the same index order as the bookmark; otherwise *bookmarksEqual()* will return *false*. When comparing two bookmarks, they must have been taken when the same index was in effect; if not, they will not match.

See also *bookmark()*

cacheUpdates

Whether to cache updates locally instead of writing to disk as they occur.

Property of Database

Description Normally, when a row buffer is saved, it is written to disk. By setting the *cacheUpdates* property to *true*, those changes are cached locally instead of being written to disk. One reason to do this is to reduce network traffic. Changes are accumulated and then posted with the *applyUpdates()* method, after a certain amount of time or a certain number of changes have been made.

Another reason is to simulate a transaction when you have more than one change in an all-or-nothing situation. For example, if you need to fill a customer order and reduce the stock in inventory, you cannot let one happen and not the other. When the changes are posted with *applyUpdates()*, they are applied inside a transaction at the database level. Because you cannot nest transactions, you cannot have a transaction with *beginTrans()* and use cached updates at the same time. If any of the changes do not post, for example one of the records is locked, all of the changes that did post are undone and *applyUpdates()* returns *false* to indicate failure. The cached updates remain cached so that you can retry the posting. If all the changes are posted successfully, *applyUpdates()* returns *true*.

Finally, because of the all-or-nothing nature of cached updates, you can use them to allow the user to tentatively make changes that you can simply discard as a group. For example, you could allow a user to modify a lookup table. If the user submits the changes they are applied, but if the user chooses to cancel, any changes made can be discarded by calling the *abandonUpdates()* method. Note that with cached updates, the changes aren't actually written until posted. In contrast, transaction logging actually makes the changes as they happen, but allows you to undo them if desired.

See also *abandonUpdates()*, *applyUpdates()*, *beginTrans()*, *commit()*, *rollback()*

canAbandon

Event fired when attempt to abandon rowset occurs; return value determines if changes to row are abandoned.

Parameters none

Property of Rowset

Description A rowset may be abandoned explicitly by calling its *abandon()* method, or implicitly via the user interface by pressing *Esc* or choosing Abandon Row from the default Table menu or toolbar while editing table rows. *canAbandon* may be used to verify that the user wants to abandon any changes that they have made. You may check the *modified* property first to see if there are any changes to abandon; if not, there is no need to ask.

The *canAbandon* event handler must return *true* or *false* to indicate whether the changes to the rowset, if any, are abandoned.

Example The following method handles these conditions:

- Abandoning changes to an existing row
- Abandoning a new row
- Choosing to abandon an existing row when there are no changes
- Choosing to abandon a new row when there are no changes
- Abandoning other rowset modes

It uses manifest constants created with the `#define` preprocessor directive (and available in the `VDBASE.H` include file) to represent the options of the *state* property, which makes the code more readable, and macro-functions to display a simple alert dialog box to display a yes/no dialog box and return *true* or *false*.

```
#define STATE_CLOSED    0
#define STATE_BROWSE    1
#define STATE_EDIT      2
#define STATE_APPEND    3
#define STATE_FILTER    4
#define STATE_LOCATE    5
#define alert(m)        (msgbox(m,"Alert",64))
#define confirm(m)      (msgbox(m,"Confirm",4+32)==6)

:

function Rowset_canAbandon
    if this.state == STATE_EDIT
        if this.modified
            return confirm( "Abandon changes?" )
        else
            alert( "No changes made; nothing to abandon" )
            return false    // Do not fire onAbandon
        endif
    elseif this.state == STATE_APPEND
        if this.modified
            return confirm( "Abandon new entry?" )
        else
            return true      // Discard new blank row
        endif
    else
        return true          // OK to abandon Filter and Locate modes
    endif
endfunction
```

See also *abandon()*, *modified*, *onAbandon*

canAppend

Event fired when attempting to put rowset in Append mode; return value determines if the mode switch occurs.

Parameters none

Property of Rowset

Description A rowset may be put in Append mode explicitly by calling its *beginAppend()* method, or implicitly via the user interface by choosing Append Row from the default Table menu or toolbar while editing table rows. *canAppend* may be used to verify that the user wants to add a new row. You can check the *modified* property first to see if the user has made any changes to the current row; if not, you may not want to ask.

The *canAppend* event handler must return *true* or *false* to indicate whether *beginAppend()* proceeds. For information on how *canAppend* interacts with other events and implicit saves, see *beginAppend()*.

See also *beginAppend()*, *canSave*, *modified*, *onAppend*

canChange

Event fired when a change to the *value* property of a Field object is attempted; return value determines if the change occurs.

Parameters **<new value>** The proposed new value.

Property of Field

Description Use *canChange* to determine whether changes to individual fields occur. *canChange* fires when something is assigned to the *value* property of a Field object, either directly or through a *dataLinked* control. The proposed new value is passed as a parameter to the *canChange* event handler. If the *canChange* event handler returns *false*, the Field object's *value* remains unchanged.

While *canChange* provides field-level validation to see whether changes are saved into the row buffer, use *canSave* to provide row-level validation to determine whether the buffer can be saved to disk. You should always do row-level validation no matter whether you do field-level validation or not.

The *canChange* event operates separately from database engine-level validation. Even if *canChange* returns *true*, attempting to write an invalid value to a field, for example exceeding a field's maximum allowed value, will fail and the field's *value* property will remain unchanged.

You can also use *canChange* to reverse the field morphing performed by *beforeGetValue*. Inside the *canChange* event handler, examine the *<new value>* parameter and assign the value you want to store in the table directly to the *value* property of the Field object. Doing so does not fire *canChange* recursively. Then have the *canChange* event handler return *false* so that the *<new value>* does not get saved into the row buffer.

Example In this example, a table of messages stores a message section number, but in the form, the section name is displayed in a ComboBox component. When a section is chosen by name, the section number is stored in the table instead with the following *canChange* event handler. The table of section numbers is opened in the query *sections1*.

```
function messages1_section_canChange( newValue )
  local r
  r = this.parent.parent.parent.parent.sections1.rowset; // Lookup table
  if r.applyLocate( ["Name" = ' ] + newValue +[' ] )      // If name found
    this.value = r.fields[ "Section #" ].value           // save section #
  endif
```

```
return false    // Always return false so that newValue is not saved
```

In the event handler, *this* refers to the field. *this.parent.parent* refers to the rowset that contains the field (the first *parent* is the *fields* array). The form that contains the query that contains the rowset is *this.parent.parent.parent.parent*, from which you can reference the other queries on the form.

An SQL expression to perform the section name lookup is passed to *applyLocate()*. If a match is found, the value of the corresponding section number field is stored in the *value* property of the field. Then the event handler returns *false*. If no match is found, the field is not changed.

See also *beforeGetValue*, *canSave*, *onChange*, *onGotValue*, *value*
canChange is also an event of the *TreeView* class (page 15-67).

canClose

Event fired when there's an attempt to deactivate a query or stored procedure; return value determines if the object is deactivated.

Parameters none

Property of Query, StoredProc

Description If the *active* property of a Query or StoredProc object is set to *false*, that object's *canClose* event fires. If the *canClose* event handler returns *false*, the close attempt fails and the *active* property remains *true*.

A StoredProc object may be deactivated only if it returns a rowset. If it returns values only, the *active* property is automatically reset to *false* after the stored procedure is called; there is nothing to deactivate.

Normally when a Query or StoredProc object closes, it saves any changes in its rowset's row buffer, if any. In attempting to save those changes, the rowset's *canSave* event is also fired, before *canClose*. If *canSave* returns *false*, the row is not saved, and the object is not closed.

If you want to abandon uncommitted changes instead of saving them when closing the object, call the rowset's *abandon()* method before closing.

See also *abandon()*, *active*, *canSave*, *onClose*
canClose is also an event of the *Form* class (page 15-68).

canDelete

Event fired when attempting to delete the current row; return value determines if the row is deleted.

| | |
|--------------------|--|
| Parameters | none |
| Property of | Rowset |
| Description | <p>A row may be deleted explicitly by calling the <i>delete()</i> method, or implicitly via the user interface by choosing Delete Rows from the default Table menu or toolbar. <i>canDelete</i> may be used to make sure that the user wants to delete the current row.</p> <p><i>canDelete</i> may also be used to do something with the current row, just before you delete it. In this case, the <i>canDelete</i> event handler would always return <i>true</i>.</p> <p>The <i>canDelete</i> event handler must return <i>true</i> or <i>false</i> to indicate whether the row is deleted. For information on how <i>canDelete</i> interacts with other events, see <i>delete()</i></p> |

Example The following event handler copies the row that is about to be deleted to a separate archive table that is opened in another query in the form.

```
function Rowset_canDelete
    local n, rArchive
    rArchive = this.parent.parent.archive1.rowset
    rArchive.beginAppend()
    for n = 1 to this.fields.size
        rArchive.fields[ n ].value = this.fields[ n ].value
    endfor
    rArchive.save()
    return true
```

This *canDelete* event handler always returns *true* after making the copy so that the row is deleted.

See also *delete()*, *onDelete*

canEdit

Event fired when attempting to put rowset in Edit mode; return value determines if the mode switch occurs.

| | |
|--------------------|--|
| Parameters | none |
| Property of | Rowset |
| Description | <p>The <i>beginEdit()</i> method is called (implicitly or explicitly) to put the rowset in Edit mode. <i>canEdit</i> may be used to verify that the user is allowed to or wants to edit the row.</p> <p>The <i>canEdit</i> event handler must return <i>true</i> or <i>false</i> to indicate whether the switch to Edit mode proceeds.</p> |

See also *beginEdit()*, *onEdit*

canGetRow

Event fired when attempting to read a row into the row buffer; return value determines if the row stays in or is filtered out.

Parameters none

Property of Rowset

Description In addition to setting an SQL filter expression in the *filter* property, you can filter out rows through dBASE code with *canGetRow*. In a *canGetRow* handler, the rowset acts as if the row is read into the row buffer. You can test the *value* properties of the field objects, or anything else.

If *canGetRow* returns *true*, that row is kept. If it returns *false*, the row is discarded and the next row is tried.

Note that *canGetRow* fires before applying the constrain on a detail table linked through *masterRowset* or *masterSource*. Therefore, when using this type of link, you cannot check for the existence of detail rows (by checking the detail rowset's *endOfSet* property) or get the values of the first matching detail row in the *canGetRow* event handler. To access the matching rows in the linked table during the *canGetRow* event, you must manually apply the constrain (using the *setRange()* or *query()* methods) inside the *canGetRow* instead of using the built-in properties. Then you are free to access the detail table as usual.

Example Suppose a message database supports private messages that can be seen only by the sender and the recipient. You can prevent others from seeing private messages with a *canGetRow* event handler. The name of the user is stored as a property of the form. That name must match either the From or To fields in the message.

```
function messages1_canGetRow()
    return this.fields[ "From" ].value == this.parent.parent.userName or
           this.fields[ "To"   ].value == this.parent.parent.userName
```

See also *count()*, *filter*

canNavigate

Event fired when attempting navigation in a rowset; return value determines if row cursor is moved.

Parameters none

Property of Rowset

Description Navigation in a rowset may occur explicitly by calling a navigation method like *next()* or *goto()*, or implicitly via the user interface by choosing a navigation

option from the default Table menu or toolbar while viewing a rowset. *canNavigate* may be used to verify that the user wants to leave the current row to go to another. You may check the *modified* property first to see if the user has made any changes to the current row; if not, you may not want to ask.

canNavigate may also be used to do something with the current row, just before you leave it. In this case, the *canNavigate* event handler would always return *true*.

The *canNavigate* event handler must return *true* or *false* to indicate whether the navigation occurs. For information on how *canNavigate* interacts with other events and implicit saves, see *next()*.

See also *canSave*, *first()*, *goto()*, *last()*, *modified*, *next()*, *onNavigate*
canNavigate is also an event of the Form class (page 15-71).

canOpen

Event fired when attempting to open a query or stored procedure; return value determines if object is opened.

Parameters none

Property of Query, StoredProc

Description *canOpen* fires when a Query or StoredProc object's *active* property is set to *true*. If an event handler is assigned to the *canOpen* property, the event handler must return *true* or *false* to indicate whether the object is opened and activated. *canOpen* may also be used to do something with the query, just before you open it. In this case, the *canOpen* event handler would always return *true*.

See also *active*, *onOpen*

canSave

Event fired when attempting to save the row buffer; return value determines if the buffer is written.

Parameters none

Property of Rowset

Description The row buffer may be saved explicitly by calling *save()* or implicitly, usually by navigating in the rowset. Use *canSave* to verify that the data is good before attempting to write it to the disk.

The *canSave* event handler must return *true* or *false* to indicate whether the row is saved. If the user has changed the current row and attempts to append a new row or navigate, *canAppend* or *canNavigate* fires first. If that event returns *true*, then the *canSave* event fires. If *canSave* returns *false*, the row is not saved, and the attempted action does not occur. If *canSave* returns *true*, then the row is saved and the action occurs. This allows you to put row validation code in the *canSave* event handler that you do not need to duplicate in either *canAppend* or *canNavigate*.

The *canSave* event operates separately from database engine-level validation. Even if *canSave* returns *true*, attempting to write an invalid row, for example one that fails to pass a table constraint, will fail and cause an exception.

Example The following event handler verifies that required fields are filled in, and displays a dialog box detailing any missing data.

```
function Rowset_canSave()
    local cErrors
    cErrors = "" // String for errors
    if empty( this.fields[ "Last name" ].value )
        cErrors += "- LAST NAME cannot be blank" + chr(13)
    endif
    if empty( this.fields[ "ZIP" ].value )
        cErrors += "- ZIP CODE cannot be blank" + chr(13)
    endif
    if "" # cErrors
        msgbox( "Can't save current entry because:" + chr(13) + cErrors, "Bad entry", 48 )
        return false
    else
        return true
    endif
end
```

See also *canAppend*, *canNavigate*, *onSave*, *save*()

changedTableName

Name of the table for which you want to collect copies of original values of rows that were changed.

Property of UpdateSet

Description When doing an *update*() or *appendUpdate*(), rows will be changed. The original contents of the rows that are changed are copied to the table specified by the *changedTableName* property. If the table does not exist, it is created. If it does exist, it is erased first so that it contains only those rows that were changed on the last update.

By making copies of the original values of the rows that are changed, you can undo the changes by doing another *update*(), using the *changedTableName* table as the *source* table.

See also *appendUpdate(), keyViolationTableName, problemTableName, source, update()*

clearFilter()

Clears any active filter on a rowset.

Syntax *<oRef>.clearFilter()*
<oRef> The rowset whose filter to clear.

Property of Rowset

Description *clearFilter()* clears the *filter* property and any filter set through the rowset's Filter mode, thereby deactivating any filters. Rows that were hidden by the filter become visible. The row cursor is not moved.

See also *applyFilter(), beginFilter(), filter*

clearRange()

Clears any active range on a rowset.

Syntax *<oRef>.clearRange()*
<oRef> The rowset whose range to clear.

Property of Rowset

Description *clearFilter()* clears the range set by the *setRange()* method. The row cursor is not moved.

See also *setRange()*

close()

Closes a database connection.

Syntax This method is called implicitly by the Database object.

Property of Database

Description The *close()* method closes the database connection. It is called implicitly when you set the Database object's *active* property to *false*. In typical usage, you do not call this method directly.

Advanced applications may override the definition of this method to perform supplementary actions when closing the database connection. Custom data drivers must define this method to perform the appropriate actions to close their database connection.

See also *active, open()*

close() is also a method of the File (page 11-7) and Form (page 15-74) classes.

commit()

Clears the transaction log, committing all logged changes

Syntax <oRef>.commit()

<oRef> The database whose changes you want to commit.

Property of Database

Description A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling the *rollback()* method. Otherwise, *commit()* is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

Example See *beginTransaction()*.

See also *beginTransaction()*, *cacheUpdates*, *rollback()*

constrained

Specifies whether updates to a rowset will be constrained by the WHERE clause of the query's SQL SELECT command. Applies to Standard tables only.

Property of Query

Description When *constrained* is set to *true*, any time a row is saved, if the query's SQL SELECT statement—which was stored in the *sql* property and used to generate the rowset—contains a WHERE clause, the newly saved row is evaluated against the WHERE clause. If the row no longer matches the condition set by the WHERE clause, the row is considered to be out-of-set, and the row cursor moves to the next row in the set, or to the end-of-set if already at the last row.

This property applies only to Standard tables and defaults to *false*, which means that the SQL SELECT statement is used only to generate the rowset, not to actively constrain it. By setting the *constrained* property to *true*, Standard tables behave more like SQL-server based tables, which always constrain rows according to the WHERE clause.

See also *sql*

copy()

Copies one rowset or table to another rowset or table.

Syntax `<oRef>.copy()`

<oRef> The UpdateSet object that describes the copy.

Property of UpdateSet

Description The Database's *copyTable()* method is used to copy all of the rows from a single table in a database to another new table in the same database. The *copy()* method can be used for any other type of row copy: from one rowset to another in the same database, from one rowset to another in a different database, from one rowset to a table, or from one table to a rowset.

The *source* and *destination* properties specify what to copy and where to copy it. Because you can use a rowset as a *source*, you can copy only part of a table, by selecting only those rows you want to copy for the rowset. When using a table name as a *destination*, that table is created, or overwritten if it already exists. To convert from one table type to another, create a rowset of the desired result type and assign it to the *destination* property.

See also *copyTable()*, *destination*, *source*
copy() is also a method of the File class (page 11-8).

copyTable()

Makes a copy of one table to create another table in the same database.

Syntax `<oRef>.copyTable(<source table expC>, <destination table expC>)`

<oRef> The database in which you want to copy the table.

<source table expC> The name of the table you want to duplicate.

<destination table expC> The name of the table you want to create.

Property of Database

Description *copyTable()* copies all of the rows from a single table in a database to another new table in the same database. The resulting destination table will be the same table type as the source table. Use the UpdateSet's *copy()* method for any other type of row copy.

The table to copy should not be open.

`copyToFile ()`

To make a copy of a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].copyTable( "Stuff", "CopyOfStuff" )
```

See also `copy()`

copyToFile()

Copies the contents of a BLOB field to a new file.

Syntax `<oRef>.copyToFile(<file name expC>)`

<oRef> The BLOB field to copy.

<file name expC> The name of the file you want to create.

Property of Field

Description `copyToFile()` copies the specified BLOB field (including memo fields) to the named file.

Example The following event handler copies the contents of a memo field named Notes in the current row to a text file.

```
function exportMemoButton_onClick
    local cFile
    cFile = putfile( "Export memo", "*.txt" )
    if "" # cFile
        form.rowset.fields[ "Notes" ].copyToFile( cFile )
    endif
```

See also `replaceFromFile()`

count()

Returns the number of rows in a rowset, respecting any filter conditions and events.

Syntax `<oRef>.count()`

<oRef> The rowset you want to measure.

Property of Rowset

Description `count()` returns the number of rows in the current rowset. For a rowset generated by a simple query like the following, which selects all the fields from

a single table with no conditions, *count()* returns the number of rows in the table:

```
select * from CUSTOMER
```

You can use *count()* while a filter is active—with the *filter* property or the *canGetRow* event—to count the number of rows that match the filter condition. This may be time-consuming with large rowsets.

See also *canGetRow, filter*

count() is also a method of the *AssocArray* class (page 10-9).

database

The Database object to which the query or stored procedure is assigned.

Property of Query, StoredProc

Description A query or stored procedure must be assigned to the database that provides access to the tables it wants before it is activated. When created, a Query or StoredProc object is assigned to the default database in the default session.

To assign the object to the default database in another session, assign that session to the *session* property. Assigning the *session* property always sets the *database* property to the default database in that session.

To assign the object to another database in another session, assign the object to that session first. This makes the databases in that session available to the object.

See also class Database

databaseName

The BDE alias that the object represents.

Property of Database

Description To use a BDE alias, create a Database object and assign the alias to the object's *databaseName* property. Then set the *active* property to *true* to activate the database. While the database is active, you cannot change the *databaseName* property.

The *databaseName* property for a session's default database is always blank.

See also *active*

dataModClass

The class name of the desired data module.

Property of DataModRef

Description After setting the *filename* property to the file that contains the data module class definition, set the *dataModClass* property to the name of the desired class.

Note When declaring a class name, the name may exceed 32 characters, but the rest are ignored. When attempting to use a class, the name should not exceed 32 characters; otherwise the named class may not be found.

See also *active, filename*

decimalLength

The number of decimal places in a DBF (dBASE) numeric or float field.

Property of DbfField

Description The DBF (dBASE) table format supports two kinds of fields that store numbers: numeric and float. Both field types have a fixed number of decimal places. The *decimalLength* property represents the number of decimal places for any Field objects that represent a numeric or float field. For other field types, *decimalLength* is zero.

See also *readOnly*

default

The default value for a field.

Property of DbfField, PdxField

Description *default* indicates the default value of the field represented by the field object. When a rowset switches to Append mode to add a new row, the field objects take on their default values.

For date fields, the special value TODAY indicates today's date. For timestamp fields, the special value NOW indicates the current date and time.

See also *required*

delete() [Rowset]

Deletes the current row.

Syntax <oRef>.delete()

<oRef> The rowset whose current row you want to delete.

Property of Rowset

Description *delete()* deletes the current row in the rowset. When *delete()* is called, the *canDelete* event is fired. If there is no *canDelete* event handler or the event handler returns *true*, the current row is deleted, the *onDelete* event fires, and the row cursor moves to the next row, or to the end-of-set if the last row was the one that was deleted. This movement is not considered navigation, so the rowset's *onNavigate* does not fire. If you have an *onNavigate* event handler, call it from the *onDelete* event.

While the DBF (dBASE) table format supports soft deletes, in which the rows are only marked as deleted and not actually removed until the table is packed, there is no method in the data access classes to recall those records. Therefore a *delete()* should always be considered final.

See also *canDelete*, *onDelete*, *packTable()*

delete() is also a method of the Array (page 10-10), File (page 11-13), and UpdateSet (page 14-65) classes.

delete() [UpdateSet]

Deletes the rows in the destination that are listed in the source.

Syntax <oRef>.delete()

<oRef> The UpdateSet object that describes the delete.

Property of UpdateSet

Description *delete()* deletes the rows listed in the *source* rowset or table from the *destination* rowset or table. The *destination* must be indexed.

See also *destination*, *source*

delete() is also a method of the Array (page 10-10), File (page 11-13), and Rowset (page 14-65) classes.

destination

The target rowset or table of an UpdateSet operation.

Property of UpdateSet

Description The *destination* property contains an object reference to a rowset or the name of a table that is the target of an UpdateSet operation. For an *append()*, *update()*, or *appendUpdate()*, it refers to the rowset or table that is changed. For a *copy()*, it refers to the rowset or table that receives the copies. If a table name is specified, that table is created, or overwritten if it already exists. For a *delete()*, the *destination* property refers to the table from which rows are deleted.

The *source* property specifies the other end of the UpdateSet operation.

See also *append()*, *appendUpdate()*, *copy()*, *delete()*, *source*, *update()*

driverName

The database driver used for the database connection.

Property of Database

Description The *driverName* property reflects the database driver used for the connection. It's determined by the database driver for the database's BDE alias and set automatically once the database is successfully made active.

For default databases, the *driverName* matches the System setting in the BDE Administrator.

See also *databaseName*

dropTable()

Deletes (drops) a table from a database.

Syntax <oRef>.dropTable(<table name expC>)

<oRef> The database in which the table exists.

<table name expC> The name of the table you want to delete.

Property of Database

Description *dropTable()* deletes a table and any existing secondary files, like memo files and indexes. *dropTable()* does not ask for confirmation; the deletion is immediate.

The table cannot be open anywhere at the time of the *dropTable()*; if it is, *dropTable()* fails.

To delete a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].dropTable( "Temp" )
```

See also *emptyTable()*

emptyTable()

Deletes all the rows in a table.

Syntax *<oRef>.emptyTable(<table name expC>)*
<oRef> The database in which the table exists.
<table name expC> The name of the table you want to empty.

Property of Database

Description *emptyTable()* deletes all of the rows in a table, leaving an empty table structure, as if the table was just created. *emptyTable()* does not ask for confirmation; the deletion is immediate. The table cannot be open anywhere at the time of the *emptyTable()*; if it is, *emptyTable()* fails.

To empty a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].emptyTable( "YtdSales" )
```

See also *dropTable()*

endOfSet

Specifies whether the row cursor is at the end-of-set.

Property of Rowset

Description The row cursor is always positioned at either a valid row or the end-of-set. There are two end-of-set positions: one before the first row and one after the last row. *endOfSet* is *true* if the row cursor is positioned at either end-of-set position. When you first make a query active successfully, *endOfSet* is *true* if there are no rows that match the specified criteria in the query's SQL SELECT statement, or simply no rows in the tables selected.

`execute()`

When you apply a filter by calling *applyFilter()* or setting the *filter* property, *endOfSet* becomes *true* if there are no rows that match the filter criteria. Otherwise, the row cursor is positioned at the first matching row.

If you navigate backward before the first row in the set or after the last row in the set, this moves the row cursor to the end-of-set, so *endOfSet* becomes *true*. You can call the *first()* or *last()* methods to attempt to move the row cursor to the first or last row in the set. If after calling one of those methods, *endOfSet* is still *true*, then there are no visible rows in the current set.

Attempting to read or change a field value while at end-of-set causes an error.

See also *applyFilter(), filter, first(), last()*

execute()

Executes a query or stored procedure.

Syntax This method is called implicitly by the Query or StoredProc object.

Property of Query, StoredProc

Description The *execute()* method executes a query or stored procedure. It is called implicitly after *prepare()* when you set the object's *active* property to *true*. In typical usage, you do not call this method directly.

Advanced applications may override the definition of this method to perform supplementary actions when executing the query or stored procedure. Custom data drivers must define this method to perform the appropriate actions to retrieve a rowset or call a stored procedure.

See also *active, prepare()*

executeSQL()

Executes the specified SQL statement.

Syntax `<oRef>.executeSQL(<SQL expC>)`
<oRef> The database in which you want to execute the SQL statement.
<SQL expC> The SQL statement.

Property of Database

Description Use *executeSQL()* to perform an SQL operation that does not have a data access object equivalent, for example, to use data definition language (DDL) SQL where no rowset is desired, and for server-specific SQL.

fieldName

The name of the field represented by the Field object.

Property of Field (including DbfField, PdxField, SqlField)

Description The *fieldName* property contains the name of the field that the Field object represents. The *fieldName* property is automatically filled in when the rowset object is generated.

For a calculated field, the *fieldName* contains the name of the field assigned when the Field object is created.

See also *name*

fields

An array that contains the Field objects in a rowset.

Property of Rowset

Description A rowset's *fields* property contains an object reference to the array of field objects in the rowset. These fields can be accessed by their field name or their ordinal position; for example, if *this* refers to a rowset:

```
this.fields[ "State" ].value = "CA"    // Assign "CA" to State field
this.fields[ 1 ].value = 12           // Assign 12 to first field
```

To access the value of the field, you must reference the field's *value* property. You can use the *add()* method to add new Field objects to the *fields* array as calculated fields.

See also *add()*, *value*

fields is also a property of the Browse class (page 15-96)

filename

The name of the file that contains the desired data module.

Property of DataModRef

Description After setting the *filename* property to the file that contains the data module class definition, set the *dataModClass* property to the name of the desired class. Data modules are stored in files with a .DMD extension.

See also *active*, *dataModClass*

filename is also a property of the ReportViewer class (page 15-97)

filter

An SQL expression that filters out rows that do not match specified criteria.

Property of Rowset

Description A filter is a mechanism by which you can temporarily hide, or filter out, those rows that do not match certain criteria so that you can see only those rows that do match. The criteria is in the form of a character string that contains an SQL expression, like the one used in the WHERE clause of an SQL SELECT. Simple comparisons using the basic SQL comparison operators (`=`, `<>`, `<`, `>`, `<=`, `>=`) are supported; other predicates like NULL and LIKE are not. Multiple comparisons may be joined by AND or OR. For example,

```
"Firstname = 'Waldo'"
```

In this case, you would see only those rows in the current rowset whose Firstname field was “Waldo”. You can use the rowset’s Filter mode, initiated by calling the *beginFilter*() method, to build the expression automatically, and then apply it with the *applyFilter*() method. The alternative is to assign the character string directly to the *filter* property.

If the filter expression contains a quoted string that contains an apostrophe, precede the apostrophe with a backslash. Note that the single quote used in SQL expressions for strings and the apostrophe are represented by the same single quote character on the keyboard. For example, if *this* is the rowset and you want to display rows with the Lastname “O’Dell”:

```
this.filter := "Lastname = 'O\'Dell'"
```

Setting the *filter* property causes the row cursor to move to the first matching row. If no rows match the filter expression, the row cursor is moved to the end-of-set; the *endOfSet* property is set to *true*.

While a filter is active, the row cursor will always be at either a matching row or the end-of-set. Any time you attempt to navigate to a row, the row is evaluated to see if it matches the filter condition. If it does, then the row cursor is allowed to position itself at that row and the row can be seen. If the row does not match the filter condition, the row cursor continues in the direction it was moving to find the next matching row. It will continue to move in that direction until it finds a match or gets to the end-of-set. For example, suppose that *this* is the rowset, and you execute the following to your program. If no filter is active, you would move four rows forward, toward the last row:

```
this.next( 4 )
```

If a filter is active, the row cursor will move forward until it has encountered four rows that match the filter condition, and stop at the fourth. That may be the next four rows in the rowset, if they all happen to match, or the next five, or the

next 400, or never, if there aren't four rows after the current row that match. In that last case, the row cursor will be at the end-of-set.

In other words, when there is no filter active, every row is considered a match. By setting a filter, you filter out all the rows that don't match certain criteria.

To clear a filter, you can assign an empty string to the *filter* property, or call the *clearFilter()* method.

In addition to using an SQL expression, you can filter out rows with more complex code by using the *canGetRow* event.

See also *applyFilter()*, *beginFilter()*, *canGetRow*, *clearFilter()*, *endOfSet*, *filterOptions*, *setRange()*

filterOptions

Determines how values are matched for filtering.

Property of Rowset

Description The *filterOptions* property is an enumerated property that controls how the *value* properties in the field objects entered during Filter mode are matched against the values in the table. These are the options:

| Value | Effect |
|-------|--------------------------------------|
| 0 | Match length and case |
| 1 | Match partial length |
| 2 | Ignore case |
| 3 | Match partial length and ignore case |

When matching partial length, the entire search value must match all or part of the value in the table, starting at the beginning of the field. For example, searching for "Central Park", will match "Central Park West", but "West" alone would not.

filterOptions also determines how fields are matched when specifying an SQL expression in the *filter* property.

The *filterOptions* property takes effect when you assign the SQL expression to the *filter* property or call *applyFilter()*. Changing *filterOptions* after activating the filter has no effect (until you change the filter).

See also *applyFilter()*, *filter*

findKey()

Finds the row with the exact matching key value.

`findKeyNearest()`

Syntax `<oRef>.findKey(<exp>)`

<oRef> The rowset in which to do the search.

<exp> The value to search for.

Property of Rowset

Description *findKey()* performs an indexed search in the rowset, using the index specified by the rowset's *indexName* property. It looks for the first row in the index whose index key value matches *<exp>*, returning *true* or *false* to indicate whether a match is found.

findKey() is a navigation method; calling it fires the *canNavigate* event. If it returns *false*, no search is attempted. If the search fails, the row cursor goes to the current row. *onNavigate* always fires after a search attempt. For more information on how navigation methods interact with navigation events and implicit saves, see *next()*.

findKey() always performs a partial key match with strings. For example, *findKey("S")* will find "Sanders", or whatever is the first key value that starts with the letter "S". To perform a full key match, pad *<exp>* with extra spaces, enough spaces to match the length of the index key value.

See also *applyLocate()*, *findKeyNearest()*

findKeyNearest()

Finds the row with the nearest matching key value.

Syntax `<oRef>.findKeyNearest(<exp>)`

<oRef> The rowset in which to do the search.

<exp> The value to search for.

Property of Rowset

Description *findKey()* performs an indexed search in the rowset, using the index specified by the rowset's *indexName* property. It looks for the first row in the index whose index key value matches *<exp>*, returning *true* or *false* to indicate if an exact match is found. If an exact match is not found, the row cursor is left at the nearest match; the row where the match would have been. For example, if "Smith" is followed by "Smythe" in the index, and the search expression is "Smothers", the search will fail and the row cursor will be left at "Smythe". "Smothers" comes after "Smith" and before "Smythe", so if it was in the index, it would be where "Smythe" is.

You can think of this exact or nearest matching as "equal or the one after," as long as you remember that "after" depends on the index order. If the index is descending instead of ascending, then in the previous example, "Smythe"

would be followed by “Smith”, and a search for “Smothers” would end up on “Smith”. The row cursor will end up on the end-of-set if the search value comes after the last value in the index.

findKeyNearest() is a navigation method; calling it fires the *canNavigate* event. If it returns *false*, no search is attempted. *onNavigate* always fires after a search attempt. For more information on how navigation methods interact with navigation events and implicit saves, see *next()*.

findKeyNearest() always performs a partial key match with strings. For example, *findKeyNearest* (“Smi”) will find “Smith”. To perform a full key match, pad *<exp>* with extra spaces, enough spaces to match the length of the index key value.

See also *applyLocate()*, *findKey()*

first()

Moves the row cursor to the first row in the rowset.

Syntax *<oRef>.first()*

<oRef> The rowset in which you want to move the row cursor.

Property of Rowset

Description Call *first()* to move the row cursor to the first row in the rowset. If a filter is active, it moves the row cursor to the first row in the rowset that matches the filter criteria.

As a navigation method, *first()* interacts with *canNavigate*, *onNavigate*, and implicit saves. For more information, see *next()*.

If the *endOfSet* property is *true* after a call to *first()*, then there are no rows that match the filter criteria if there is a filter set. If there is no filter, then that means there are no rows at all in that rowset.

See also *endOfSet*, *filter*, *last()*, *next()*

flush()

Commits data buffers to disk.

Syntax *<oRef>.flush()*

<oRef> The rowset you want to write to disk.

Property of Rowset

Description When a row is saved, the changes are written to the rowset data buffer in memory. This buffer is written to disk only as needed; for example, before another block of rows are read into the buffer. This eliminates redundant disk writes that would slow your application.

flush() explicitly writes the rowset's data buffers to disk. Note that if a disk cache is active, the buffer is written to the disk cache; the cache decides when to actually write the data onto the physical disk.

refresh() is similar to *flush()* because in purging cached rows, *refresh()* writes any rows that have been changed but not yet committed to disk. *flush()* writes the rows, but does not purge the data buffer; the rows are still cached.

Example The following *onSave* event handler calls the rowset's *flush()* method to make sure that the data is written to disk as each record is saved:

```
function Rowset_onSave
    this.flush()
```

See also *refresh()*, *save()*

flush() is also a method of the File class (page 11-20)

getSchema()

Returns information about a database.

Syntax `<oRef>.getSchema(<item expC>)`

<oRef> The database you want to get information about.

<item expC> The information to retrieve, which may be one of the following strings (which are not case-sensitive):

| String | Information |
|------------|---|
| DATABASES | A list of all databases aliases |
| PROCEDURES | A list of stored procedures defined in the database |
| TABLES | A list of all tables in the database |
| VIEWS | A list of all views in the database |

Property of Database

Description Use *getSchema()* to get a list of all database aliases, or to get information about a specific database. Some databases may not support PROCEDURES or VIEWS. All lists are returned in an Array object; if the item is not supported, the array is empty.

Custom data drivers must define this method to return the appropriate information for their database.

Example The following class subclasses the Database class to provide support for RapidFile tables. The code shown here implements the *getSchema()* method:

```
class RapidFileDatabase of Database
  this.path = ""

  function getSchema( cArg )
    local cItem
    cItem = upper( cArg )
    do case
      case cItem == "DATABASES"
        return super::getSchema( "DATABASES" )
      case cItem == "TABLES"
        local aRet, nFiles
        aRet = new Array()
        nFiles = aRet.dir( this.path + "*.RPD" ) // Get all RapidFile files
        if nFiles > 0
          aRet.resize( nFiles, 1, 1 ) // Filenames only in 2-D array
          aRet.resize( nFiles, 0 ) // Convert to 1-D array
        endif
        return aRet
      case cItem == "PROCEDURES" or cItem == "VIEWS"
        return new Array()
      otherwise
        return super::getSchema( cItem )
    endcase
  endfunction
endclass
```

goto()

Moves the row cursor to a specific row in the rowset.

Syntax <oRef>.goto(<bookmark>)

<oRef> The rowset in which you want to move the row cursor.

<bookmark> The bookmark you want to move to.

Property of Rowset

Description Call *goto()* to move the row cursor to a specific row in the rowset. Store the current row position in a bookmark with the *bookmark()* method. Then you can return to that row later by calling *goto()* with that bookmark as long as the rowset has remained open. If the rowset has been closed, the bookmark is not guaranteed to return you to the correct row, since the table may have changed.

The bookmark uses the current index represented by the *indexName* property, if any. The same physical row in the table returns different bookmarks when different indexes are in effect. When you *goto()* a bookmark, the index that was in effect when the bookmark was returned is automatically activated.

If you attempt to `goto()` a row that is out-of-set, you will generate an error.

As a navigation method, `goto()` interacts with `canNavigate`, `onNavigate`, and implicit saves. For more information, see `next()`.

Example See `applyFilter()`.

See also `bookmark()`, `endOfSet`, `next()`

handle

The BDE handle of the object.

Property of Database, Query, Rowset, Session, StoredProc

Description The *handle* property represents the BDE handle for the object in question. The handle can be used if you want to call BDE functions directly.

See also n/a
handle is also a property of the File class (page 11-25).

indexName [Rowset]

The name of the index to use in the rowset.

Property of Rowset

Description *indexName* contains the name of the active controlling index tag for those table types that support index tags. It is set automatically when the query is activated to represent the tag used in the SQL SELECT's ORDER BY clause, if the ORDER BY is satisfied by an index. Assigning a new value to *indexName* supersedes any ORDER BY designated in the SQL SELECT statement.

For tables with primary keys, a blank *indexName* indicates that the primary key is the controlling index.

The index tag is also used in a master-detail link. The index tag of the detail rowset must match the field or fields specified in the *masterFields* property.

When specifying an *indexName* for data in a report, be sure to set the report's *autoSort* property to *false* to prevent the report from modifying the SQL statement. The modified SQL statement may generate a temporary result set that has no indexes; attempting to designate an *indexName* would cause an error.

Example In the following example, a form has a number of radio buttons with text labels like Name and Address that by design match the name of indexes in the primary rowset of the form. The following *onChange* event handler, used by all

the radio buttons, sets the index of the primary rowset to the selected radio button.

```
function indexRadio_onChange
  if this.value
    form.rowset.indexName := this.text
  endif
```

See also *autoSort* (page 17-19), *masterFields*, *setRange()*, *sql*
indexName is also a property of the UpdateSet class (page 14-77).

indexName [UpdateSet]

The name of the index to use for indexed UpdateSet operations.

Property of UpdateSet

Description The *destination* rowset or table must be indexed for the *update()*, *appendUpdate()*, and *delete()* operations. The *indexName* property specifies the key or tag name that is to be used. For tables with primary keys, the primary key is used by default. Set the *indexName* property only if you want to use another key. For DBF (dBASE) tables, you must specify an index tag name.

See also *appendUpdate()*, *delete()*, *destination*, *update()*
indexName is also a property of the Rowset class (page 14-76).

isolationLevel

Determines the isolation level of a transaction.

Property of Database

Description The *isolationLevel* property is an enumerated property that determines the isolation level of a transaction. It applies to SQL-server database transactions only. For Standard table transactions, it has no effect. These are the options:

| Value | Effect |
|-------|------------------|
| 0 | Read uncommitted |
| 1 | Read committed |
| 2 | Repeatable read |

The default is Read committed.

See also *beginTrans()*

keyViolationTableName

Name of the table in which you want to collect rows that could not be added because they would have caused a key violation.

Property of UpdateSet

Description In tables with primary keys, only one row in the table may have a particular primary key value. If the row to be added during an *append()* contains a key value that is the same as an already-existing primary key, that row cannot be added to the table, since it would have caused a primary key violation. Instead of being added to the *destination* rowset or table, that row is copied to the table specified by the *keyViolationTableName* property.

See also *append()*, *changedTableName*, *destination*, *problemTableName*

last()

Moves the row cursor to the last row in the rowset.

Syntax <oRef>.last()

<oRef> The rowset in which you want to move the row cursor.

Property of Rowset

Description Call *last()* to move the row cursor to the last row in the rowset. If a filter is active, it moves the row cursor to the last row in the rowset that matches the filter criteria.

As a navigation method, *last()* interacts with *canNavigate*, *onNavigate*, and implicit saves. For more information, see *next()*.

If the *endOfSet* property is *true* after a call to *last()*, then there are no rows that match the filter criteria if there is a filter set. If there is no filter, then that means there are no rows at all in that rowset.

Going to the last row in a rowset may not be an optimized operation on some SQL servers. For those servers, calling *last()* may take a long time for large rowsets.

See also *endOfSet*, *filter*, *first()*

length

The maximum length of the field.

Property of Field

Description A field's length represents the number of bytes used in the table for that field, and for character and numeric fields, the maximum length of the item that it can store.

For character fields, the *length* property represents the maximum number of characters in the string. Attempting to store more characters in that field results in the string being truncated.

For numeric fields, the *length* property represents the maximum number of characters in the number, including the digits, and any sign or decimal point. Attempting to store a number with more digits than the maximum results in numeric overflow, in which the actual value of the number is lost, and is simply considered to be bigger than the maximum allowed; it is usually represented by a string of asterisks.

See also *decimalLength*, *type*

live

Specifies whether the rowset can be modified.

Property of Rowset

Description Before making a query active, you can determine whether the rowset that is generated is editable or not. You can choose to make it not editable to prevent accidental modification of the data.

The *live* property is read-only.

See also *requestLive*

locateNext()

Applies the locate criteria again to search for another row.

Syntax `<oRef>.locateNext([<rows expN>])`

<oRef> The rowset in which to move the row cursor.

<rows expN> The Nth row to find. By default, the next row forward.

Property of Rowset

Description When the *applyLocate()* method is called, it moves the row cursor to the first row that matches the locate criteria. From then on, you can move forward and backward to other rows that match the same criteria by calling *locateNext()*.

locateNext() takes an optional numeric parameter that specifies in which direction, forward or backward, to look and at which match to stop, relative to the current row position. A negative number indicates a search backward, toward the first row; a positive number indicates a search forward, toward the last row. For example, a parameter of -3 means to look backward from the current row to find the third matching row.

If the row cursor encounters the end-of-set before the desired match is found, the search stops, leaving the row cursor at the end-of-set.

As a navigation method, *locateNext()* interacts with *canNavigate*, *onNavigate*, and implicit saves. For more information, see *next()*.

locateNext() returns *true* to indicate that the desired match was found and *false* to indicate that it wasn't.

See also *applyLocate()*, *beginLocate()*, *endOfSet*

locateOptions

Determines how values are matched for locating.

Property of Rowset

Description The *locateOptions* property is an enumerated property that controls how the *value* properties in the field objects entered during Locate mode are matched against the values in the table. These are the options:

| Value | Effect |
|-------|--------------------------------------|
| 0 | Match length and case |
| 1 | Match partial length |
| 2 | Ignore case |
| 3 | Match partial length and ignore case |

When matching partial length, the entire search value must match all or part of the value in the table, starting at the beginning of the field. For example, searching for "Century City", will match "Century City East", but "East" alone would not.

locateOptions also determines how fields are matched when using an SQL expression with the *applyLocate()* method.

See also *applyLocate()*, *locateNext()*

lock

The date and time of the last successful lock made to the row.

Property of LockField

Description Use *lock* after a failed lock attempt to determine the date and time of the current lock that is blocking your lock attempt. The date and time are represented in a string in the following format:

MM/DD/YY HH:MM:SS

This format is accepted by the constructor for a Date object, so you can easily convert the *update* string into an actual date/time.

This property is available only for DBF tables that have been CONVERTed.

Example The following form method attempts to lock the current row in the form's primary rowset. If the lock cannot be secured, it displays information about the current lock.

```
function lockRow()
  local cMsg
  form.rowset.parent.session.lockRetryCount := 1
  do while true
    if form.rowset.lockRow()
      return true
    else
      cMsg = "Locked by: " + form.rowset.fields[ "_DBASELOCK" ].user + chr(13) + ;
            "since: " + form.rowset.fields[ "_DBASELOCK" ].lock
      if msgbox( cMsg, "Record is locked by another", 5 + 48 ) == 2
        return false
      endif
    endif
  enddo
```

The *lockRetryCount* for the rowset's query's session is set to 1 so that the *lockRow()* method will try the lock only once before failing. If left at its default value of zero, *Visual dBASE* would display its own lock failure dialog, which doesn't display as much information, and retries continuously to get the lock, which you don't necessarily want to do.

The MSGBOX() used is a Retry/Cancel dialog box. The button number, which MSGBOX() returns, is 2 if the Cancel button is clicked or the user presses *Esc*.

See also CONVERT, *update*, *user*

lockRetryCount

The number of times to retry a lock attempt.

Property of Session

Description Any attempt to change the data in a row, for example, typing a letter in a *dataLinked* Entryfield control, causes an automatic row lock to be attempted. In

addition to the automatic row locking, you may request an explicit row or rowset lock with the *lockRow()* and *lockSet()* methods.

If someone else already has a conflicting lock, the initial lock attempt fails. The *lockRetryCount* property indicates the number of times the lock attempt will be retried, while the *lockRetryInterval* indicates the number of seconds to wait between each attempt. If after all the attempts the lock has not been secured, the lock request fails.

See also class Rowset, *lockRetryInterval*, *lockRow()*, *lockSet()*

lockRetryInterval

The number of seconds to wait between each lock retry attempt.

Property of Session

Description Any attempt to change the data in a row, for example, typing a letter in a *dataLinked* Entryfield control, causes an automatic row lock to be attempted. In addition to the automatic row locking, you may request an explicit row or rowset lock with the *lockRow()* and *lockSet()* methods.

If someone else already has a conflicting lock, the initial lock attempt fails. The *lockRetryCount* property indicates the number of times the lock attempt will be retried, while the *lockRetryInterval* indicates the number of seconds to wait between each attempt. If after all the attempts, the lock has not been secured, the lock request fails.

See also class Rowset, *lockRetryCount*, *lockRow()*, *lockSet()*

lockRow()

Attempts to lock the current row.

Syntax `<oRef>.lockRow()`
<oRef> The rowset in which you want to lock the current row.

Property of Rowset

Description An automatic row lock is attempted whenever the *value* property of a Field object is modified, either directly by assignment, or indirectly through a *dataLinked* control.

You may use *lockRow()* to attempt an explicit row lock. Whether the lock is automatic or explicit, it will fail if the current row or the entire rowset is already locked.

lockRow() returns *true* to indicate that the lock was successful and *false* to indicate that it wasn't.

Row locking support varies among different table types. The Standard (DBF and DB) tables fully support row locking; most SQL servers do not. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

Example The following is a custom Query control designed to work with a table that contains generated key values. Whenever a new key value is needed, the *genKey()* method is called. After getting an explicit row lock, the previous key value is read, incremented, and saved. The row is then unlocked.

```
class KeyGenerator of Query custom
  this.session := new Session()
  this.session.lockRetryCount := 5
  this.sql      := "select * from GENKEY"

  function genKey( cField, kGen )
    this.active := true
    this.rowset.locateOptions := 2 // Ignore case
    local xRet
    if this.rowset.applyLocate( "KEY_FIELD = '" + upper( cField ) + "'" )
      if argcount() < 2
        if isblank( this.rowset.fields[ "KEY_GEN" ].value )
          // Set default incrementor -- add one to number
          kGen := {|x| val( x ) + 1}
        else
          try
            private cb
            cb = this.rowset.fields[ "KEY_GEN" ].value
            kGen := &cb.
          catch ( Exception e )
            local e
            e = new DbException()
            e.message := "Key generator failure. " + ;
                      "Bad generator algorithm for field: " + cField
            throw e
          endtry
        endif
      endif
      do ; until this.rowset.lockRow() // Wait for lock
      xRet = "" + kGen( this.rowset.fields[ "KEY_VALUE" ].value )
      this.rowset.fields[ "KEY_VALUE" ].value := xRet
      this.rowset.save()
      this.rowset.unlock()
    else
      local e
      e = new DbException()
      e.message := "Key generator failure. Unknown key field: " + cField
      throw e
    endif
  endfunction
endclass
```

```
lockSet ( )
```

```
    this.active := false  
    return xRet
```

```
endclass
```

The Genkey table has three character fields:

- Key_field contains the name of the key field being maintained
- Key_value contains the last generated value of the key field
- Key_gen optionally contains the text of a codeblock that will generate the next key value.

If the Key_gen field is blank, the default action is to treat the key field simply as a number, and increment it. The key field is always stored as a string, so the VAL() function must be used to convert it to a number first. If a generator codeblock is specified, it is copied into a private variable and evaluated with the macro operator.

The next step is to get a lock on that field. The method uses a DO...UNTIL loop to secure the lock; it simply keeps trying until it gets it. Even if many people are attempting to get a lock, the entire process is quick enough so that there shouldn't be much if any of a noticeable delay.

After the lock is secured, the current value is passed through the generator, which returns the next value. This value is converted back into a string and stored in the table. The row is saved and unlocked. Finally, the new key value string is returned.

See also *lockSet(), lockRetryCount, unlock(), value*

lockSet()

Attempts to lock the entire rowset.

Syntax *<oRef>.lockSet()*

<oRef> The rowset you want to lock.

Property of Rowset

Description You may use *lockSet()* to attempt to lock the entire rowset. The rowset cannot be locked if someone else already has any other row or set locks on the rowset.

Set locks are session-based. Once a *lockSet()* attempt succeeds, all other *lockSet()* requests for the same set from rowsets in queries assigned to the same session will succeed. Query objects must be assigned to different Session objects for set locking to work properly.

Locking the rowset is not the same as accessing the table exclusively. Exclusive access means that you are the only one who has the table open. In contrast, locking a rowset allows others to view, but not modify, the rowset.

lockSet() returns *true* to indicate that the lock was successful and *false* to indicate that it wasn't.

Set locking support varies among different table types. The Standard (DBF and DB) tables fully support set locking, as do a few SQL servers. For servers that do not support true locks, the Borland Database Engine emulates optimistic locking. Any lock request is assumed to succeed. Later, when the actual attempt to change the data occurs, if the data has changed since the lock attempt, an error occurs.

See also *lockRow()*, *lockRetryCount*, *unlock()*, *value*

login()

Logs in user to DBF table security for a session.

Syntax `<oRef>.login(<group name expC>, <user name expC>, <password expC>)`

<oRef> The session to log into.

<group name expC> The group name.

<user name expC> The user name.

<password expC> The password.

Property of Session

Description DBF table security is session-based. All queries assigned to the same session in their *session* property have the same access level.

If someone attempts to open an encrypted table and has not logged in to the session, they will be prompted for the group name, user name, and password. Responding attempts to log the user into the session.

The *login()* method allows you to log in to the session directly. You can do this if you're assigning a default access level, so that users won't be prompted; or if you're writing your own custom login form, in which case you will need to call *login()* with the values you have gotten.

login() returns *true* or *false* to indicate whether the login was successful.

Example The following *onClick* event handler for the login button on a custom login form logs in the user with the values typed in the form and runs the main form:

```
function loginButton_onClick()
    if form.rowset.parent.session.login( form.groupNameText.value, ;
                                         form.userNameText.value, ;
                                         form.password1.value )
        do MAIN.WFM
    endif
```

See also `access()`, `addPassword()`, `user()`

loginString

The user name and password to use to log in to a database.

Property of Database

Description Some databases require that you log in to them to access their tables. When you set the Database object's *active* property to *true* to open the connection, a login dialog will appear, prompting the user for the user name and password.

You can prevent the login dialog from appearing by setting the *loginString* property to a string containing a valid user name and password of the form "userName/password". If the user name and password provided through *loginString* are not valid, the login dialog will appear when you attempt to activate the database.

lookupRowset

The rowset containing lookup values for a field.

Property of Field

Description Use *lookupSQL* or *lookupRowset* to implement automatic lookups for a field. For information on how automatic lookups work, see *lookupSQL*.

The simpler implementation is to set the *lookupSQL* property. This automatically generates a lookup rowset, which you can reference through the *lookupRowset* property.

The more advanced technique is to generate your own lookup rowset, which must follow the same structure as detailed for *lookupSQL*. Then assign a reference to this rowset to the *lookupRowset* property. Doing so releases any internal rowset generated for *lookupSQL*, if any. This technique might be used if you want to use the same lookup for multiple fields.

Example The first example updates a Text control with the address for a name chosen from a ComboBox control. The field that is *dataLinked* to the ComboBox control has a *lookupSQL* property, so the lookup rowset is automatically generated. The address is retrieved through the *lookupRowset* property. This is the field's *onChange* event handler.

```
function name_onChange
    local f,r                                // Variables for the form and lookupRowset
    f = this.parent.parent.parent.parent    // Form is 4 parents up
    r = this.lookupRowset
    if r.endOfSet                            // No match
        f.address.text := ""                // Blank text
```

```

else
    f.address.text := r.fields[ "Address" ].value + "<BR>" + ;
                    r.fields[ "City"    ].value + " " + ;
                    r.fields[ "State"   ].value + " " + ;
                    r.fields[ "Zip"     ].value
endif

```

Whenever the value in the ComboBox control changes, a lookup is performed, moving the row cursor in the lookup rowset. Wherever it is, you can retrieve the values for any other field in that row, as long as you include those fields in the SQL SELECT statement.

The second example assigns an existing rowset, opened earlier in the instantiation of the form, to two fields in the rowset in the query's *onOpen* event.

```

function inspection1_onOpen
    this.rowset.fields[ "Primary" ].lookupRowset := this.parent.inspector1.rowset
    this.rowset.fields[ "Backup"  ].lookupRowset := this.parent.inspector1.rowset

```

See also *lookupSQL*

lookupSQL

An SQL SELECT statement describing a rowset that contains lookup values for a field.

Property of Field

Description Use *lookupSQL* or *lookupRowset* to implement automatic lookups for a field. When a control that supports lookups, like the ComboBox control, is *dataLinked* to a field with either *lookupSQL* or *lookupRowset* defined, the control will:

- Populate itself with display values from the lookup rowset
- Lookup the true value of the field in the lookup rowset
- Display the corresponding lookup value in the control
- Do the reverse lookup when the display value in the control is changed
- Write the corresponding true value back to the field

If the display lookup fails, a blank is displayed in the control. If the reverse lookup fails, a *null* is written to the field.

The same automatic lookups are applied when accessing the *value* property of the field. The *value* of the field will appear to be the lookup value. Assigning to the *value* will perform the reverse lookup.

Setting the *lookupSQL* property is the simpler way of implementing automatic lookups. *lookupSQL* contains an SQL statement of the form:

```
SELECT <lookup field>, <display field> [,...] FROM <lookup table> [<options>]
```

The first two fields must be the lookup field and the display field, respectively. The display field may be a calculated field. You may include other fields so that

you can get information about the chosen row. The SQL SELECT statement may include the usual options; in particular, you may want the table to be ordered on the lookup field (or use a table where such an index is available) for faster lookups. The SQL statement is executed in the same database as the query (or stored procedure) that contains field's rowset.

When an SQL statement is assigned to *lookupSQL*, the *lookupRowset* property will contain a reference to the generated rowset. You may refer to the fields in the matched lookup row through this reference. For advanced applications, you may assign your own rowset to *lookupRowset*. This releases the generated rowset.

Example The following SQL SELECT statement is assigned to the Field object for a Customer ID field in an Orders table:

```
select CUST_ID, FIRST_NAME || ' ' || LAST_NAME as FULL_NAME from CUSTOMER
```

The || symbols act as concatenation operators in SQL (like the + operator in Visual dBASE). Now when *dataLinking* to the field or accessing the field's *value*, the full name will appear instead of the ID.

See also *lookupRowset*

lookupTable

The table used for a DB (Paradox) field's lookup.

Property of PdxField

Description *lookupTable* contains the name of the lookup table used to assist in the filling in of the field represented by the PdxField object. For more information on Paradox table lookups, see *lookupType*.

See also *lookupType*

lookupType

The type of lookup used by a DB (Paradox) field.

Property of PdxField

Description *lookupType* specifies the type of lookup used to assist in the filling in of the field represented by the *PdxField* object. It is an enumerated property that can have one of the following values:

| Value | Description |
|-------|---|
| 0 | No lookup |
| 1 | Lookup field only, no help |
| 2 | Lookup and fill all corresponding fields, no help |
| 3 | Lookup field only, with help |
| 4 | Lookup and fill all corresponding fields, with help |

Visual dBASE does not support the user interface required for Paradox lookup help. Also, validity checking is not performed whenever all corresponding fields are filled; this is so that (in Paradox) you can substitute the field value with the value of a same-named field in the lookup table that is not the lookup field.

Therefore, the only support for Paradox lookups in *Visual* dBASE is for validity checking; to make sure the value stored in the field is listed in the lookup field in the lookup table, and only when *lookupType* is set to 1 or 3. For example, a Customer ID field in an Orders table can check that the Customer ID is listed in the Customer table. An attempt to store an unlisted value in the field results in a database engine-level exception.

Consider using the automatic lookup provided by *lookupSQL* and *lookupRowset* instead.

See also *lookupTable*

masterFields

A list of fields in the master rowset that link it to the detail rowset.

Property of Rowset

Description The *masterFields* property is set in the detail rowset. It is a string that contains a list of fields in the master rowset that are matched against the detail rowset's active controlling index, as specified by the *indexName* property. By setting the property in the detail rowset, one master rowset can control multiple detail rowsets.

The *masterRowset* property should be set before *masterFields*. Once *masterFields* is set, the detail rowset is constrained to show the detail rows that match the current row in the master rowset. You may cancel the master-detail link by setting either property to an empty string.

For table formats that support multi-field indexes (DBF does not—it uses expression indexes instead), multiple fields in the *masterFields* list are separated by semicolons.

You may link the rowsets through an expression by creating a calculated field in the master rowset and using that calculated field name in the *masterFields* list.

Example Suppose you have a Customer.dbf table that you want to link to an Orders.dbf table, to show each customer's orders by date. The Customer table has an autoincrement field named Cust_id. The Orders table also has a Cust_id field and an Order_date field. The index on the Cust_id and Order_date field is defined as:

```
str( CUST_ID ) + dtos( ORDER_DATE )
```

Because the Cust_id field is converted to a string in the expression index, the Cust_id field in the Customer table must also be converted to string in a calculated field to link the two tables. (If the Cust_id field was a character field in both tables, this extra step would be unnecessary, because you could use the Cust_id field as-is to link to the expression index.)

Use the Customer query's *onOpen* event to create the calculated field, arbitrarily named Cust_link:

```
function customer1_onOpen()
    c = new Field()
    c.fieldName := "CUST_LINK"
    this.rowset.fields.add( c )
    c.beforeGetValue := {|| str( this.parent[ "CUST_ID" ].value )}
```

Note that when working in the Form (or other) designer, creating the *onOpen* event handler for the Customer query does not immediately execute it. The calculated field will not be present until the query is reexecuted. Toggling the query's *active* property alone won't work, because although the event has been assigned, its code has not been compiled and is therefore not available. You can force the designer to recompile all the code and reexecute the query by making a change in the constructor of the form (adding and removing a blank line is sufficient); then the calculated field will be present.

Once the calculated field is present (it will be in the Field palette), specify the Customer rowset as the *masterRowset* of the Orders rowset, and the Cust_link field in the *masterFields* property.

After the calculated field is created, its *beforeGetValue* will be streamed in the form class constructor in the WITH block of the query's rowset (right after the query itself). This means that the *beforeGetValue* code is present in two places, both in the constructor and the *onOpen* event handler. You can leave them both there, but if you change the code in the *onOpen*, you must either also change or remove the assignment in the WITH block, because it executes after the query's *onOpen* event. Or you can remove the code in the *onOpen*, and assign the *beforeGetValue* directly to the calculated Field object in the Inspector.

If you use this relation often, you can create and reuse a data module that contains this code.

See also *indexName*, *masterRowset*

masterRowset

A reference to the master rowset that is linked the detail rowset.

Property of Rowset

Description The *masterRowset* property is set in the detail rowset. It is an object reference to the master rowset that constrains the detail rowset. By setting the property in the detail rowset, one master rowset can control multiple detail rowsets.

The *masterRowset* property should be set before *masterFields*. Once *masterFields* is set, the detail rowset is constrained to show the detail rows that match the current row in the master rowset. You may cancel the master-detail link by setting either property to an empty string.

Example The following example links an employee to the various positions they have held over the years:

```
emp = new Query()
emp.sql = "select * from EMPLOYEE"
emp.active = true

pos = new Query()
pos.sql = "select * from POSITION"
pos.active = true
pos.rowset.indexName = "EMP_ID"

pos.rowset.masterRowset = emp.rowset    // Identify master rowset
pos.rowset.masterFields = "EMP_ID"      // Field matches index order
```

See also *indexName*, *masterFields*, *setRange()*

masterSource

A reference to the rowset that acts as the master in a master-detail link and provides parameter values.

Property of Query

Description Use *masterSource* to create a master-detail link between two queries where parameters are used in the detail query. *masterSource* is assigned a reference to the *rowset* in the master query.

By setting the *masterSource* property, the parameters in the SQL statement are automatically substituted with matching fields from the master rowset, thereby constraining the detail query. Calculated fields may be used. The fields are matched to the parameters by name. The field name match is not case-sensitive.

As navigation occurs in the *masterSource* rowset, the parameter values are resubstituted and the detail query is requested.

An alternate approach to creating a master-detail link is through the *masterRowset* and *masterFields* properties. While *masterRowset* and *masterFields* are used to link one rowset to another using an index and matching field values, *masterSource* creates a query-to-rowset link between the parameters in the detail query and the master rowset.

Example Suppose you have a table of customers named CUST, and a table of their orders named ORDERS. The customers and their orders are both identified by a customer ID field, that happens (by design) to be named CUST_ID in both tables. The following statements create a master-detail link between two queries.

```
qCust = new Query()
qCust.sql = "select * from CUST"
qCust.active = true
qOrder = new Query()
qOrder.sql = "select * from ORDERS where CUST_ID = :CUST_ID order by ORDER_DATE"
qOrder.masterSource = qCust.rowset
qCust.active = true
```

The parameter CUST_ID in the SQL statement for the ORDERS table is automatically filled in with the CUST_ID field in the CUST table.

See also *params, sql*

maximum

The maximum allowed value of a field.

Property of DbfField, PdxField

Description *maximum* specifies the maximum allowed value of the field represented by the field object. A blank value indicates no maximum. The *maximum* is the same data type as the field, except for numeric fields that have no *maximum*; in that case, *maximum* is *null*.

Only character, date, and numeric fields (all variations) have a *maximum*. DBF tables must be level 7 to support *maximum*.

If you *dataLink* a SpinBox component to a field with a *maximum*, that value becomes the default *rangeMax* property of that component.

See also *minimum*

minimum

The minimum allowed value of a field.

Property of DbfField, PdxField

Description *minimum* specifies the minimum allowed value of the field represented by the field object. A blank value indicates no minimum. The *minimum* is the same data type as the field, except for numeric fields that have no *minimum*; in that case, *minimum* is *null*.

Only character, date, and numeric fields (all variations) have a *minimum*. DBF tables must be level 7 to support *minimum*.

If you *dataLink* a SpinBox component to a field with a *minimum*, that value becomes the default *rangeMin* property of that component.

See also *maximum*

modified

A flag to indicate whether the current row has been modified.

Property of Rowset

Description The *modified* property indicates whether the current row has been modified. It is automatically set to *true* whenever the *value* of any Field object is changed, either directly by assignment, or indirectly through a *dataLinked* control.

If *modified* is *true*, then an attempt to save the row is made if there is navigation off the row or a *state* switch in the rowset. If *modified* is *false*, then this implicit save is not attempted.

modified is set to *false* whenever a row is read into the row buffer after navigating to it, is refreshed by *refreshRow()* or *refresh()*, or is saved. You may also set the *modified* property to *true* or *false* manually. For example, you can set *modified* to *false* after assigning some *value* properties during an *onAppend* event. This makes the values you filled in default values, and the row will not be automatically saved if the user does not add more information.

In addition to tracking changes during normal data entry, the *modified* property is also set to *true* during Filter and Locate modes. This allows you to determine if any criteria have been specified before attempting an *applyFilter()* or *applyLocate()*. When in either of these modes, navigation cancels the mode and moves the row cursor relative to the last row position, but no save is attempted, even if *modified* is *true*.

Example The following example is the *onClick* event handler for a Reply button in an E-mail viewer. It copies the name from the From field of the original to the To field of the reply and duplicates the Subject field. After setting the *value* properties, the rowset's *modified* property is set to *false* to indicate that these are the default values.

```
function replyButton_onClick()
    local cTo, cSubject
```

```
next ( )
```

```
cTo      = form.rowset.fields[ "From"    ].value
cSubject = form.rowset.fields[ "Subject" ].value
if form.rowset.beginAppend()
    form.rowset.fields[ "To"      ].value = cTo
    form.rowset.fields[ "Subject" ].value = cSubject
    form.rowset.modified = false
endif
```

See also `refresh()`, `refreshRow()`, `value`

next()

Moves the row cursor to another row relative to the current position.

Syntax `<oRef>.next([<rows expN>])`

<oRef> The rowset in which you want to move the row cursor.

<rows expN> The number of rows you want to move. By default, the next row forward.

Property of Rowset

Description `next()` takes an optional numeric parameter that specifies in which direction, forward or backward, to move and how many rows to move through, relative to the current row position. A negative number indicates a search backward, toward the first row; a positive number indicates a search forward, toward the last row. For example, a parameter of 2 means to move forward two rows.

If a filter is active, it is honored.

If the row cursor encounters the end-of-set while moving, the movement stops, leaving the row cursor at the end-of-set, and `next()` returns *false*. Otherwise `next()` returns *true*.

Navigation methods such as `next()` will cause the rowset to attempt an implicit save if the rowset's *modified* property is *true*. The order of events when calling `next()` is as follows:

- 1 If the rowset has a *canNavigate* event handler, it is called. If not, it's as if *canNavigate* returns *true*.
- 2 If the *canNavigate* event handler returns *false*, nothing else happens and `next()` returns *false*.
- 3 If the *canNavigate* event handler returns *true*, the rowset's *modified* property is checked.
- 4 If *modified* is *true*:
 - 1 The rowset's *canSave* event is fired. If there is no *canSave* event, it's as if *canSave* returns *true*.
 - 2 If *canSave* returns *false*, nothing else happens and `next()` returns *false*.

- 3 If *canSave* returns *true*, Visual dBASE tries to save the row. If the row is not saved, perhaps because it fails some database engine-level validation, a *DbException* occurs—*next()* does not return.
- 4 If the row is saved, the *modified* property is set to *false*, and the *onSave* event is fired.
- 5 After the current row is saved (if necessary):
 - 1 The row cursor moves to the designated row.
 - 2 The *onNavigate* event fires.
 - 3 *next()* returns *true* (if the navigation did not end up at the end-of-set).

Other navigation methods go through a similar chain of events.

See also *endOfSet*, *filter*, *locateNext()*

notifyControls

Specifies whether *dataLinked* controls are updated as field values change or the row cursor moves.

Property of Rowset

Description *notifyControls* is usually *true* so that *dataLinked* controls are automatically updated as you navigate from row to row or when you directly assign values to the *value* property of Field objects.

You may set *notifyControls* to *false* if you are performing some data manipulation and don't want the overhead of constantly updating the controls.

When *notifyControls* is set to *true*, the controls are always refreshed, as if *refreshControls()* was called.

See also *refreshControls()*, *value*

onAbandon

Event fired after the rowset is successfully abandoned.

Parameters none

Property of Rowset

Description A rowset may be abandoned explicitly by calling its *abandon()* method, or implicitly via the user interface by pressing Esc or choosing Abandon Row from the default Table menu or toolbar while editing table rows. While the *canAbandon* event fires first to see if the abandon actually takes place, *onAbandon* fires after the abandon occurs.

If you are abandoning changes made to a row, the row is automatically refreshed, so there is no need to call *refreshRow()* in the *onAbandon*. However, this is not considered navigation, so if you have an *onNavigate* event handler, you should call it from *onAbandon*.

Example This basic *onAbandon* event handler calls *onNavigate* if one is defined.

```
function Rowset_onAbandon
    if not empty( this.onNavigate )
        this.onNavigate()
    endif
```

See also *abandon()*, *canAbandon*

onAppend

Event fired after the rowset successfully enters Append mode.

Parameters none

Property of Rowset

Description A rowset may be put in Append mode explicitly by calling its *beginAppend()* method, or implicitly via the user interface by choosing Append Row from the default Table menu or toolbar while editing table rows. While the *canAppend* event fires first to see if the new append actually takes place, *onAppend* fires after the row buffer has been cleared and is ready for new values.

You can use *onAppend* to do things like automatically time stamp the new row or fill in default values. If you use *onAppend* to set field values, set the *modified* property to *false* at the end of the event handler to indicate that the row hasn't been changed by the user. This way, if the user does not add any more data, the row will not be saved automatically if they navigate to another row or try to append another.

Example The following example saves the user's network ID to all newly created rows.

```
function invoice_onAppend()
    this.fields[ "USER_ID" ].value := id()
    this.modified := false
```

This event handler could be used in combination with a table that has default values set for certain fields, like a timestamp for the date and time of entry. The user's network ID is not something the database engine can get, so you have to set this manually.

See also *beginAppend()*, *canAppend*, *modified*

onChange

Event fired after a field's *value* property is successfully changed.

Parameters none

Property of Field (including DbfField, PdxField, SqlField)

Description A Field object's *value* property may be changed directly by assigning a value to it, or indirectly through a *dataLinked* control. When assigning a value, the change occurs during the assignment statement. When using a *dataLinked* control, the change doesn't happen until the user tries to move the focus to another control. In both cases, *canChange* fires first to see if the change can actually take place. If it does, the value is changed and then *onChange* is fired.

Example The following *canChange* and *onChange* event handlers work together to record all change attempts to a field (even if the modified row is abandoned later). An audit table is open in another query on the form.

```
function Field_canChange()
    this.initValue = this.value    // Save initial value
    return true                    // Always allow change

function Field_onChange()
    if not this.initValue == this.value    // Perform exact comparison
        local r                          // Assign reference to audit rowset
        r = this.parent.parent.parent.audit1.rowset
        if r.beginAppend()
            r.fields[ "Field" ].value := this.fieldName
            r.fields[ "Old value" ].value := this.initValue
            r.fields[ "New value" ].value := this.value
            r.fields[ "User" ].value := id()
            r.save()
        endif
    endif
endif
```

See also *canChange*, *value*

onChange is also an event of the ListBox, ComboBox, Entryfield, and Editor classes (page 15-127).

onClose

Event fired after a query or stored procedure is successfully closed.

Parameters none

Property of Query, StoredProc

Description An attempt to close a query or stored procedure occurs when its *active* property, or the *active* property of the object's database, is set to *false*. If the object's rowset has been modified, *Visual* dBASE will try to save it, so the close attempt can be canceled by the rowset's *canSave* event handler. If not, the row is saved.

The close can also be prevented by the Query or StoredProc object's *canClose* event handler. If not, the object is closed, and its *onClose* event fires.

Because *onClose* fires after the rowset has closed, you can no longer affect its fields. If you want to do something with the rowset's data when the rowset closes, use the *canClose* event instead, and have the event handler return *true*.

See also *active, canClose, canSave*

onDelete

Event fired after a row is successfully deleted.

Parameters none

Property of Rowset

Description A row may be deleted explicitly by calling the *delete()* method, or implicitly via the user interface by choosing Delete Rows from the default Table menu or toolbar while editing table rows. While the *canDelete* fires first to determine if the row is actually deleted, *onDelete* fires after the row has been removed.

Because the row has been removed by the time *onDelete* fires, the row cursor is at the next row or the end-of-set when *onDelete* fires. However, this movement is not considered navigation, so if you have an *onNavigate* event handler, you should call it from *onDelete*.

See also *canDelete, delete()*

onEdit

Event fired after the rowset successfully enters Edit mode.

Parameters none

Property of Rowset

Description The *beginEdit()* method is called (implicitly or explicitly) to put the rowset in Edit mode. While the *canEdit* event fires first to see if the switch to Edit mode actually takes place, *onEdit* fires after the rowset has switched to Edit mode.

You can use *onEdit* to do things like automatically record when edits take place, or to save original values for auditing.

See also *autoEdit*, *beginEdit*, *canEdit*

onGotValue

Event fired after a field's *value* property is successfully read.

Parameters none

Property of Field (including DbfField, PdxField, SqlField)

Description *onGotValue* is fired when reading a field's *value* property explicitly and when it is read to update a *dataLinked* control. It does not fire when the field is accessed internally for SpeedFilters, index expressions, or master-detail links, or when calling *copyToFile()*.

See also *beforeGetValue*, *value*

onNavigate

Event fired after successful navigation in a rowset.

Parameters **<method expN>** Numeric value that indicates which method was called to fire the event:

| Value | Method |
|-------|----------------------|
| 1 | <i>next()</i> |
| 2 | <i>first()</i> |
| 3 | <i>last()</i> |
| 4 | All other navigation |

<rows expN> Number of rows *next()* method was called with. Zero if *next()* was not used.

Property of Rowset

Description Navigation in a rowset may occur explicitly by calling a navigation method like *next()* or *goto()*, or implicitly via the user interface by choosing a navigation option from the default Table menu or toolbar while viewing a rowset. While *canNavigate* fires first before the row cursor has moved to see if the navigation actually takes place, *onNavigate* fires after the row position has settled on the desired row or end-of-set.

Because *onNavigate* fires when moving to the end-of-set and you cannot access field values when you're at the end-of-set, you may want to test the rowset's *endOfSet* property before you attempt to access field values in your *onNavigate* handler.

You can use *onNavigate* to update non-*dataLinked* controls or calculated fields. In that case, you may want to call your *onNavigate* handler from the *onOpen* event as well, so that these objects are up-to-date when the rowset first opens.

When navigation occurs because a row has been abandoned or deleted, *onNavigate* does not fire. Call the *onNavigate* event handler from the *onAbandon* and *onDelete* event handler.

Example The following *onNavigate* event handler calls a custom form method called *refreshUnlinked()*, which has the job of updating any controls that are not *dataLinked* to the rowset. It also displays a message if the end-of-set has been reached.

```
function Rowset_onNavigate()
  if this.endOfSet
    msgbox( "No more entries", "Alert", 48 )
  else
    this.parent.parent.refreshUnlinked()
  endif
```

In most applications, when navigating to the end-of-set the row cursor is always put back to the previous valid row. Therefore, the message displayed here will appear when the row cursor is on the end-of-set. Once the dialog box is dismissed, the row cursor will be moved back. There is no reason to call *refreshUnlinked()* when at end-of-set either, because the navigation that follows will cause the method to be called again.

See also *canNavigate*, *first()*, *goto()*, *last()*, *next()*, *onOpen*
onNavigate is also an event of the Form class (page 15-138).

onOpen

Event fired after query or stored procedure is opened successfully.

Parameters none

Property of Query, StoredProc

Description *onOpen* fires after the Query or StoredProc object has successfully opened after its *active* property has been set to *true*.

Example The following *onOpen* event handler adds a calculated field to a query.

```
function invoice1_onOpen()
  c = new Field()
  c.fieldName := "Total"
  this.rowset.fields.add( c )
  c.beforeGetValue := {|this.parent["Qty"].value * this.parent["PricePer"].value}
```

Note that the *this* in the second statement refers to the query, but in the codeblock, *this* refers to the calculated field.

See also *canOpen*

onOpen is also an event of most form objects (page 15-139).

onProgress

Event fired periodically during long-running data processing operations.

Parameters **<percent expN>** The approximate percent-complete of the operation, from 0 to 100. When a message is passed, <percent expN> is the value -1.

<message expC> A text message from the database engine.

Property of Session

Description Use *onProgress* to display progress information during data processing operations such as copying or indexing.

onProgress fires for the following operations:

| | | | |
|--------------------------|------------|-----------------------|-------------|
| Database::copyTable() | COPY TABLE | All UpdateSet methods | APPEND FROM |
| Database::createIndex() | COPY TO | INDEX ON | SORT |

The *onProgress* event handler receives two parameters, but only one of them is valid for any given event. You may get either:

- A percent-complete from 0 to 100 in <percent expN>, in which case <message expC> is a blank string, or
- A message in <message expC>, in which case <percent expN> is -1.

Example Suppose you want to rebuild an index tag as part of a table maintenance feature, and while it's working, display progress information in a form. You could create a form with a Progress control to display the percentage complete and a Text control for messages. Here are the two pertinent form methods:

```
function Form_onOpen
  _app.session.onProgress = class::onProgress
  _app.session.form       = form
  index on LAST_NAME + FIRST_NAME tag FULL_NAME
  _app.session.onProgress = null

function onProgress( nPercent, msg )
  if nPercent >= 0
    this.form.progress1.value = nPercent
  elseif not empty( msg )
    this.form.message1.text    = msg
  endif
```

By using an *onOpen* event, everything happens simply by opening the form. First, the *onProgress()* method in the form is assigned as the *onProgress* event handler for the *_app* object's *session*. Then a reference to the current form is

assigned as a property of the session so that the session's event handler can easily access the form. Then the actual indexing is performed, and when it's done, the *onProgress* event is cleared by assigning *null*.

In the *onProgress()* method, the *nPercent* parameter is checked. If it's greater than zero, the method is being passed an updated percentage, so the *Progress* control is updated. Otherwise, if the *msg* parameter is not blank, its contents are displayed in the *Text* control on the form.

See also none

onSave

Event fired after successfully saving the row buffer.

Parameters none

Property of Rowset

Description The row buffer may be saved explicitly by calling *save()*, or implicitly by navigating in the rowset or closing the rowset. While *canSave* is fired first to verify that data is good before allowing it to be written, *onSave* fires after the row has been saved.

Example The following *onSave* event handler calls the rowset's *flush()* method to make sure that the data is written to disk as each record is saved:

```
function Rowset_onSave
    this.flush()
```

See also *canSave*, *save()*

open()

Opens a database connection.

Syntax This method is called implicitly by the Database object.

Property of Database

Description The *open()* method opens the database connection. It is called implicitly when you set the Database object's *active* property to *true*. In typical usage, you do not call this method directly.

Advanced applications may override the definition of this method to perform supplementary actions when opening the database connection. Custom data drivers must define this method to perform the appropriate actions to open their database connection.

See also *active*, *close*()

packTable()

Packs a Standard table by removing all deleted rows.

Syntax `<oRef>.packTable(<table name expC>)`
<oRef> The database in which the table exists.
<table name expC> The name of the table you want to pack.

Property of Database

Description For DBF (dBASE) tables, *packTable*() removes all the records in a table that have been marked as deleted, making all the remaining records contiguous. As a result, the records are assigned new record numbers and the disk space used is reduced to reflect the actual number of records in the table.

For DB (Paradox) tables, *packTable*() removes all deleted records and redistributes the remaining records in the record blocks, optimizing the block structure.

Packing is a maintenance operation and requires exclusive access to the table; no one else may have it open at the time, or *packTable*() will fail.

To refer to a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].packTable( "Customer" )
```

See also *delete*(), *emptyTable*()

params

Parameters for an SQL statement or stored procedure call.

Property of Query, StoredProc

Description The *params* property contains an associative array that contains parameter names and values, if any, for an SQL statement in a Query object or a stored procedure call in a StoredProc object.

For a Query object, assigning an SQL statement with parameters to the *sql* property automatically creates the corresponding elements in the *params* array. Parameters are indicated by colons. The values you want to substitute are then assigned to the array elements in one of two ways:

- Manually, before the query is activated or requiered with *requery*().

- By assigning a *masterSource* to the query, in which case parameters are substituted with the matching fields from the *fields* array of the *masterSource*'s *rowset*. Parameters are matched to fields by name.

For a StoredProc object, the Borland Database Engine will try to get the names and types of any parameters needed by a stored procedure, once the procedure name is assigned to the *procedureName* property. This works to varying degrees for most SQL servers. If it succeeds, the *params* array is filled automatically with the corresponding Parameter objects. You must then assign the values you want to substitute to the *value* property of those objects.

For SQL servers that do not return the necessary stored procedure information, include the parameters, preceded with colons, in parentheses after the procedure name. The corresponding Parameter objects in the *params* array will be created for you; then you must assign the necessary *type* and *value* information.

Example The following statements create a query with a parameter. The parameter in the SQL statement, preceded by a colon, automatically creates the corresponding element in the *params* array.

```
q = new Query()
q.sql = "select * from CUST where CUST_ID = :custid"
q.params[ "custid" ] = 123
q.active = true
```

See also *masterSource, procedureName, requery(), sql*

picture

A template that formats input to a DB (Paradox) field.

Property of PdxField

Description A *picture* uses special template symbols to format data entry into a field. However, many Paradox template symbols do not match *Visual* dBASE template symbols, so a *picture* for a DB field probably won't work as-is in the *picture* property of a control unless it's very simple, for example "999.99".

Visual dBASE does not enforce the DB field template. The *picture* property is informational only.

picture is also a property of some form components (page 15-148).

precision

The number of digits of precision of an SQL-based field.

Property of SqlField

Description *precision* represents the numeric accuracy to which numbers are stored in the field represented by the `SqlField` object.

See also *scale*

prepare()

Prepares an SQL statement or stored procedure.

Syntax `<oRef>.prepare()`

<oRef> The object you want to prepare.

Property of Query, StoredProc

Description *prepare()* prepares the stored procedure named in the *procedureName* property of a `StoredProc` object or the SQL statement stored in the *sql* property of a `Query` object. If the object is connected to an SQL-server-based database, the prepare message is passed on to the server.

Preparing an SQL statement or stored procedure call includes compiling the statement and setting up any optimizations. If the statement includes parameters, the statement can be prepared first, and, sometime later, you can get the parameter values from the client. Then the prepared statement and its parameters are ready for execution. By separating the client and server activities, things run a bit faster.

Preparing is part of the process that occurs when you set an object's *active* property to *true*, so you're never required to call *prepare()* explicitly.

Example In this example, a query is executed using a value that is entered by the user. You can prepare the query first, placing the parameter in the SQL statement with a colon in front of it:

```
q = new Query()
q.database = someDatabase
q.sql = "select * from EMPLOYEE where EMP_ID = :id"
q.prepare()
```

Then when the user enters the value of the parameter, you assign it to the query's *params* array and execute the query:

```
q.params[ "id" ] = someForm.empIdText.value
q.active = true
```

Because the query has already been prepared, making it active takes less time. Later, when the parameter changes, you reassign the parameter and requery:

```
q.params[ "id" ] = someForm.empIdText.value
q.requery()
```

See also *requery()*

problemTableName

Name of the table in which you want to collect rows that could not be used during an update operation because of some problem other than a key violation.

Property of UpdateSet

Description In addition to key violations, problems during update operations are often caused by things like mismatched fields. If a row could not be transferred from the *source* to the *destination* because of a problem, it is instead copied to the table specified by the *problemTableName* property.

See also *changedTableName, destination, keyViolationTableName, source*

procedureName

The name of the stored procedure to call.

Property of StoredProc

Description Set the *procedureName* property to the name of the procedure to call. The Borland Database Engine will try to get the names and types of any parameters needed by the stored procedure.

The following databases return complete parameter name and type information:

- InterBase
- Oracle
- ODBC, if the particular ODBC driver provides it

The following databases return the parameter name but not the type:

- Microsoft SQL Server
- Sybase

The following database does not return any parameter information:

- Informix

If the BDE can get the parameter names, the *params* array is filled automatically with the corresponding Parameter objects. You must then assign the values to substitute to the *value* property of those objects.

For SQL servers that do not return the necessary stored procedure information, include the parameters, preceded with colons, in parentheses after the procedure name. Empty Parameter objects will be created.

If the *type* of the parameter or the data type of the *value* for output parameters is not provided automatically, it must be set before calling the stored procedure, in addition to any input values.

Example The following statements call a stored procedure that returns an output parameter. The result is displayed in the result pane of the Command window.

```
d = new Database()
d.databaseName = "IBLOCAL"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "DEPT_BUDGET"
p.params[ "DNO" ].value = "670"
p.active = true
? p.params[ "TOT" ].value // Display output
```

The following statement calls a stored procedure in a database that does not return any parameter information. Therefore, the parameters must be declared in the *procedureName* property. Note that the parameter names are case-sensitive, and you must initialize any output parameters by assigning a dummy value of the correct data type.

```
#define PARAMETER_TYPE_INPUT      0
#define PARAMETER_TYPE_OUTPUT    1
#define PARAMETER_TYPE_INPUT_OUTPUT 2
#define PARAMETER_TYPE_RESULT    3

d = new Database()
d.databaseName = "WIDGETS"
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "PROJECT_SALES( :month, :units )"
p.params[ "month" ].type = PARAMETER_TYPE_INPUT
p.params[ "month" ].value = 6
p.params[ "units" ].type = PARAMETER_TYPE_OUTPUT
p.params[ "units" ].value = 0 // Output will be numeric
p.active = true
? p.params[ "TOT" ].value // Display output
```

See also *params*

readOnly

Whether a DBF (dBASE) or DB (Paradox) field is read-only.

Property of DbfField, PdxField

Description *readOnly* indicates whether the field represented by the Field object is read-only or not.

See also *required*

ref

A reference to the active data module object.

Property of DataModRef

Description After activating the DataModRef object, you may reference the data module object through the DataModRef object's *ref* property.

Example The following statement, generated by the Form designer, assigns the data module's primary rowset, the one in its *teacher1* query, as the form's primary rowset.

```
this.rowset = this.dataModRef1.ref.teacher1.rowset
```

See also *active, filename*

ref is also a property of the ReportViewer class (page 15-155).

refresh()

Refreshes data in the entire rowset.

Syntax `<oRef>.refresh()`

<oRef> The rowset you want to refresh.

Property of Rowset

Description To increase performance, rows are cached in memory as they are encountered. If the row cursor revisits a cached row, it can be reread quickly from memory instead of the disk. *refresh()* purges all cached rows—not to be confused with cached updates—for the rowset, forcing *Visual dBASE* to reread the data from disk. It discards any changes to the row buffer, so a row that has been modified is not saved. When the rowset is refreshed, any *dataLinked* controls are also refreshed with values for the current row if *notifyControls* is *true*.

refresh() does not regenerate the rowset. If the rowset is not *live*, *refresh()* has no effect. Use *requery()* to regenerate the rowset.

See also *flush()*, *live*, *refreshControls()*, *refreshRow()*, *requery()*, *requestLive*

refresh() is also a method of the Form class (page 15-155)

refreshControls()

Refreshes any controls that are *dataLinked* to the current row.

Syntax <oRef>.refreshControls()

<oRef> The rowset you want to refresh.

Property of Rowset

Description *refreshControls()* updates any controls that are *dataLinked* to Field objects in the rowset, regardless of the setting of the *notifyControls* property. The controls are updated with the values in the row buffer, not the values on disk.

Use *refreshRow()* first to refresh the fields in the row buffer with the values on disk if desired.

See also *notifyControls*, *refreshRow()*

refreshRow()

Refreshes data in the current row.

Syntax <oRef>.refresh()

<oRef> The rowset you want to refresh.

Property of Rowset

Description *refreshRow()* rereads the data for the current row from disk. It discards any changes to the row buffer, so a row that has been modified is not saved. When the row is refreshed, any *dataLinked* controls are also refreshed if *notifyControls* is *true*.

Use *refresh()* to refresh the entire rowset.

See also *notifyControls*, *refreshControls()*, *refresh()*

reindex()

Rebuilds a Standard table's indexes from scratch.

Syntax <oRef>.reindex(<table name expC>)

<oRef> The database in which the table exists.

<table name expC> The name of the table you want to reindex.

`renameTable ()`

Property of Database

Description Indexes can become unbalanced during normal use. Occasionally, they can also be corrupted. In both cases, you can fix the problem by using *reindex()*, which rebuilds the indexes from scratch.

Reindexing is a maintenance operation and requires exclusive access to the table; no one else may have it open at the time, or *reindex()* will fail.

To refer to a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].reindex( "Customer" )
```

See also n/a

renameTable()

Renames a table in a database.

Syntax `<oRef>.renameTable(<old name expC>, <new name expC>)`

<oRef> The database in which to rename the table.

<old name expC> The current name of the table.

<new name expC> The new name of the table.

Property of Database

Description *renameTable()* renames a table in a database, including all secondary files such as index and memo files.

The table to rename should not be open.

To rename a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].renameTable( "Before", "After" )
```

See also *copyTable()*

replaceFromFile()

Copies the contents of a file into a BLOB field.

Syntax `<oRef>.replaceFromFile(<file name expC> [, <append expl>])`

<oRef> The BLOB field you want to copy into.

<file name expC> The name of the file you want to copy.

<append expL> Whether to append the new data or overwrite.

Property of Field

Description *replaceFromFile()* copies the contents of the named file into the specified BLOB field.

By specifying *<append expL>* as *true*, the contents of the file are added to the end of the current contents of the BLOB field. If the parameter is specified as *false* or left out, the BLOB field will be overwritten and end up containing only the contents of the file.

Example The following event handler copies the contents of an image file on disk to a binary field named Mugshot in the current row.

```
function importImageButton_onClick
    local cFile
    cFile = getFile( "*.bmp", "Import mugshot image" )
    if "" # cFile
        form.rowset.fields[ "Mugshot" ].replaceFromFile( cFile )
    endif
```

See also *copyToFile()*

requery()

Re-executes the query or stored procedure, regenerating the rowset.

Syntax *<oRef>.requery()*

<oRef> The query or stored procedure you want to re-execute.

Property of Query, StoredProc

Description *requery()* re-executes a stored procedure or a query's SQL statement, generating an up-to-date rowset. Calling *requery()* is similar to setting the object's *active* property to *false* and back to *true*, except that *requery()* does not prepare the SQL statement. This includes attempting to save the current row if necessary and closing the object, firing all the events along the way. If those actions are halted by the *canSave* or *canClose* event handlers, the *requery()* attempt will stop at that point.

Use *requery()* when a parameter in the SQL statement has changed to re-execute the query with the new value.

Use *refresh()* to refresh the rowset without re-executing the query, which is faster. But *refresh()* has no effect on a rowset that is not *live*; use *requery()* instead.

Example In this example, a query is executed using a value that is entered by the user. You can prepare the query first, placing the parameter in the SQL statement with a colon in front of it:

```
q = new Query()
q.database = someDatabase
q.sql = "select * from EMPLOYEE where EMP_ID = :id"
q.prepare()
```

Then when the user enters the value of the parameter, you assign it to the query's *params* array and execute the query:

```
q.params[ "id" ] = someForm.empIdText.value
q.active = true
```

Because the query has already been prepared, making it active takes less time. Later, when the parameter changes, you reassign the parameter and requery:

```
q.params[ "id" ] = someForm.empIdText.value
q.requery()
```

See also *active*, *live*, *prepare()*, *refresh()*, *requestLive*

requestLive

Specifies whether the query should generate an editable rowset.

Property of Query

Description Before making a query active, you can determine whether the rowset that is generated is editable or not. You can choose to make it not editable to prevent accidental modification of the data.

requestLive defaults to *true*.

See also *live*

required

Whether a field is required to be filled in and not left blank.

Property of DbfField, PdxField

Description *required* indicates whether the field represented by the Field object is a required field; that is, whether it must be filled in.

See also *readOnly*

rollback()

Cancels the transaction by undoing all logged changes

Syntax <oRef>.rollback()

<oRef> The database whose changes you want to rollback.

Property of Database

Description A transaction works by logging all changes. If an error occurs while attempting one of the changes, or the changes need to be undone for some other reason, the transaction is canceled by calling the *rollback()* method. Otherwise, *commit()* is called to clear the transaction log, thereby indicating that all the changes in the transaction were committed and that the transaction as a whole was posted.

Since new rows have already been written to disk, rows that were added during the transaction are deleted. In the case of DBF (dBASE) tables, the rows are marked as deleted, but are not physically removed from the table. If you want to actually remove them, you can pack the table with *packTable()*. Rows that were just edited are returned to their saved values.

All locks made during a transaction are maintained until the transaction is completed. This ensures that no one else can make any changes until the transaction is committed or abandoned.

Example See the example for *beginTrans()* for an example using *rollback()*.

See also *beginTrans()*, *cacheUpdates*, *commit()*

rowCount()

Returns the logical row count.

Property of Rowset

Description *rowCount()* returns the logical row count of the rowset, if known. The logical row count is the number of rows in the rowset, using the rowset's current index and filter conditions.

Determining the logical row count is often an expensive operation, requiring that the rows actually be counted individually. When the count is not known, *rowCount()* returns the value -1; it does not attempt to get the count. If your application requires the actual row count, use the *count()* method to count the rows if *rowCount()* returns -1.

Note *rowCount()* is different from the function RECCOUNT(). RECCOUNT() returns the number of physical records in a table. *rowCount()* returns the logical count in a rowset. These numbers are not guaranteed to be the same, even with

a SELECT * query of a DBF table, because *rowCount()* must consider deleted records—it does not know if there are any unless it actually looks—while RECCOUNT() does not.

See also *count(), rowNo()*

rowNo()

Returns the current logical row number in the rowset.

Property of Rowset

Description *rowNo()* returns the current logical row number in the rowset, if known. The logical row number is the relative row number, using the rowset's current index and filter conditions. The first row is row number 1 and the last row is equal to the number of rows in the current rowset.

In some cases, for example scrolling with the scrollbar in a grid to an arbitrary location and clicking on a row, the logical row number is not known, and would have to be calculated. In contrast, if you were to page down repeatedly to that same location, the row number is known, because it is updated as you move from page to page in the grid. When the row number is not known, *rowNo()* returns the value -1.

Note *rowNo()* is different from the function RECNO(). RECNO() returns the physical record number of the current row in a DBF table, which never changes (unless the table is PACKed). *rowNo()* returns the logical row number; the same physical record will have a different logical row number, depending on the current index and filter.

See also *bookmark(), count(), rowCount()*

rowset

A reference to the query's or stored procedure's rowset, or a data module's primary rowset.

Property of DataModule, Query, StoredProc

Description A Query object always contains a *rowset* property, but that property does not refer to a valid Rowset object until the query has been activated and the rowset has been opened.

Some stored procedures generate rowsets. If that is the case, the StoredProc object's *rowset* property refers to that rowset after the stored procedure is executed.

A data module may designate a primary rowset. This rowset is assigned to a form's *rowset* property by the Form designer when the data module is used in the form.

For Query and StoredProc objects, the *rowset* property is read-only.

See also *active, fields*

rowset is also a property of the Form and StreamSource classes (page 15-157).

save()

Saves the current row buffer.

Syntax <oRef>.save()
 <oRef> The rowset you want to save.

Property of Rowset

Description A row is saved automatically if it has been modified and there is either navigation in the rowset, a mode change, or the rowset's query is closed. You may call *save()* explicitly to write the row buffer. By design, *save()* has no effect if the rowset's *modified* property is *false*, because supposedly there are no changes to save; and a successful *save()* sets the *modified* property to *false*, to reflect that the values in the controls are the ones on disk. You can manipulate the *modified* property to control this designed behavior.

The *canSave* event fires after calling *save()*. If there is no *canSave* event handler, or *canSave* returns *true*, then the row buffer is saved, the *modified* property is set to *false*, and the *onSave* event fires.

The row cursor does not move after a *save()* unless the values that were saved cause the row to become out-of-set, in which case the row cursor is moved to the next available row, or the end-of-set if there are no more available rows.

Changes are written to disk unless the *cacheUpdates* property is set to *true*, in which case the changes are cached. Whether the changes are actually written to a physical disk depends on the operating system and its own disk caches, if any.

See also *cacheUpdates, canSave, flush(), modified, onSave*

scale

The scale of an SQL-based field.

Property of SqlField

Description *scale* represents the scale of the field represented by the SqlField object.

See also *precision*

session

The Session object to which the database, query, or stored procedure is assigned.

Property of Database, Query, StoredProc

Description A database must be assigned to a session. When created, a Database object is assigned to the default session.

A query or stored procedure must be assigned to a database, which in turn is assigned to a session. When created, a Query or StoredProc object is assigned to the default database in the default session.

To assign the object to the default database in another session, assign that session to the *session* property. Assigning the *session* property always sets the *database* property to the default database in that session.

To assign the object to another database in another session, assign the object to that session first. This makes the databases in that session available to the object.

See also class Session

setRange()

Constrains the rowset to those rows whose key field values falls within a range.

Syntax `<oRef>.setRange(<key exp>)`

or

`<oRef>.setRange(<startKey exp> | null, <endKey exp> | null)`

<oRef> The rowset you want to constrain.

<key exp> Shows only those rows whose key value matches *<key exp>*.

<startKey exp> Shows those rows whose key value is equal to or greater than *<startKey exp>*.

<endKey exp> Shows those rows whose key value is less than or equal to *<endKey exp>*.

There are four ways to use *setRange()*:

- Exact match: *setRange(<key exp>)*
- Range from start to end: *setRange(<startKey exp>, <endKey exp>)*
- Range from starting value: *setRange(<startKey exp>, null)*

- Range up to ending value: *setRange(null, <endKey exp>)*

Property of Rowset

Description *setRange()* is similar to a filter; *setRange* uses the rowset's current index (represented by its *indexName* property) and shows only those rows whose key value matches a single value or falls within a range of values. This is referred to as a key constraint. Because it uses an index, a key constraint is instantaneous, while a filter condition must be evaluated for each row. Use *clearRange()* to remove the constraint.

The key range values must match the key expression of the index. For example, if the index key is UPPER(Name), specify uppercase letters in the range expressions. For character expressions, the key match is always a partial string match (starting at the beginning of the expression); therefore, an exact match with *<key exp>* could match multiple key values if the *<key exp>* is shorter than the key expression.

When you use both *setRange()* and a filter (and *canGetRow*) for the same rowset, you get those rows that are within the index range and that also meet the filter condition(s).

Rowsets that use *masterRowset* for master-detail linkage internally apply *setRange()* in the detail rowset. If you use *setRange()* in the detail rowset, it overrides the master-detail key constraint. Navigation in the master rowset would reapply the master-detail constraint.

See also *clearRange()*, *filter*, *indexName*, *masterRowset*

share

How to share data access resources.

Property of Database, DataModRef

Description The *share* property controls how database connections and data modules are shared. *share* is an enumerated property that can be one of the following:

| Value | Description |
|-------|-------------|
| 0 | None |
| 1 | All |

Database objects Multiple Database objects may share the same database connection. Sharing database connections reduces resource usage on both the client and server. Some servers have a maximum number of simultaneous connections, so sharing connections will also allow more users to connect to the server.

When set to All (the default), all Database objects with the same *databaseName* property (running in the same instance of *Visual dBASE*) will share the same database connection. When set to None, each Database object will use its own connection.

DataModref objects When set to All, all DataModRef objects with the same *dataModClass* property will share the same instance of that class; the same DataModule object. This means that, for example navigation performed by one user of the DataModRef is seen by all users of that same *dataModClass* if their *share* property is also All. Data module sharing is only useful in limited cases. For typical usage, *share* should be None, the default.

source

The source rowset or table of an UpdateSet operation.

Property of UpdateSet

Description The *source* property contains an object reference to a rowset or the name of a table that is the source of an UpdateSet operation. For an *append()*, *update()*, or *appendUpdate()*, it refers to the rowset or table that contains the new data. For a *copy()*, it refers to the rowset or table that is to be duplicated. For a *delete()*, the *source* property refers to the table that contains the list of rows to be deleted.

The *destination* property specifies the other end of the UpdateSet operation.

See also *append()*, *appendUpdate()*, *copy()*, *delete()*, *destination*, *update()*

sql

The SQL statement that describes the query.

Property of Query

Description The *sql* property of a Query object contains an SQL SELECT statement that describes the rowset to be generated. To use a stored procedure in an SQL server that returns a rowset, use the *procedureName* property of a StoredProc object instead.

The *sql* property must be assigned before the Query object is activated.

The SQL SELECT statement may contain an ORDER BY clause to set the row order, a WHERE clause to select a subset of rows, perform a JOIN, or any other SQL SELECT clause.

But to take full advantage of the data access objects' features—such as locating and filtering—with SQL-server-based tables, the SQL SELECT used to access a table must be a simple SELECT: all the fields from a single table, with no options. For example,

```
select * from CUSTOMER
```

If the SQL statement is not a simple SELECT, locating and filtering is performed locally, instead of by the SQL server. If the result of the SELECT is a small rowset, local searching will be fast; but if the result is a large rowset, local searching will be slow. For large rowsets, you should use a simple SELECT, or use parameters in the SQL statement and *requery()* as needed instead of relying on the Locate and Filter features.

Master-detail linking through the *masterRowset* and *masterFields* properties with SQL-server-based tables also requires a simple SELECT. An alternative is master-detail linking through Query objects with the *masterSource* property and parameters in the SQL statement. There is no simple SELECT restriction when using Standard tables.

Parameters in an SQL statement are indicated by a colon. For example,

```
select * from CUST where CUST_ID = :cust_id
```

Whenever the SQL property is assigned, it is scanned for parameters. *Visual dBASE* automatically creates corresponding elements in the query's *params* array, with the name of the parameter as the array index. For more information, see the *params* property.

In addition to assigning the SQL statement directly to the *sql* property, you may also use an SQL statement in an external file. To use an external file, place an "@" symbol before the file name in the *sql* property. For example,

```
@ORDERS.SQL
```

The external file must be a text file that contains an SQL statement.

Example The following SQL statement will select all the fields in the table MESSAGES:

```
select * from MESSAGES
```

See also *active*, *executeSQL()*, *params*, SELECT (page 13-13)

state

An enumerated value indicating the rowset's current mode.

Property of Rowset

Description The *state* property is read-only, indicating which mode the rowset is in, as listed in the following table:

| Value | Mode |
|-------|--------|
| 0 | Closed |
| 1 | Browse |
| 2 | Edit |
| 3 | Append |

`tableExists ()`

| Value | Mode |
|-------|--------|
| 4 | Filter |
| 5 | Locate |

- When the rowset’s query is not active, the rowset is Closed.
- While the query is active, the rowset is in Browse mode when it’s not in one of the next four modes.
- The rowset is in Edit mode after a successful *beginEdit()* (implicit or explicit) and it stays in that mode until the row is saved or abandoned.
- After a successful *beginAppend()*, it is in Append mode. It stays in that mode until the new row is saved or abandoned.
- After a *beginFilter()*, it is in Filter mode. It stays in that mode until there is an *applyFilter()* or the Filter mode is abandoned.
- After a *beginLocate()*, it is in Locate mode. It stays in that mode until there is an *applyLocate()* or the Locate mode is abandoned.

Example The following *onClick* event handler for a button labeled “Apply” tests the rowset’s *state* property so that it calls either *applyFilter()* or *applyLocate()*, depending on the rowset’s current mode. It uses manifest constants created with the *#define* preprocessor directive (and available in the *VDBASE.H* include file) to represent the options of the *state* property, which makes the code more readable.

```
#define STATE_CLOSED    0
#define STATE_BROWSE   1
#define STATE_EDIT     2
#define STATE_APPEND   3
#define STATE_FILTER    4
#define STATE_LOCATE   5

:
function applyButton_onServerClick() // Apply Filter or Locate
do case
    case form.rowset.state == STATE_FILTER
        this.form.rowset.applyLocate()
    case form.rowset.state == STATE_LOCATE
        this.form.rowset.applyFilter()
endcase
```

See also *abandon()*, *active*, *applyFilter()*, *applyLocate()*, *autoEdit*, *beginAppend()*, *beginEdit()*, *beginFilter()*, *beginLocate()*, *save()*

tableExists()

Checks to see if a specified table exists in a database.

Syntax `<oRef>.tableExists(<table name expC>)`

<oRef> The database in which to see if the table exists.

<table name expC> The name of the table you want to look for.

Property of Database

Description `tableExists()` returns *true* if a table with the specified name exists in the database. To look for a Standard table, you can always use the default database in the default session by referring to it through the *databases* array property of the *_app* object. For example,

```
_app.databases[ 1 ].tableExists( "Billing" )
```

If you do not specify an extension, *Visual* dBASE will look for both a DBF (dBASE) and DB (Paradox) table with that name.

type [Field]

The data type of the value stored in a field.

Property of Field (including DbfField, PdxField, SqlField)

Description The *type* property reflects the data type stored in the field represented by the Field object.

See also *beforeGetValue*, *value* [Field]

type [Parameter]

An enumerated value indicating the type of parameter.

Property of Parameter

Description The *type* property indicates the type of parameter a Parameter object represents, as listed in the following table:

| Value | Type |
|-------|-------------|
| 0 | Input |
| 1 | Output |
| 2 | InputOutput |
| 3 | Result |

See the Parameter object's *value* property for details on each type.

See also *value* [Parameter]

unidirectional

Specifies whether to assume forward-only navigation to increase performance on SQL-based servers.

Property of Query

Description If *unidirectional* is set to *true*, previously visited rows are not cached and less communication is required between *Visual* dBASE and the SQL server. This results in fewer resources consumed and better performance, but is worthwhile only if you never want to go backward in the rowset.

If *unidirectional* is *true*, you may still be able to go backward, depending on the server, but if so it would be time-consuming.

See also *updateWhere*

unlock()

Releases row and rowset locks.

Syntax <oRef>.unlock()

<oRef> The rowset that contains the lock.

Property of Rowset

Description *unlock()* releases automatic row locks and locks set by *lockRow()* and *lockSet()*. You cannot release locks during a transaction.

Example See the example for *lockRow()* for an example using *unlock()*.

See also *beginTrans()*, *lockRow()*, *lockSet()*

unprepare()

Releases the server resources used by a query or stored procedure.

Syntax This method is called implicitly by the Query or StoredProc object.

Property of Query, StoredProc

- Description** The *unprepare()* method cleans up after a query or stored procedure is deactivated. It is called implicitly when you set the object's *active* property to *false*. In typical usage, you do not call this method directly.
- Advanced applications may override the definition of this method to perform supplementary actions when deactivating the query or stored procedure. Custom data drivers must define this method to perform any necessary actions to clean up when a query or stored procedure is deactivated.
- See also** *active, prepare()*

update

The date and time of the last update made to the row.

Property of LockField

Description Use *update* to determine the date and time the row or table was last updated. The date and time are represented in a string in the following format:

MM/DD/YY HH:MM:SS

This format is accepted by the constructor for a Date object, so you can easily convert the *update* string into an actual date/time.

This property is available only for DBF tables that have been CONVERTed.

Example The following event handler displays the last update date and time in a Text control on a form for the current row, in GMT format.

```
function Rowset_onNavigate
    local dUpdate
    dUpdate = new Date( this.fields[ "_DBASELOCK" ].update ) // Convert to Date
    this.parent.parent.updateText.text := iif( dUpdate.getDay() == 0, ;
        "No update on record", "Last updated on " + dUpdate.toGMTString() )
```

The *day* of an invalid or blank date is zero.

See also CONVERT, *lock, user*

update()

Updates existing rows in one rowset from another.

Syntax <oRef>.update()

<oRef> The UpdateSet object that describes the update.

Property of UpdateSet

Description Use *update()* to update a rowset. You must specify the UpdateSet object's *indexName* property that will be used to match the records. The index must exist for the destination rowset. The original values of all changed records will be copied to the table specified by the UpdateSet object's *changedTableName* property.

To add new rows and update existing rows only, use the *appendUpdate()* method instead.

See also *appendUpdate()*, *destination*, *changedTableName*, *source*

updateWhere

Determines which fields to use in constructing the WHERE clause in an SQL UPDATE statement. SQL-based servers only.

Property of Query

Description *updateWhere* is an enumerated property that may be one of the following values:

| Value | Description |
|-------|-------------------------------|
| 0 | All fields |
| 1 | Key fields |
| 2 | Key fields and changed fields |

See also *unidirectional*

user

The name of the user that last locked or updated the row.

Property of LockField

Description Use *user* to determine the username of the person that currently has a lock when a lock attempt fails, or the name of the user that last had a lock on the row. The maximum length of *user* depends on the size of the _DBASELOCK field specified when the table was CONVERTed.

This property is available only for DBF tables that have been CONVERTed.

Example See *lock*.

See also CONVERT, *lock*, *update*

user()

Returns the login name of the user currently logged in to the session.

Syntax <oRef>.user()

<oRef> The session you want to check.

Property of Session

Description *user()* returns the login name of the user currently logged in to a session on a system that has DBF table security in place. If no DBF table security has been configured, or no one has logged in to the session, *user()* returns an empty string.

See also *access()*, *addPassword()*, *login()*

value [Field]

The value of a field in the row buffer.

Property of Field (including DbfField, PdxField, SqlField)

Description All of the Field objects in the rowset's *fields* array property have a *value* property, which reflects the value of the field in the row buffer, which in turn reflects the values of the fields in the current row.

You may attempt to change the value of a *value* property directly by assignment, in which case the attempt occurs immediately, or through a *dataLinked* control, in which case the attempt occurs when the control loses focus. In either case, the field's *canChange* property fires to see whether the change is allowed. If *canChange* returns *false*, then the assignment doesn't take; if the change was through a *dataLinked* control, the control still contains the proposed new value. If *canChange* returns *true* or there is no *canChange* event handler, the field's value is changed and the *onChange* event fires.

When a field is changed, the rowset's *modified* property is automatically set to *true* to indicate that the rowset has been changed.

By using a field's *beforeGetValue* event, you can make the *value* property appear to be something else besides what is in the row buffer.

Example The following example is the *onClick* event handler for a Reply button in an E-mail viewer. It copies the name from the From field of the original to the To field of the reply and duplicates the Subject field. After setting the *value* properties, the rowset's *modified* property is set to *false* to indicate that these are the default values.

```
function replyButton_onClick()
```

`value [Parameter]`

```
local cTo, cSubject
cTo      = form.rowset.fields[ "From"      ].value
cSubject = form.rowset.fields[ "Subject"   ].value
if form.rowset.beginAppend()
    form.rowset.fields[ "To"               ].value = cTo
    form.rowset.fields[ "Subject"          ].value = cSubject
    form.rowset.modified = false
endif
```

See also *beforeGetValue, canChange, modified, onChange, onGotValue*

value is also a property of the `Parameter` (page 14-126) and many form (page 15-179) classes.

***value* [Parameter]**

The input, output, or result value of a stored procedure.

Property of `Parameter`

Description Values are transmitted to and from stored procedures through `Parameter` objects. Each object's *type* property indicates what type of parameter the object represents. Depending on which one of the four types the parameter is, its *value* property is handled differently.

- **Input:** an input value for the stored procedure. The *value* must be set before the stored procedure is called.
- **Output:** an output value from the stored procedure. The *value* must be set to the correct data type before the stored procedure is called; any dummy value may be used. Calling the stored procedure sets the *value* property to the output value.
- **InputOutput:** both input and output. The *value* must be set before the stored procedure is called. Calling the stored procedure updates the *value* property with the output value.
- **Result:** the result value of the stored procedure. In this case, the stored procedure acts like a function, returning a single result value, instead of updating parameters that are passed to it. Otherwise, the *value* is treated like an output value. The name of the Result parameter is always "Result".

If a `Parameter` object is assigned as the *dataLink* of a component in a form, changes to the component are reflected in the *value* property of the `Parameter` object, and updates to the *value* property of the `Parameter` object are displayed in the component.

Example The following statements call a stored procedure that returns an output parameter. The result is displayed in the Script Pad.

```
d = new Database()
d.databaseName = "IBLOCAL"
```

```
d.active = true
p = new StoredProc()
p.database = d
p.procedureName = "DEPT_BUDGET"
p.params[ "DNO" ].value = "670"           // Set input parameter
p.active = true
? p.params[ "TOT" ].value                 // Display output
```

See also *type*

value is also a property of the Field (page 14-125) and many form (page 15-179) classes.

Form objects

Forms are the primary visual components in *Visual* dBASE applications. You can create forms visually through the Form wizard or Form designer, or programatically by writing code and saving your work as a .WFM file.

Common visual component properties

These properties, events, and methods are common to many visual form components:

| Property | Default | Description |
|----------------------|---------|---|
| <i>before</i> | | The next object in the z-order |
| <i>borderStyle</i> | Default | Specifies whether a box border appears (0=Default, 1=Raised, 2=Lowered, 3=None, 4=Single, 5=Double, 6=Drop Shadow, 7=Client, 8=Modal, 9=Etched In, 10=Etched Out) |
| <i>enabled</i> | true | Whether a component can get focus and operate |
| <i>fontBold</i> | false | Whether the text in a component appears in bold face |
| <i>fontItalic</i> | false | Whether the text in a component appears italicized |
| <i>fontName</i> | Arial | The typeface of the text in a component |
| <i>fontSize</i> | 10 | The point size of the text in a component |
| <i>fontStrikeout</i> | false | Whether the text in a component appears striked-through |
| <i>fontUnderline</i> | false | Whether the text in a component is displayed underlined |
| <i>form</i> | | The form that contains a component |
| <i>height</i> | | Height in the form's current <i>metric</i> units |
| <i>helpFile</i> | | Help file name |

| Property | Default | Description |
|----------------------|---------|--|
| <i>helpId</i> | | Help index topic or context number for context-sensitive help |
| <i>hWnd</i> | | The Windows handle for a component |
| <i>id</i> | -1 | Supplementary control ID number |
| <i>left</i> | 0 | The location of the left edge of a component in the form's current <i>metric</i> units, relative to the left edge of its container |
| <i>mousePointer</i> | 0 | The mouse pointer type when the pointer is over a component |
| <i>name</i> | | The name of a component |
| <i>pageNo</i> | 1 | The page of the form on which a component appears |
| <i>parent</i> | | A component's immediate container (Property discussed in Chapter 5, "Core language.") |
| <i>printable</i> | true | Whether a component is printed when the form is printed |
| <i>speedTip</i> | | Tool tip displayed when pointer hovers over a component |
| <i>statusMessage</i> | | Message displayed in status bar when a component has focus |
| <i>tabStop</i> | true | Whether a component is in the tab sequence |
| <i>top</i> | 0 | The location of the top edge of a component in the form's current <i>metric</i> units, relative to the top edge of its container |
| <i>visible</i> | true | Whether a component is visible |
| <i>width</i> | | Width in the form's current <i>metric</i> units |

| Event | Parameters | Description |
|------------------------|---|--|
| <i>canRender</i> | | Reports only: before a component is rendered; return value determines whether component is rendered. (See page 17-21.) |
| <i>onDesignOpen</i> | <from palette expL> | After a component is first added from the palette and then every time the form is opened in the Form Designer |
| <i>onGotFocus</i> | | After a component gains focus |
| <i>onHelp</i> | | When <i>F1</i> is pressed—overrides context-sensitive help |
| <i>onLeftDbClick</i> | <flags expN>, <column expN>, <row expN> | When the left mouse button is double-clicked |
| <i>onLeftMouseDown</i> | <flags expN>, <column expN>, <row expN> | When the left mouse button is pressed |
| <i>onLeftMouseUp</i> | <flags expN>, <column expN>, <row expN> | When the left mouse button is released |
| <i>onLostFocus</i> | | After a component loses focus |

| Event | Parameters | Description |
|--------------------------|--|---|
| <i>onMiddleDbClick</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the middle mouse button is double-clicked |
| <i>onMiddleMouseDown</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the middle mouse button is pressed |
| <i>onMiddleMouseUp</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the middle mouse button is released |
| <i>onMouseMove</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the is moved over a component |
| <i>onOpen</i> | | After the form containing a component is opened |
| <i>onRender</i> | | Reports only: after a component is rendered. (See page 17-33.) |
| <i>onRightDbClick</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the right mouse button is double-clicked |
| <i>onRightMouseDown</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the right mouse button is pressed |
| <i>onRightMouseUp</i> | <i><flags expN></i> , <i><column expN></i> , <i><row expN></i> | When the right mouse button is released |
| <i>when</i> | <i><form open expL></i> | When attempting to give focus to a component; return value determines whether the component gets focus. |

| Method | Parameters | Description |
|--------------------|---|---|
| <i>move()</i> | <i><left expN></i> <i>[,<top expN></i> <i>[,<width expN></i> <i>[,<height expN>]]]</i> | Repositions and/or resizes a component |
| <i>release()</i> | | Explicitly releases a component from memory. (See page 5-36.) |
| <i>setFocus()</i> | | Sets focus to a component |

class ActiveX

Representation of an ActiveX control.

Syntax [*<oRef>* =] new ActiveX(*<container>* [,*<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created ActiveX object.

<container> The container—typically a Form object—to which you’re binding the ActiveX object.

<name expC> An optional name for the ActiveX object. If not specified, the ActiveX class will auto-generate a name for the object.

Properties The following table lists the properties of interest in the ActiveX class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|---------------------|---------|---|
| <i>anchor</i> | None | How the ActiveX object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>classId</i> | | The ID string that identifies the ActiveX control |
| <i>className</i> | ACTIVEX | Identifies the object as an instance of the ActiveX class (Property discussed in Chapter 5, “Core language.”) |
| <i>description</i> | | A short description of the ActiveX control |
| <i>nativeObject</i> | | The object that contains the ActiveX control’s own properties, events, and methods |

The following table lists the common properties, events, and methods of the ActiveX class:

| Property | | Event | Method |
|--------------------|------------------|-------|-------------------|
| <i>before</i> | <i>pageno</i> | none | <i>move()</i> |
| <i>borderStyle</i> | <i>parent</i> | | <i>release()</i> |
| <i>form</i> | <i>printable</i> | | <i>setFocus()</i> |
| <i>height</i> | <i>top</i> | | |
| <i>left</i> | <i>width</i> | | |
| <i>name</i> | | | |

Description An ActiveX object in *Visual* dBASE is a place holder for an ActiveX control, not an actual ActiveX control.

To include an ActiveX control in a form, create an ActiveX object on the form. Set the *classId* property to the component’s ID string. Once the *classId* is set, the component inherits all the published properties, events, and methods of the ActiveX control, which are accessible through the *nativeObject* property. The object can be used just like a native *Visual* dBASE component.

See also class OLE

class Browse

A data-editing tool that displays multiple records in row-and-column format.

Syntax [*<oRef>* =] new Browse(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Browse object.

<container> The container—typically a Form object—to which you’re binding the Browse object.

<name expC> An optional name for the Browse object. If not specified, the Browse class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the Browse class.

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>alias</i> | | The table that is accessed |
| <i>anchor</i> | None | How the Browse object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>append</i> | true | Whether rows can be added |
| <i>className</i> | BROWSE | Identifies the object as an instance of the Browse class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | WindowText / Window | The color of the highlighted cell |
| <i>colorNormal</i> | WindowText / Window | The color of all other cells |
| <i>cuaTab</i> | true | Whether pressing <i>Tab</i> follows CUA behavior and moves to next control, or moves to next cell |
| <i>fields</i> | | The fields to display, and the options to apply to each field |
| <i>frozenColumn</i> | | The name of the column inside which the cursor is confined. |
| <i>lockedColumns</i> | 0 | The number of columns that remain locked on the left side of the browse grid as it is scrolled horizontally. |
| <i>modify</i> | true | Whether the user can alter data |
| <i>scrollBar</i> | Auto | When a scroll bar appears for the Browse object (0=Off, 1=On, 2=Auto, 3=Disabled) |

| Event | Parameters | Description |
|-------------------|------------|--|
| <i>onAppend</i> | | After a record is added to the table |
| <i>onChange</i> | | After the user changes a value |
| <i>onNavigate</i> | | After the user moves to a different record |

| Method | Parameters | Description |
|--------------------|------------|---|
| <i>copy()</i> | | Copies selected text to the Windows Clipboard |
| <i>cut()</i> | | Cuts selected text and to the Windows Clipboard |
| <i>keyboard()</i> | <expC> | Simulates typed user input to the Browse object |

| Method | Parameters | Description |
|----------------|------------|---|
| <i>paste()</i> | | Copies text from the Windows clipboard to the current cursor position |
| <i>undo()</i> | | Reverses the effects of the most recent <i>cut()</i> , <i>copy()</i> , or <i>paste()</i> action |

The following table lists the common properties, events, and methods of the Browse class:

| Property | | Event | | Method |
|--------------------|----------------------|-------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDblClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDblClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>form</i> | <i>statusMessage</i> | <i>onMiddleDblClick</i> | <i>when</i> | |
| <i>height</i> | <i>tabStop</i> | | | |
| <i>helpFile</i> | <i>top</i> | | | |
| <i>helpId</i> | <i>visible</i> | | | |
| <i>hWnd</i> | <i>width</i> | | | |

Description The Browse object is maintained for compatibility and is suitable only for viewing and editing tables open in work areas. For forms that use data access objects, use a Grid object instead.

Two properties specify which table is displayed in the Browse object.

- The *view* property of the parent form
- The *alias* property of the browse object

You can specify individual fields to display with the *fields* property. For example, if the browse object's form is based on a query, you use *fields* to display fields from any of the query's tables. (You must specify a file with *alias* before you can use *fields*.)

See Also class Grid

class CheckBox

A check box on a form.

Syntax [*<oRef>* =] new CheckBox(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created CheckBox object.

<container> The container—typically a Form object—to which you're binding the CheckBox object.

<name expC> An optional name for the CheckBox object. If not specified, the CheckBox class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the CheckBox class. (No particular methods are associated with this class.)

| Property | Default | Description |
|--------------------|-----------------|--|
| <i>className</i> | CHECKBOX | Identifies the object as an instance of the CheckBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/BtnFace | The color of the checkbox’s text label |
| <i>dataLink</i> | | The Field object that is linked to the CheckBox |
| <i>group</i> | | The group to which the check box belongs |
| <i>text</i> | <same as name> | The text label that appears beside the check box |
| <i>textLeft</i> | false | Whether the check box’s text label appears to the left or to the right of the check box |
| <i>transparent</i> | false | Whether the CheckBox object has the same background color or image as its container |
| <i>value</i> | | The current value of the check box (<i>true</i> or <i>false</i>) |

| Event | Parameters | Description |
|-----------------|------------|--------------------------------|
| <i>onChange</i> | | After the check box is toggled |

The following table lists the common properties, events, and methods of the CheckBox class:

| Property | | Event | | Method |
|----------------------|----------------------|-------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDblClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDblClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDblClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Use a CheckBox component to represent a *true/false* value.

See also class RadioButton

class ColumnCheckBox

A checkbox in a grid column.

Syntax These controls are created by assigning the appropriate *editorType* to the GridColumn object.

Properties The following tables list the properties of interest in the ColumnCheckBox class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>className</i> | COLUMNCHECKBOX | Identifies the object as an instance of the ColumnCheckBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | | The color of the cell containing the ColumnCheckBox object when the cell has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the cell containing the ColumnCheckBox object when the cell does not have focus |
| <i>value</i> | | The current value of the check box (<i>true</i> or <i>false</i>) |

The following table lists the common properties, events, and methods of the ColumnCheckBox class:

| Property | | Event | Method |
|-------------------|---|---|--------|
| <i>hWndparent</i> | <i>speedTip</i> <i>statusMessage</i> | <i>onGotFocus</i> <i>onLostFocus</i> | none |

Description A ColumnCheckBox is a simplified CheckBox control in a grid column. When the enumerated *editorType* property of a GridColumn control is set to CheckBox, the column uses a ColumnCheckBox control, which is accessible through the GridColumn object’s *editorControl* property.

The box around the checkmark is displayed only for the cell that has focus. For the other cells in the column that do not have focus, there is only a checkmark if the *value* is *true*; or nothing if the *value* is *false*—the cell appears empty.

As with all column controls, the *dataLink* and *width* for the control is in the parent GridColumn object, not the control itself. The *height* is controlled by the *cellHeight* of the grid.

See also class CheckBox, class ColumnHeadingControl, class GridColumn, *editorControl*, *editorType*

class ColumnComboBox

A combobox in a grid column.

Syntax These controls are created by assigning the appropriate *editorType* to the GridColumn object.

Properties The following tables list the properties of interest in the ColumnComboBox class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>className</i> | COLUMNCOMBOBOX | Identifies the object as an instance of the ColumnComboBox class (Property discussed in Chapter 5, "Core language.") |
| <i>colorHighlight</i> | | The color of the text in the ColumnComboBox object when the object has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the ColumnComboBox object when the object does not have focus |
| <i>dataSource</i> | | The option strings of the ColumnComboBox object |
| <i>dropDownHeight</i> | | The number of options displayed in the drop-down list |
| <i>dropDownWidth</i> | | The width of the drop-down list in the form's current metric units |
| <i>function</i> | | A text formatting function |
| <i>picture</i> | | Formatting template |
| <i>sorted</i> | false | Whether the options are sorted |
| <i>value</i> | | The value currently displayed in the ColumnComboBox object |

The following table lists the common properties, events, and methods of the ColumnComboBox class:

| Property | Event | Method |
|----------------------|-------------------------|--------------------|
| <i>borderStyle</i> | <i>hWndmousePointer</i> | <i>onGotFocus</i> |
| <i>fontBold</i> | | <i>onLostFocus</i> |
| <i>fontItalic</i> | <i>parent</i> | |
| <i>fontName</i> | <i>speedTip</i> | |
| <i>fontSize</i> | <i>statusMessage</i> | |
| <i>fontStrikeout</i> | | |
| <i>fontUnderline</i> | | |

Description A ColumnComboBox is a simplified ComboBox control in a grid column. The combobox is always the DropDownList style. When the enumerated *editorType* property of a GridColumn control is set to ComboBox, the column uses a ColumnComboBox control, which is accessible through the GridColumn object's *editorControl* property.

Only the cell that has focus appears as a combobox. All other cells in the column which do not have focus appear as ColumnEntryfield controls instead, with no drop-down control.

As with all column controls, the *dataLink* and *width* for the control is in the parent GridColumn object, not the control itself. The *height* is controlled by the *cellHeight* of the grid.

See also class ColumnHeadingControl, class ComboBox, class GridColumn, editorControl, editorType

class ColumnEntryfield

A single-line text input field in a grid column.

Syntax These controls are created by assigning the appropriate *editorType* to the GridColumn object.

Properties The following tables list the properties and events of interest in the ColumnEntryfield class. (No methods are associated with this class.)

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>className</i> | COLUMNENTRYFIELD | Identifies the object as an instance of the ColumnEntryfield class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | | The color of the text in the ColumnEntryfield object when the object has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the ColumnEntryfield object when the object does not have focus |
| <i>function</i> | | A text formatting function |
| <i>memoEditor</i> | | The memo editor control used when editing a memo field |
| <i>picture</i> | | Formatting template |
| <i>validErrorMsg</i> | Invalid input | The message that is displayed when the <i>valid</i> event fails |
| <i>value</i> | | The value currently displayed in the ColumnEntryfield object |

| Event | Parameters | Description |
|--------------|--|--|
| <i>key</i> | <char expN>, <position expN>, <shift expL>, <ctrl expL> | When a key is pressed. Return value may change or cancel keystroke. |
| <i>valid</i> | | When attempting to remove focus. Must return <i>true</i> , or focus remains. |

The following table lists the common properties, events, and methods of the ColumnEntryfield class:

| Property | Event | Method |
|----------------------|----------------------|--------------------|
| <i>borderStyle</i> | <i>hWndparent</i> | <i>onGotFocus</i> |
| <i>fontBold</i> | <i>speedTip</i> | <i>onLostFocus</i> |
| <i>fontItalic</i> | <i>statusMessage</i> | |
| <i>fontName</i> | | |
| <i>fontSize</i> | | |
| <i>fontStrikeout</i> | | |
| <i>fontUnderline</i> | | |

Description A ColumnEntryfield is a simplified Entryfield control in a grid column. When the enumerated *editorType* property of a GridColumn control is set to Entryfield, the column uses a ColumnEntryfield control, which is accessible through the GridColumn object's *editorControl* property.

As with all column controls, the *dataLink* and *width* for the control is in the parent GridColumn object, not the control itself. The *height* is controlled by the *cellHeight* of the grid.

See also class ColumnHeadingControl, class Entryfield, class GridColumn, *editorControl*, *editorType*

class ColumnHeadingControl

A grid column heading.

Syntax These controls are created for each GridColumn object.

Properties The following tables list the properties of interest in the ColumnHeadingControl class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|--------------------|-----------------------|--|
| <i>className</i> | COLUMNHEADINGCONTROL | Identifies the object as an instance of the ColumnHeadingControl class (Property discussed in Chapter 5, "Core language.") |
| <i>colorNormal</i> | WindowText /Window | The color of the control and its text |
| <i>function</i> | | A text formatting function |
| <i>picture</i> | | Formatting template |
| <i>value</i> | | The text displayed in the ColumnHeadingControl object |

The following table lists the common properties, events, and methods of the ColumnHeadingControl class:

| Property | | Event | | Method |
|----------------------|-------------------|-------------------------|--------------------------|--------|
| <i>fontBold</i> | <i>hWndparent</i> | <i>onLeftDblClick</i> | <i>onMiddleMouseDown</i> | none |
| <i>fontItalic</i> | | <i>onLeftMouseDown</i> | <i>onMiddleMouseUp</i> | |
| <i>fontName</i> | | <i>onLeftMouseUp</i> | <i>onRightDblClick</i> | |
| <i>fontSize</i> | | <i>onMiddleDblClick</i> | <i>onRightMouseDown</i> | |
| <i>fontStrikeout</i> | | | <i>onRightMouseUp</i> | |
| <i>fontUnderline</i> | | | | |

Description Each column in a grid has a ColumnHeadingControl object that represents the column heading. It is accessible through the GridColumn object's *headingControl* property.

As with all column controls, the *width* for the control is in the parent GridColumn object, not the control itself. The *height* is controlled by the *cellHeight* of the grid.

See also class GridColumn, *headingControl*

class ColumnSpinBox

An entryfield with a spinner for entering numeric or date values in a grid column.

Syntax These controls are created by assigning the appropriate *editorType* to the GridColumn object.

Properties The following tables list the properties and events of interest in the ColumnSpinBox class. (No methods are associated with this class.)

| Property | Default | Description |
|-----------------------|---------------------|---|
| <i>className</i> | COLUMNSPINBOX | Identifies the object as an instance of the ColumnSpinBox class (Property discussed in Chapter 5, "Core language.") |
| <i>colorHighlight</i> | | The color of the text in the ColumnSpinBox object when the object has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the ColumnSpinBox object when the object does not have focus |
| <i>function</i> | | A text formatting function |
| <i>picture</i> | | Formatting template |
| <i>rangeMax</i> | | The maximum value |
| <i>rangeMin</i> | | The minimum value |
| <i>rangeRequired</i> | false | Whether the range values are enforced even when no change has been made |
| <i>step</i> | 1 | The value added or subtracted when using the spinner |

| Property | Default | Description |
|----------------------|---------------|---|
| <i>validErrorMsg</i> | Invalid input | The message that is displayed when the <i>valid</i> event fails |
| <i>value</i> | | The value currently displayed in the ColumnSpinBox object |

| Event | Parameters | Description |
|--------------|------------|--|
| <i>valid</i> | | When attempting to remove focus. Must return <i>true</i> , or focus remains. |

The following table lists the common properties, events, and methods of the ColumnSpinBox class:

| Property | | Event | Method |
|----------------------|----------------------|--------------------|--------|
| <i>borderStyle</i> | <i>hWndparent</i> | <i>onGotFocus</i> | none |
| <i>fontBold</i> | <i>speedTip</i> | <i>onLostFocus</i> | |
| <i>fontItalic</i> | <i>statusMessage</i> | | |
| <i>fontName</i> | | | |
| <i>fontSize</i> | | | |
| <i>fontStrikeout</i> | | | |
| <i>fontUnderline</i> | | | |

Description A ColumnSpinBox is a simplified SpinBox control in a grid column. When the enumerated *editorType* property of a GridColumn control is set to SpinBox, the column uses a ColumnSpinBox control, which is accessible through the GridColumn object's *editorControl* property.

Only the cell that has focus appears as a spinbox. All other cells in the column which do not have focus appear as ColumnEntryfield controls instead, with no spinner control.

As with all column controls, the *dataLink* and *width* for the control is in the parent GridColumn object, not the control itself. The *height* is controlled by the *cellHeight* of the grid.

See also class ColumnHeadingControl, class SpinBox, class GridColumn, *editorControl*, *editorType*

class ComboBox

A component on a form which can be temporarily expanded to show a list from which you can pick a single item.

Syntax [*<oRef>* =] new ComboBox(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created ComboBox object.

<container> The container—typically a Form object—to which you’re binding the ComboBox object.

<name expC> An optional name for the ComboBox object. If not specified, the ComboBox class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of the ComboBox class.

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>autoDrop</i> | false | Whether the drop-down list automatically drops down when the combobox gets focus |
| <i>className</i> | COMBOBOX | Identifies the object as an instance of the ComboBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the ComboBox object |
| <i>dataLink</i> | | The Field object that is linked to the ComboBox object |
| <i>dataSource</i> | | The option strings of the ComboBox object |
| <i>dropDownHeight</i> | | The number of options displayed in the drop-down list |
| <i>dropDownWidth</i> | | The width of the drop-down list in the form’s current metric units |
| <i>sorted</i> | false | Whether the options are sorted |
| <i>style</i> | DropDown | The style of the ComboBox: 0=Simple, 1=DropDown, 2=DropDownList |
| <i>value</i> | | The value of the currently selected option |

| Event | Parameters | Description |
|-----------------|--|--|
| <i>key</i> | <char expN>, <position expN>, <shift expL>, <ctrl expL> | When a key is pressed. Return value may change or cancel keystroke. |
| <i>onChange</i> | | After the selection has changed and the ComboBox object loses focus, but before <i>onLostFocus</i> |

| Method | Parameters | Description |
|-------------------|------------|---|
| <i>copy()</i> | | Copies selected text to the Windows clipboard |
| <i>cut()</i> | | Cuts selected text and to the Windows clipboard |
| <i>keyboard()</i> | <expC> | Simulates typed user input to the ComboBox object |
| <i>paste()</i> | | Copies text from the Windows clipboard to the current cursor position |
| <i>undo()</i> | | Reverses the effects of the most recent <i>cut()</i> , <i>copy()</i> , or <i>paste()</i> action |

The following table lists the common properties, events, and methods of the ComboBox class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDbClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Use a ComboBox object when you want the user to pick one item from a list. When the user is not choosing an item, the list is not visible. The list of options is set with the *dataSource* property.

If a ComboBox is *dataLinked* to a field object that has implemented its *lookupSQL* or *lookupRowset* properties, the combobox will automatically be populated with the appropriate lookup values, and store the corresponding key values in the *dataLinked* field.

See also class Entryfield, class ListBox, *lookupRowset* (page 14-86), *lookupSQL* (page 14-87)

class Container

A container for other controls.

Syntax [*<oRef>* =] new Container(*<container>* [,*<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Container object.

<container> The container—typically a Form object—to which you're binding the Container object.

<name expC> An optional name for the Container object. If not specified, the Container class will auto-generate a name for the object.

Properties The following table lists the properties of interest in the Container class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|--------------------|-----------|---|
| <i>anchor</i> | None | How the Container object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>className</i> | CONTAINER | Identifies the object as an instance of the Container class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnFace | The background color |
| <i>expandable</i> | true | Reports only: whether the container expands to show all its components |
| <i>transparent</i> | false | Whether the container has the same background color or image as the its own container (usually the form) |

The following table lists the common properties, events, and methods of the Container class:

| Property | | Event | | Method |
|---------------------|------------------|------------------------|--------------------------|-------------------|
| <i>before</i> | <i>name</i> | <i>canRender</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onDesignOpen</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>first</i> | <i>parent</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> | |
| <i>form</i> | <i>printable</i> | <i>onLeftMouseDown</i> | <i>onRender</i> | |
| <i>height</i> | <i>top</i> | <i>onLeftMouseUp</i> | <i>onRightDbClick</i> | |
| <i>hWnd</i> | <i>visible</i> | <i>onMiddleDbClick</i> | <i>onRightMouseDown</i> | |
| <i>left</i> | <i>width</i> | | <i>onRightMouseUp</i> | |
| <i>mousePointer</i> | | | | |

Description Use the Container object to create groups of controls, a custom control that contain multiple controls, or to otherwise group controls in a form. When a control is dropped in a Container object, it becomes a child object of the Container object. Its *parent* property references the container, while its *form* property references the form.

To make the rectangle that contains the controls invisible, set the *borderStyle* property to None (3) and the *transparent* property to *true*.

See also class Notebook

class Editor

A multiple-line text input field on a form.

Syntax [*<oRef>* =] new Editor(<container> [,<name expC>])

<oRef> A variable or property—typically of <container>—in which to store a reference to the newly created Editor object.

<container> The container—typically a Form object—to which you’re binding the Editor object.

<name expC> An optional name for the Editor object. If not specified, the Editor class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the Editor class.

| Property | Default | Description |
|-------------------------|---------------------|--|
| <i>anchor</i> | None | How the Editor object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>border</i> | true | Whether the Editor object is surrounded by the border specified by <i>borderStyle</i> |
| <i>className</i> | EDITOR | Identifies the object as an instance of the Editor class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the Editor object |
| <i>cuaTab</i> | true | Whether pressing <i>Tab</i> follows CUA behavior and moves to next control, or inserts tab in text |
| <i>dataLink</i> | | The Field object that is linked to the Editor object |
| <i>evalTags</i> | true | Whether to evaluate any HTML formatting tags in the text or display them as-is |
| <i>lineNo</i> | 1 | The current line number in the editor |
| <i>marginHorizontal</i> | | The horizontal margin between the text and its rectangular frame. (See page 17-30.) |
| <i>marginVertical</i> | | The vertical margin between the text and its rectangular frame. (See page 17-31.) |
| <i>modify</i> | true | Whether the text is editable or not |
| <i>popupEnable</i> | true | Whether the Editor object’s context menu is available |
| <i>scrollBar</i> | Auto | When a scroll bar appears for the Editor object (0=Off, 1=On, 2=Auto, 3=Disabled) |
| <i>value</i> | | The string currently displayed in the Editor object |
| <i>wrap</i> | true | Whether to word-wrap the text in the editor |

| Event | Parameters | Description |
|-----------------|------------|--|
| <i>onChange</i> | | After the string in the Editor object has changed and the Editor object loses focus, but before <i>onLostFocus</i> |
| <i>valid</i> | | When attempting to remove focus. Must return <i>true</i> , or focus remains. |

| Method | Parameters | Description |
|--------------------|------------|---|
| <i>copy()</i> | | Copies selected text to the Windows clipboard |
| <i>cut()</i> | | Cuts selected text and to the Windows clipboard |
| <i>keyboard()</i> | <expC> | Simulates typed user input to the Editor object |

| Method | Parameters | Description |
|----------------------|------------|---|
| <code>paste()</code> | | Copies text from the Windows clipboard to the current cursor position |
| <code>undo()</code> | | Reverses the effects of the most recent <code>cut()</code> , <code>copy()</code> , or <code>paste()</code> action |

The following table lists the common properties, events, and methods of the Editor class:

| Property | | Event | | Method |
|----------------------------|----------------------------|------------------------------|--------------------------------|-------------------------|
| <code>before</code> | <code>id</code> | <code>onDesignOpen</code> | <code>onMiddleMouseDown</code> | <code>move()</code> |
| <code>borderStyle</code> | <code>left</code> | <code>onGotFocus</code> | <code>onMiddleMouseUp</code> | <code>release()</code> |
| <code>enabled</code> | <code>mousePointer</code> | <code>onHelp</code> | <code>onMouseMove</code> | <code>setFocus()</code> |
| <code>fontBold</code> | <code>name</code> | <code>onLeftDbClick</code> | <code>onOpen</code> | |
| <code>fontItalic</code> | <code>pageno</code> | <code>onLeftMouseDown</code> | <code>onRightDbClick</code> | |
| <code>fontName</code> | <code>parent</code> | <code>onLeftMouseUp</code> | <code>onRightMouseDown</code> | |
| <code>fontSize</code> | <code>printable</code> | <code>onLostFocus</code> | <code>onRightMouseUp</code> | |
| <code>fontStrikeout</code> | <code>speedTip</code> | <code>onMiddleDbClick</code> | <code>when</code> | |
| <code>fontUnderline</code> | <code>statusMessage</code> | | | |
| <code>form</code> | <code>tabStop</code> | | | |
| <code>height</code> | <code>top</code> | | | |
| <code>helpFile</code> | <code>visible</code> | | | |
| <code>helpId</code> | <code>width</code> | | | |
| <code>hWnd</code> | | | | |

Description Use an Editor component to display and edit multi-line text. To display the text but not allow changes, set the *modify* property to *false*. The Editor component understands and displays basic HTML formatting tags. It has a context menu that is accessible by right-clicking the editor (unless its *popupEnable* property is *false*). The context menu lets you find and replace text, toggle word wrapping and HTML formatting, and show or hide the Format toolbar.

See also class Entryfield, REPLACE MEMO, `replaceFromFile()`

class Entryfield

A single-line text input field on a form.

Syntax [`<oRef> =`] `new Entryfield(<container> [, <name expC>])`

<oRef> A variable or property—typically of `<container>`—in which to store a reference to the newly created Entryfield object.

<container> The container—typically a Form object—to which you're binding the Entryfield object.

<name expC> An optional name for the Entryfield object. If not specified, the Entryfield class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the Entryfield class.

| Property | Default | Description |
|-----------------------|---------------------|--|
| <i>border</i> | true | Whether the Entryfield object is surrounded by the border specified by <i>borderStyle</i> |
| <i>className</i> | ENTRYFIELD | Identifies the object as an instance of the Entryfield class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | | The color of the text in the Entryfield object when the object has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the Entryfield object when the object does not have focus |
| <i>dataLink</i> | | The Field object that is linked to the Entryfield object |
| <i>function</i> | | A text formatting function |
| <i>maxLength</i> | | The maximum length of the text in the Entryfield object |
| <i>memoEditor</i> | | The memo editor control used when editing a memo field |
| <i>phoneticLink</i> | | The control that mirrors the phonetic equivalent of the current value |
| <i>picture</i> | | Formatting template |
| <i>selectAll</i> | true | Whether the entryfield contents are initially selected when the Entryfield object gets focus |
| <i>validErrorMsg</i> | Invalid input | The message that is displayed when the <i>valid</i> event fails |
| <i>validRequired</i> | false | Whether to fire the <i>valid</i> event even when no change has been made |
| <i>value</i> | | The value currently displayed in the Entryfield object |

| Event | Parameters | Description |
|-----------------|--|--|
| <i>key</i> | <char expN>, <position expN>, <shift expL>, <ctrl expL> | When a key is pressed. Return value may change or cancel keystroke. |
| <i>onChange</i> | | After the string in the Entryfield object has changed and the Entryfield object loses focus, but before <i>onLostFocus</i> |
| <i>valid</i> | | When attempting to remove focus. Must return <i>true</i> , or focus remains. |

| Method | Parameters | Description |
|-------------------|------------|---|
| <i>copy()</i> | | Copies selected text to the Windows clipboard |
| <i>cut()</i> | | Cuts selected text and to the Windows clipboard |
| <i>keyboard()</i> | <expC> | Simulates typed user input to the Entryfield object |
| <i>paste()</i> | | Copies text from the Windows clipboard to the current cursor position |

| Method | Parameters | Description |
|-------------------------|------------|---|
| <i>showMemoEditor()</i> | | Opens the specified <i>memoEditor</i> |
| <i>undo()</i> | | Reverses the effects of the most recent <i>cut()</i> , <i>copy()</i> , or <i>paste()</i> action |

The following table lists the common properties, events, and methods of the Entryfield class:

| Property | | Event | | Method |
|----------------------|----------------------|-------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDblClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDblClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDblClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Entryfield objects are the primary data display and entry component.

See also class ComboBox, class Editor, class SpinBox

class Form

A Form object.

Syntax [*<oRef>* =] new Form([*<title expC>*])

<oRef> A variable or property in which to store a reference to the newly created Form object.

<title expC> An optional title for the Form object. If not specified, the title will be "Form".

Properties The following tables list the properties, events, and methods of the Form class.

| Property | Default | Description |
|----------------------|---------|---|
| <i>activeControl</i> | | The currently active control |
| <i>autoCenter</i> | false | Whether the form automatically centers on-screen when it is opened |
| <i>autoSize</i> | false | Whether the form automatically sizes itself to display all its components |

| Property | Default | Description |
|----------------------|---------|--|
| <i>background</i> | | Background image |
| <i>className</i> | FORM | Identifies the object as an instance of the Form class (Property discussed in Chapter 5, “Core language.”) |
| <i>clientEdge</i> | false | Whether the edge of the form has the sunken client appearance |
| <i>colorNormal</i> | BtnFace | Background color |
| <i>designView</i> | | A view that is used when designing the form |
| <i>elements</i> | | An array containing object references to the components on the form |
| <i>escExit</i> | true | Whether pressing <i>Esc</i> closes the form |
| <i>first</i> | | The first component on the form in the z-order |
| <i>hWndClient</i> | | The Windows handle for the form's client area |
| <i>icon</i> | | An icon file or resources that displays when the form is minimized |
| <i>inDesign</i> | | Whether the form was instantiated by the Form designer |
| <i>maximize</i> | true | Whether the form can be maximized when not MDI |
| <i>mdi</i> | true | Whether the form is MDI or SDI |
| <i>menuFile</i> | | The name of the form's .MNU menu file |
| <i>metric</i> | Chars | Units of measurement (0=Chars, 1=Twips, 2=Points, 3=Inches, 4=Centimeters, 5=Millimeters, 6=Pixels) |
| <i>minimize</i> | true | Whether the form can be minimized when not MDI |
| <i>moveable</i> | true | Whether the form is moveable when not MDI |
| <i>nextObj</i> | | The object that's about to receive focus |
| <i>popupMenu</i> | | The form's Popup menu object |
| <i>refreshAlways</i> | true | Whether to refresh the form after all form-based navigation and updates |
| <i>rowset</i> | | The primary rowset |
| <i>scaleFontBold</i> | false | Whether the base font used for the Chars <i>metric</i> is boldface |
| <i>scaleFontName</i> | Arial | The base font used for the Chars <i>metric</i> |
| <i>scaleFontSize</i> | 10 | The point size of the base font used for the Chars <i>metric</i> |
| <i>scrollBar</i> | Off | When a scroll bar appears for the form (0=Off, 1=On, 2=Auto, 3=Disabled) |
| <i>showSpeedTip</i> | true | Whether to show tool tips |
| <i>sizeable</i> | true | Whether the form is resizeable when not MDI |
| <i>smallTitle</i> | false | Whether the form has the smaller palette-style title bar when not MDI |
| <i>sysMenu</i> | true | Whether the form's system menu icon and close icon are displayed when not MDI |
| <i>text</i> | | The text that appears in the form's title bar |
| <i>topMost</i> | false | Whether the form stays on top when not MDI |
| <i>useTablePopup</i> | false | Whether to use the default table navigation popup when no popup is assigned as the form's <i>popupMenu</i> . |

| Property | Default | Description |
|--------------------|---------|--|
| <i>view</i> | | The query or table on which the form is based |
| <i>windowState</i> | Normal | The state of the window (0=Normal, 1=Minimized, 2=Maximized) |

| Event | Parameters | Description |
|--------------------|-------------------|---|
| <i>canClose</i> | | When attempting to close form; return value allows or disallows closure |
| <i>canNavigate</i> | <workarea expN> | When attempting to navigate in work area; return value allows or disallows leaving current record |
| <i>onAppend</i> | | After a new record is added |
| <i>onChange</i> | <workarea expN> | After leaving a record that was changed, before <i>onNavigate</i> |
| <i>onClose</i> | | After the form has been closed |
| <i>onMove</i> | | After the form has been moved |
| <i>onNavigate</i> | <workarea expN> | After navigation in a work area |
| <i>onSelection</i> | <control ID expN> | After the form is submitted |
| <i>onSize</i> | <expN> | After the form is resized or changes <i>windowState</i> |

| Method | Parameters | Description |
|--------------------------|------------|--|
| <i>abandonRecord()</i> | | Abandons changes to the current record |
| <i>beginAppend()</i> | | Starts append of new record |
| <i>close()</i> | | Closes the form |
| <i>isRecordChanged()</i> | | Checks whether the current record buffer has changed |
| <i>open()</i> | | Loads and opens the form |
| <i>pageCount()</i> | | Returns the highest <i>pageno</i> of any component |
| <i>print()</i> | | Prints the form |
| <i>readModal()</i> | | Opens the form modally |
| <i>refresh()</i> | | Redraws the form |
| <i>saveRecord()</i> | | Saves changes to the current or new record |
| <i>showFormatBar()</i> | <expL> | Displays or hides the formatting toolbar |

The following table lists the common properties, events, and methods of the Form class:

| Property | | Event | | Method |
|-----------------|----------------------|------------------------|--------------------------|-------------------|
| <i>enabled</i> | <i>mousePointer</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>height</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>helpFile</i> | <i>statusMessage</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>helpId</i> | <i>top</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>hWnd</i> | <i>visible</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>left</i> | <i>width</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| | | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| | | <i>onMiddleDbClick</i> | | |

Description A Form object acts as a container for other visual components (also known as controls) and the data access objects that are linked to them. Consequently, releasing a form object from memory automatically releases the objects it contains.

An object reference to all the visual components in a form is stored in its *elements* array. All of the visual components have a *form* property that points back to the form.

The form has a *rowset* property that refers to its primary rowset. Components can access this rowset in their event handlers generically with the object reference *form.rowset*. For example, a button on a form that goes to the first row in the rowset would have an *onClick* event handler like this:

```
function firstButton_onClick()
    form.rowset.first()
```

If the form has more than one rowset, each one can be addressed through the *rowset* property of the Query objects, which are properties of the form. For example, to go to the last row in the rowset of the Query object *members1*, the *onClick* event handler would look like this:

```
function lastMemberButton_onClick()
    form.members1.rowset.last()
```

A form can consist of more than one page. One way to implement multi-page forms is to use the *pageno* property of controls to determine on which page they appear, and use a TabBox control to let users easily switch between pages. You may also use a NoteBook control to create a multi-page container in a form.

You can create two types of forms: *modal* and *modeless*. A modal form halts execution of the routine that opened it until the form is closed. When active, it takes control of the user interface; users can't switch to another window in the same application without exiting the form. A dialog box is an example of a modal form; when it is opened, program execution stops and focus can't be given to another window until the user closes the dialog box.

In contrast a modeless form window allows users to freely switch to other windows in an application. Most forms that you create for an application will be modeless. A modeless form window conforms to the Multiple Document Interface (MDI) protocol, which lets you open multiple document windows within an application window.

To create and use a modeless form, set the *mdi* property to *true* and open the form with the *open()* method. To create and use a modal form, set *mdi* to *false* and open the form with the *readModal()* method.

You can also create SDI (Single Document Interface) windows that appear like application windows. To do so, set the *mdi* property to *false* and use *SHELL(false)*. *SHELL(false)* hides the standard Visual dBASE environment and lets your form take over the user interface. The Visual dBASE application window disappears, and the form name appears in the Windows Task List.

See also class NoteBook, class TabBox

class Grid

A grid of other controls.

Syntax [*<oRef>* =] new Grid(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Grid object.

<container> The container—typically a Form object—to which you’re binding the Grid object.

<name expC> An optional name for the Grid object. If not specified, the Grid class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the Grid class.

| Property | Default | Description |
|--------------------------|---------|--|
| <i>allowAddRows</i> | true | Whether navigating down past the last row automatically calls <i>beginAppend()</i> |
| <i>allowColumnMoving</i> | true | Whether columns may be moved with the mouse |
| <i>allowColumnSizing</i> | true | Whether columns may be sized with the mouse |
| <i>allowEditing</i> | true | Whether editing is allowed or the grid is read-only |
| <i>allowRowSizing</i> | true | Whether rows may be sized with the mouse |
| <i>anchor</i> | None | How the Grid object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>bgColor</i> | white | Background color |
| <i>cellHeight</i> | | The height of each cell |
| <i>className</i> | GRID | Identifies the object as an instance of the Grid class (Property discussed in Chapter 5, “Core language.”) |
| <i>columnCount</i> | | The number of columns in the grid |
| <i>columns</i> | | An array of objects for each column in the grid |
| <i>cuaTab</i> | false | Whether pressing <i>Tab</i> follow CUA behavior and moves to next control, or moves to next cell |
| <i>currentColumn</i> | | The number of the column that has focus in the grid |
| <i>dataLink</i> | | The Rowset object that is linked to the grid |
| <i>dragScrollRate</i> | 300 | The delay time, in milliseconds, between each column scroll when dragging columns |
| <i>frozenColumn</i> | | The name of the column inside which the cursor is confined. |
| <i>gridLineWidth</i> | 1 | Width of grid lines in pixels (0=no grid lines) |
| <i>hasColumnHeadings</i> | true | Whether column headings are displayed |

| Property | Default | Description |
|---------------------------|---------|---|
| <i>hasColumnLines</i> | true | Whether column (vertical) grid lines are displayed |
| <i>hasIndicator</i> | true | Whether the indicator column is displayed |
| <i>hasRowLines</i> | true | Whether row (horizontal) grid lines are displayed |
| <i>hasVScrollHintText</i> | true | Whether the relative row count is displayed as the grid is scrolled vertically |
| <i>hScrollBar</i> | Auto | When a horizontal scrollbar appears (0=Off, 1=On, 2=Auto, 3=Disabled) |
| <i>integralHeight</i> | false | Whether a partial row at the bottom of the grid is displayed |
| <i>lockedColumns</i> | 0 | The number of columns that remain locked on the left side of the grid as it is scrolled horizontally. |
| <i>multiSelect</i> | false | Whether multiple rows may be visually selected |
| <i>rowSelect</i> | false | Whether the entire row is visually selected |
| <i>vScrollBar</i> | Auto | When a vertical scrollbar appears (0=Off, 1=On, 2=Auto, 3=Disabled) |

| Event | Parameters | Description |
|--------------------|------------|---|
| <i>onFormSize</i> | | After the form containing the grid is resized |
| <i>onSelChange</i> | | After moving to another row in the grid |

| Method | Parameters | Description |
|-------------------|------------|---|
| <i>refresh()</i> | | Repaints the grid |
| <i>selected()</i> | | Returns an array of bookmarks for the currently selected rows in the grid |

The following table lists the common properties, events, and methods of the Grid class:

| Property | | Event | Method |
|--------------------|------------------|------------------------|---|
| <i>before</i> | <i>name</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> <i>release()</i> |
| <i>enabled</i> | <i>parent</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> <i>setFocus()</i> |
| <i>form</i> | <i>printable</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> |
| <i>height</i> | <i>speedTip</i> | <i>onLeftMouseUp</i> | <i>onRightDbClick</i> |
| <i>helpFile</i> | <i>tabStop</i> | <i>onLostFocus</i> | <i>onRightMouseDown</i> |
| <i>helpId</i> | <i>top</i> | <i>onMiddleDbClick</i> | <i>onRightMouseUp</i> |
| <i>hWnd</i> | <i>visible</i> | | |
| <i>id</i> | <i>width</i> | | |
| <i>left</i> | | | |

Description The Grid object is a multi-column grid control for displaying the contents of a rowset. The *dataLink* property is set to the rowset. Columns are automatically created for each field in the rowset.

Each column is represented by a GridColumn object. If the default columns are used, these objects are hidden, and all fields are displayed. By explicitly creating a GridColumn object for each column as an element in the grid's *columns* array, you may control the fields that are displayed and assign different kinds of controls in different columns.

Navigation in the rowset updates any grids that are *dataLinked* to the rowset, and vice versa. When you explicitly create GridColumn objects, you may set their *dataLink* properties to fields in other rowsets, like the fields in a linked detail table.

See Also class GridColumn

class GridColumn

A column in a grid.

Syntax [*<oRef>* =] new GridColumn(*<grid>*)

<oRef> A variable or property—typically an array element of the *<grid>* object's *columns* array—in which to store a reference to the newly created GridColumn object.

<grid> The Grid object that contains the GridColumn object.

Properties The following tables list the properties of interest of the GridColumn class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|-----------------------|------------|--|
| <i>className</i> | GRIDCOLUMN | Identifies the object as an instance of the GridColumn class (Property discussed in Chapter 5, "Core language.") |
| <i>dataLink</i> | | The field object that is linked to the column in the grid |
| <i>editorControl</i> | | The editable control that comprises the body of the grid in the column |
| <i>editorType</i> | Default | The type of editing control (0=Default, 1=EntryField, 2=CheckBox, 3=SpinBox, 4=ComboBox) in the column |
| <i>headingControl</i> | | The control that displays the grid column heading |

The following table lists the common properties, events, and methods of the GridColumn class:

| Property | | Event | Method |
|---------------|--------------|-------|--------|
| <i>parent</i> | <i>width</i> | none | none |

Description Each column in a grid is represented by a GridColumn object. Each GridColumn object is an element in the grid's *columns* array, and contains a reference to a

heading control and an edit control. You may assign different kinds of controls in different columns. The following types of controls are supported:

- Entryfield
- CheckBox
- SpinBox
- ComboBox

When these controls are used in a grid, they have a reduced property set. Each type of field has a default control type. Logical and boolean fields default to CheckBox. Numeric and date fields default to SpinBox.

See Also class ColumnCheckBox, class ColumnComboBox, class ColumnEntryfield, class ColumnSpinBox, class Grid

class Image

A rectangular region on a form that displays a bitmap image.

Syntax [*<oRef>* =] new Image(*<container>* [,*<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Image object.

<container> The container—typically a Form object—to which you’re binding the Image object.

<name expC> An optional name for the Image object. If not specified, the Image class will auto-generate a name for the object.

Properties The following table lists the properties of interest in the Image class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|-------------------|---------|---|
| <i>alignment</i> | Stretch | Determines the size and position of the graphic inside the Image object (0=Stretch, 1=Top left, 2=Centered, 3=Keep aspect stretch, 4=True size) |
| <i>anchor</i> | None | How the Image object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>className</i> | IMAGE | Identifies the object as an instance of the Image class (Property discussed in Chapter 5, “Core language.”) |
| <i>dataSource</i> | | The file or field that is displayed in the Image object |
| <i>fixed</i> | false | Whether the Image object’s position is fixed or if it can be “pushed down” or “pulled up” by the rendering or suppression of other objects. (See page 17-24.) |

The following table lists the common properties, events, and methods of the Image class:

| Property | | Event | | Method |
|---------------------|------------------|------------------------|--------------------------|-------------------|
| <i>before</i> | <i>name</i> | <i>canRender</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>enabled</i> | <i>pageno</i> | <i>onDesignOpen</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>form</i> | <i>parent</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> | |
| <i>height</i> | <i>printable</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> | |
| <i>hWnd</i> | <i>top</i> | <i>onLeftMouseUp</i> | <i>onRender</i> | |
| <i>id</i> | <i>visible</i> | <i>onMiddleDbClick</i> | <i>onRightDbClick</i> | |
| <i>left</i> | <i>width</i> | | <i>onRightMouseDown</i> | |
| <i>mousePointer</i> | | | <i>onRightMouseUp</i> | |

Description Use an Image object to display a bitmap image. The image can be data from a field, or a static image like a company logo.

Visual dBASE supports the following bitmap image formats:

- Graphics Interchange Format (GIF), including animated GIF
- Joint Photographic Experts Group (JPG, JPEG)
- Portable Network Graphics (PNG)
- X BitMap (XBM)
- Windows bitmap (BMP)
- Windows icon (ICO)
- Device Independent Bitmap (DIB)
- Windows metafile (WMF)
- Enhanced Windows metafile (EMF)
- PC Paintbrush (PCX)
- Tag Image File Format (TIF, TIFF)
- Encapsulated PostScript (EPS)

Visual dBASE will resize images according to the Image object's *alignment* property. When resizing, transparent GIF backgrounds are lost. To prevent resizing, set the *alignment* property to 4 (True size).

For TIFF, *Visual* dBASE supports uncompressed, single-bit Group 3, PackBits, and LZW (Lempel-Ziv & Welch) compression. Group 4 compression is not supported. Color TIFF images must have a palette. Except when rendering an EPS file on a PostScript-capable printer, *Visual* dBASE uses the bitmap preview in the EPS file, which must be in TIFF or WMF format.

See also class Form, class Line, REPLACE BINARY, *replaceFromFile()*

class Line

A line on a form.

Syntax [*<oRef>* =] new Line(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Line object.

<container> The container—typically a Form object—to which you're binding the Line object.

<name expC> An optional name for the Line object. If not specified, the Line class will auto-generate a name for the object.

Properties The following tables list the properties of interest of the Line class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|--------------------|------------|--|
| <i>bottom</i> | | The location of the bottom end of the Line in the form's current metric units, relative to the top edge of its container |
| <i>className</i> | LINE | Identifies the object as an instance of the Line class (Property discussed in Chapter 5, "Core language.") |
| <i>colorNormal</i> | WindowText | Color of the line |
| <i>fixed</i> | false | Whether the Line object's position is fixed or if it can be "pushed down" or "pulled up" by the rendering or suppression of other objects. (See page 17-24.) |
| <i>left</i> | | The location of the left end of the Line in the form's current metric units, relative to the left edge of its container |
| <i>pen</i> | Solid | The pen style used to draw the line (0=Solid, 1=Dash, 2=Dot, 3=DashDot, 4=DashDotDot) |
| <i>right</i> | | The location of the right end of the Line in the form's current metric units, relative to the left edge of its container |
| <i>top</i> | | The location of the top of the Line in the form's current metric units, relative to the top edge of its container |
| <i>width</i> | 1 | Width in pixels |

The following table lists the common properties, events, and methods of the Line class:

| Property | | Event | | Method |
|---------------|------------------|---------------------|-----------------|------------------|
| <i>before</i> | <i>pageno</i> | <i>canRender</i> | <i>onOpen</i> | <i>release()</i> |
| <i>form</i> | <i>parent</i> | <i>onDesignOpen</i> | <i>onRender</i> | |
| <i>id</i> | <i>printable</i> | | | |
| <i>name</i> | <i>visible</i> | | | |

Description Use a Line object to draw a line in a form or report. Note that the position properties—*top*, *left*, *bottom*, and *right*—work different for the Line object than they do with other components. The *width* property controls the thickness of the line.

A Line has no *hWnd* because it is drawn on the surface of the form; it is not a genuine Windows control. Despite its position in the form's z-order, a Line can never be drawn on top of another component (other than a Line or Shape).

See also class Form, class Image

class ListBox

A selection list on a form, from which you can pick multiple items.

Syntax [*<oRef>* =] new ListBox(*<container>* [,*<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created ListBox object.

<container> The container—typically a Form object—to which you’re binding the ListBox object.

<name expC> An optional name for the ListBox object. If not specified, the ListBox class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the ListBox class.

| Property | Default | Description |
|-----------------------|---------------------------|---|
| <i>className</i> | LISTBOX | Identifies the object as an instance of the ListBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | HighlightText / Highlight | The color of selected options |
| <i>colorNormal</i> | WindowText / Window | The color of unselected options |
| <i>curSel</i> | | The number of the option that has the focus rectangle |
| <i>dataSource</i> | | The options strings of the ListBox object |
| <i>multiple</i> | false | Whether the ListBox object allows selection of more than one option |
| <i>sorted</i> | false | Whether the options are sorted |
| <i>value</i> | | The value of the option that currently has focus |

| Event | Parameters | Description |
|--------------------|------------|---|
| <i>onSelChange</i> | | After the focus moves to another option in the list |

| Method | Parameters | Description |
|-------------------|------------|--|
| <i>count()</i> | | Returns the number of options in the list |
| <i>selected()</i> | | Returns the currently selected option(s) or checks if a specified option is selected |

The following table lists the common properties, events, and methods of the ListBox class:

| Property | | Event | | Method |
|----------------------|----------------------|-------------------------|--------------------------|--------------------|
| <i>before</i> | <i>hWnd</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>id</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>left</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>mousePointer</i> | <i>onLeftDbtClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>name</i> | <i>onLeftMouseDown</i> | <i>onRightDbtClick</i> | |
| <i>fontName</i> | <i>pageno</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>parent</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>printable</i> | <i>onMiddleDbtClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |

Description Use a ListBox object to present the user with a scrollable list of items. If the *multiple* property is *true*, the user can choose more than one item. The list of options is set with the *dataSource* property. The list of items selected is returned by calling the *selected()* method.

See also class ComboBox

class NoteBook

A multi-page container with rectangular tabs on top.

Syntax [*<oRef>* =] new NoteBook(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created NoteBook object.

<container> The container—typically a Form object—to which you’re binding the NoteBook object.

<name expC> An optional name for the NoteBook object. If not specified, the NoteBook class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the NoteBook class. (No particular methods are associated with this class.)

| Property | Default | Description |
|--------------------|----------|--|
| <i>anchor</i> | None | How the NoteBook object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>buttons</i> | false | Whether the notebook tabs appear as buttons instead |
| <i>className</i> | NOTEBOOK | Identifies the object as an instance of the NoteBook class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnFace | Color of the notebook background |
| <i>curSel</i> | | The number of the currently selected tab |

| Property | Default | Description |
|--------------------|---------------|---|
| <i>dataSource</i> | | The tab names for the notebook |
| <i>focus</i> | Normal | When to give focus to the notebook tabs when they are clicked (0=Normal, 1=On Button Down, 2=Never) |
| <i>multiple</i> | false | Whether the notebook tabs are displayed in multiple rows, or in a single row with a scrollbar |
| <i>visualStyle</i> | Right Justify | The style of the notebook tabs (0=Right Justify, 1=Fixed Width, 2=Ragged Right) |

| Event | Parameters | Description |
|--------------------|------------|--|
| <i>onSelChange</i> | | After a different notebook tab is selected |

The following table lists the common properties, events, and methods of the NoteBook class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|--------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>first</i> | <i>name</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>fontBold</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>fontItalic</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontName</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontSize</i> | <i>speedTip</i> | <i>onMiddleDbClick</i> | <i>when</i> | |
| <i>fontStrikeout</i> | <i>statusMessage</i> | | | |
| <i>fontUnderline</i> | <i>tabStop</i> | | | |
| <i>form</i> | <i>top</i> | | | |
| <i>height</i> | <i>visible</i> | | | |
| <i>helpFile</i> | <i>width</i> | | | |
| <i>helpId</i> | | | | |
| <i>hWnd</i> | | | | |

Description The NoteBook object combines aspects of the Form, Container, and TabBox objects. It's a multi-page control, like the Form; it acts as a container, and it has tabs, although they're on top. Selecting a tab automatically changes the page of the notebook to display the controls assigned to that page. The notebook's *pageno* property indicates which page of the form the notebook is in. The notebook's *curSel* property indicates the current page the notebook is displaying.

See also class Container, class TabBox

class OLE

Displays an OLE document that is stored in an OLE field, and lets the user initiate an action in the server application that created the document.

Syntax [*<oRef>* =] new OLE(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created OLE object.

<container> The container—typically a Form object—to which you’re binding the OLE object.

<name expC> An optional name for the OLE object. If not specified, the OLE class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the OLE class.

| Property | Default | Description |
|---------------------|---------|--|
| <i>alignment</i> | Stretch | Determines the size and position of the contents of the OLE object (0=Stretch, 1=Top left, 2=Centered, 3=Keep aspect stretch, 4=True size) |
| <i>anchor</i> | None | How the OLE object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>border</i> | false | Whether the OLE object is surrounded by the border specified by <i>borderStyle</i> |
| <i>className</i> | OLE | Identifies the object as an instance of the OLE class (Property discussed in Chapter 5, “Core language.”) |
| <i>dataLink</i> | | The Field object that is linked to the OLE object |
| <i>linkFileName</i> | | The OLE document file (if any) that is linked with the current OLE field. |
| <i>oleType</i> | | Number that reflects whether an OLE field is empty (0), contains an embedded document (1), or contains a link to a document file (2) |
| <i>serverName</i> | | The server application that is invoked when the user double-clicks on an OLE viewer object |

| Event | Parameters | Description |
|-----------------|------------|--|
| <i>onChange</i> | | After the contents of the OLE object have changed |
| <i>onClose</i> | | After the form containing the OLE object has been closed |

| Method | Parameters | Description |
|-----------------|---|------------------------------|
| <i>doVerb()</i> | <i><OLE verb expN></i> , <i><title expC></i> | Starts an OLE server session |

The following table lists the common properties, events, and methods of the OLE class:

| Property | | Event | | Method |
|---------------------|----------------------|---------------------|--------------------|-------------------|
| <i>before</i> | <i>name</i> | <i>onDesignOpen</i> | <i>onLostFocus</i> | <i>release()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onOpen</i> | <i>setFocus()</i> |
| <i>enabled</i> | <i>parent</i> | | | |
| <i>form</i> | <i>printable</i> | | | |
| <i>height</i> | <i>statusMessage</i> | | | |
| <i>hWnd</i> | <i>tabStop</i> | | | |
| <i>id</i> | <i>top</i> | | | |
| <i>left</i> | <i>visible</i> | | | |
| <i>mousePointer</i> | <i>width</i> | | | |

Description Place an OLE object in a form to view and edit a document stored in an OLE field. For example, if an OLE field contains a bitmap image created in Paintbrush, double-clicking the OLE object linked to the field starts a session in Paintbrush and places the image in the Paintbrush work area.

OLE stands for object linking and embedding. When you *link* a document to an OLE object, the OLE field does not contain the document itself; instead, it holds a link to a file containing the document. When you embed a document in an OLE field, a copy of the document is inserted into the OLE field, and no connection is made to a document file.

By double-clicking the OLE object, the user can invoke the application that created the OLE document. Therefore, if an image was created in Paintbrush and linked or embedded in the OLE field, double-clicking on the field starts a session in Paintbrush; the image is displayed in the Paintbrush drawing area, ready for editing. If the object was linked, any changes made in the Paintbrush session are stored in the document file; if the object was embedded, the changes are stored in the OLE field only.

An OLE viewer window object displays the contents of an OLE field. (Use the *dataLink* property to identify the field.) Each time the record pointer is moved, the contents of the viewer window are refreshed to display the OLE field in the current record.

See Also class ActiveX, class Image, REPLACE OLE, *replaceFromFile()*

class PaintBox

A generic control that can be placed on a form.

Syntax [*<oRef>* =] new PaintBox(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created PaintBox object.

<container> The container—typically a Form object—to which you’re binding the PaintBox object.

<name expC> An optional name for the PaintBox object. If not specified, the PaintBox class will auto-generate a name for the object.

Properties

The following tables list the properties and events of interest in the PaintBox class. (No particular methods are associated with this class.)

| Property | Default | Description |
|--------------------|-----------------|--|
| <i>className</i> | PAINTBOX | Identifies the object as an instance of the PaintBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/BtnFace | The color of the paintbox |
| <i>transparent</i> | false | Whether the paintbox background is the same as the background color or image of its container |

| Event | Parameters | Description |
|-------------------|--|---|
| <i>onChar</i> | <char expN>, <repeat expN>, <flags expN> | After a non-cursor key or key combination is pressed |
| <i>onClose</i> | | After the form containing the PaintBox object has been closed |
| <i>onFormSize</i> | | After the form containing the paintbox is resized |
| <i>onKeyDown</i> | <char expN>, <repeat expN>, <flags expN> | After any key is pressed |
| <i>onKeyUp</i> | <char expN>, <repeat expN>, <flags expN> | After any key is released |
| <i>onPaint</i> | | Whenever the paintbox needs to be redrawn |

The following table lists the common properties, events, and methods of the PaintBox class:

| Property | | Event | | Method |
|---------------------|----------------------|------------------------|--------------------------|--------------------|
| <i>before</i> | <i>name</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>parent</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>form</i> | <i>printable</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> | |
| <i>height</i> | <i>statusMessage</i> | <i>onLeftMouseUp</i> | <i>onRightDbClick</i> | |
| <i>hWnd</i> | <i>tabStop</i> | <i>onLostFocus</i> | <i>onRightMouseDown</i> | |
| <i>id</i> | <i>top</i> | <i>onMiddleDbClick</i> | <i>onRightMouseUp</i> | |
| <i>left</i> | <i>visible</i> | | | |
| <i>mousePointer</i> | <i>width</i> | | | |

Description

The PaintBox object is a generic control you can use to create a variety of objects. It is designed for advanced developers who want to create their own custom controls using the Windows API. It is simply a rectangular region of a form that has all the standard control properties such as *height*, *width*, and *before*, as well as all the standard mouse events.

In addition to the standard events or properties, the PaintBox object has three events that let you detect keystrokes entered when it has focus: *onChar*, *onKeyDown*, and *onKeyUp*. These let you create customized editing controls. The

onPaint and *onFormSize* events let you modify the appearance of the object based on user interaction.

class Progress

A progress indicator.

Syntax [*<oRef>* =] new Progress(<container> [, <name expC>])

<oRef> A variable or property—typically of <container>—in which to store a reference to the newly created Progress object.

<container> The container—typically a Form object—to which you’re binding the Progress object.

<name expC> An optional name for the Progress object. If not specified, the Progress class will auto-generate a name for the object.

Properties The following tables list the properties of interest in the Progress class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|------------------|----------|--|
| <i>className</i> | PROGRESS | Identifies the object as an instance of the Progress class (Property discussed in Chapter 5, “Core language.”) |
| <i>rangeMax</i> | | The maximum value |
| <i>rangeMin</i> | | The minimum value |
| <i>value</i> | | The current value |

The following table lists the common properties, events, and methods of the Progress class:

| Property | | Event | | Method |
|---------------------|------------------|------------------------|--------------------------|------------------|
| <i>before</i> | <i>pageno</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>parent</i> | <i>onLeftMouseDown</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>form</i> | <i>printable</i> | <i>onLeftMouseUp</i> | <i>onMouseMove</i> | |
| <i>height</i> | <i>speedTip</i> | | <i>onOpen</i> | |
| <i>hWnd</i> | <i>top</i> | | <i>onRightMouseDown</i> | |
| <i>left</i> | <i>visible</i> | | <i>onRightMouseUp</i> | |
| <i>mousePointer</i> | <i>width</i> | | | |
| <i>name</i> | | | | |

Description Use a Progress object to graphically indicate progress during processing. For example to display percentage completed, set the *rangeMin* to 0 and the *rangeMax* to 100. Then as the process progresses, set the *value* to the approximate percentage. The control will display the percentage graphically.

class PushButton

A button on a form.

Syntax [*<oRef>* =] new PushButton(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created PushButton object.

<container> The container—typically a Form object—to which you’re binding the PushButton object.

<name expC> An optional name for the PushButton object. If not specified, the PushButton class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the PushButton class. (No particular methods are associated with this class.)

| Property | Default | Description |
|-----------------------|------------------------|--|
| <i>className</i> | PUSHBUTTON | Identifies the object as an instance of the PushButton class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/ BtnFace | The color of the button |
| <i>default</i> | false | Whether the button is the default button on the form |
| <i>disabledBitmap</i> | | The bitmap to display on the button when it’s disabled |
| <i>downBitmap</i> | | The bitmap to display on the button when it’s pressed down |
| <i>focusBitmap</i> | | The bitmap to display on the button when it has focus |
| <i>group</i> | | The group to which the button belongs |
| <i>speedBar</i> | false | Whether the button acts like a tool button, which never gets focus |
| <i>text</i> | <same as <i>name</i> > | The text that appears on the PushButton face |
| <i>textLeft</i> | false | When a button has both a bitmap and text label, whether the text appears to the left or right of the bitmap |
| <i>toggle</i> | false | Whether the button acts like a toggle switch, staying down when pressed |
| <i>upBitmap</i> | 0 | The bitmap to display on the button when it’s not down and does not have focus |
| <i>value</i> | false | Whether the button is pressed (used when <i>toggle</i> is true) |

| Event | Parameters | Description |
|----------------|------------|---------------------------------|
| <i>onClick</i> | | After the PushButton is clicked |

The following table lists the common properties, events, and methods of the PushButton class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|--------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDbClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Use a PushButton object to execute an action when the user clicks it.

class RadioButton

A single radio button on a form. The user may choose one from a set.

Syntax [*<oRef>* =] new RadioButton(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created RadioButton object.

<container> The container—typically a Form object—to which you’re binding the RadioButton object.

<name expC> An optional name for the RadioButton object. If not specified, the RadioButton class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the RadioButton class. (No particular methods are associated with this class.)

| Property | Default | Description |
|--------------------|------------------------|---|
| <i>className</i> | RADIOBUTTON | Identifies the object as an instance of the RadioButton class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/BtnFace | The color of the radio button’s text label |
| <i>dataLink</i> | | The Field object that is linked to the Radio object |
| <i>group</i> | | The group to which the radio button belongs |
| <i>text</i> | <same as <i>name</i> > | The text label that appears beside the radio button |
| <i>textLeft</i> | false | Whether the radio button’s text label appears to the left or to the right of the radio button |

| Property | Default | Description |
|--------------------|---------|--|
| <i>transparent</i> | false | Whether the RadioButton object has the same background color or image as its container |
| <i>value</i> | | Whether the radio button is visually marked as selected |

| Event | Parameters | Description |
|-----------------|------------|---|
| <i>onChange</i> | | After the radio button gets selected or loses its selection |

The following table lists the common properties, events, and methods of the RadioButton class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDbClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Use a group of RadioButton objects to present the user a set of multiple choices, from which they can choose only one.

Each set of choices on a form must have the same *group* property. If there is only one group of radio buttons on a form, the *group* can be left blank. You may use any string or number as the *group*.

You may also use *true* and *false* in the *group* property to create radio button groups. Use *true* for the first button in each radio button group, and *false* for the rest. For example, if you create seven radio buttons and set the *group* property of the first and fourth radio button to *true*, the first three buttons form one group, and the last four form another. The two groups are independent; the user can select one button in the first group and one button in the other.

See also class CheckBox

class Rectangle

A rectangle with a caption.

Syntax [*<oRef>* =] new Rectangle(*<container>* [,*<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Rectangle object.

<container> The container—typically a Form object—to which you’re binding the Rectangle object.

<name expC> An optional name for the Rectangle object. If not specified, the Rectangle class will auto-generate a name for the object.

Properties The following tables list the properties of interest of the Rectangle class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|---------------------|---------------------------|--|
| <i>border</i> | true | Whether the Rectangle object’s rectangle is visible |
| <i>borderStyle</i> | Default | Specifies the rectangle style (0=Default, 1=Raised, 2=Lowered, 3=None, 4=Single, 5=Double, 6=Drop Shadow, 7=Client, 8=Modal, 9=Etched In, 10=Etched Out) |
| <i>className</i> | RECTANGLE | Identifies the object as an instance of the Progress class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/ BtnFace | The color of the caption and the rectangle fill |
| <i>patternStyle</i> | Solid | The fill pattern style (0=Solid, 1=BDiagonal, 2=Cross, 3=Diagcross, 4=FDiagonal, 5=Horizontal, 6=Vertical) |
| <i>text</i> | <same as <i>name</i> > | The text caption that appears at the top right of the rectangle |

The following table lists the common properties, events, and methods of the Rectangle class:

| Property | | Event | Method |
|----------------------|---------------------|-------------------------|-----------------------------------|
| <i>before</i> | <i>hWnd</i> | <i>onDesignOpen</i> | <i>onMouseDown</i> <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onLeftDblClick</i> | <i>onMouseUp</i> <i>release()</i> |
| <i>fontBold</i> | <i>mousePointer</i> | <i>onLeftMouseDown</i> | <i>onMouseMove</i> |
| <i>fontItalic</i> | <i>name</i> | <i>onLeftMouseUp</i> | <i>onOpen</i> |
| <i>fontName</i> | <i>pageno</i> | <i>onMiddleDblClick</i> | <i>onRightDblClick</i> |
| <i>fontSize</i> | <i>parent</i> | | <i>onRightMouseDown</i> |
| <i>fontStrikeout</i> | <i>printable</i> | | <i>onRightMouseUp</i> |
| <i>fontUnderline</i> | <i>top</i> | | |
| <i>form</i> | <i>visible</i> | | |
| <i>height</i> | <i>width</i> | | |

Description Use a Rectangle object to enclose an area of a form. For example, you can use a Rectangle object to draw a border around a group of related objects, such as a group of radio buttons.

To assign a label that describes the group of objects, use the *text* property. The label appears in the top left corner of the rectangle.

A Rectangle object does not affect other objects. The user can't give focus to the Rectangle object, and it doesn't display or modify data.

See Also Class Line, Class Shape

class ReportViewer

A control to display a report on a form.

Syntax [*<oRef>* =] new ReportViewer(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created ReportViewer object.

<container> The container—typically a Form object—to which you're binding the ReportViewer object.

<name expC> An optional name for the ReportViewer object. If not specified, the ReportViewer class will auto-generate a name for the object.

Properties The following tables list the properties and methods of interest in the ReportViewer class. (No particular events are associated with this class.)

| Property | Default | Description |
|------------------|--------------|--|
| <i>anchor</i> | None | How the ReportViewer object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>className</i> | REPORTVIEWER | Identifies the object as an instance of the ReportViewer class (Property discussed in Chapter 5, "Core language.") |
| <i>filename</i> | | The name of the .REP file containing the report to view |
| <i>params</i> | | Parameters passed to the .REP file |
| <i>ref</i> | | A reference to the Report object being viewed |
| <i>scrollBar</i> | Auto | When a scroll bar appears for the ReportViewer object (0=Off, 1=On, 2=Auto, 3=Disabled) |

| Method | Parameters | Description |
|--------------------|------------|------------------------|
| <i>reExecute()</i> | | Regenerates the report |

The following table lists the common properties, events, and methods of the ReportViewer class:

| Property | | Event | Method |
|--------------------|---------------|-------|--------------------|
| <i>before</i> | <i>name</i> | none | <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | | <i>release()</i> |
| <i>form</i> | <i>parent</i> | | <i>setFocus()</i> |
| <i>height</i> | <i>top</i> | | |
| <i>left</i> | <i>width</i> | | |

Description Use a ReportViewer object to view a report in a form. Assign any parameters to the *params* property, then set the *filename* property to the name of the .REP file; this executes the named report file. You may access the report object being viewed through the *ref* property.

If report parameters are assigned after setting the *filename* property, you must call the *reExecute()* method to regenerate the report.

class ScrollBar

A vertical or horizontal scrollbar used to represent a range of numeric values.

Syntax [*<oRef>* =] new ScrollBar(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created ScrollBar object.

<container> The container—typically a Form object—to which you’re binding the ScrollBar object.

<name expC> An optional name for the ScrollBar object. If not specified, the ScrollBar class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the ScrollBar class. (No particular methods are associated with this class.)

| Property | Default | Description |
|--------------------|-----------|---|
| <i>className</i> | SCROLLBAR | Identifies the object as an instance of the ScrollBar class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | ScrollBar | The color of the scrollbar |
| <i>dataLink</i> | | The Field object that is linked to the ScrollBar object |
| <i>rangeMax</i> | | The maximum value |
| <i>rangeMin</i> | | The minimum value |
| <i>value</i> | | The current value |
| <i>vertical</i> | true | Whether the scrollbar is vertical or horizontal |

| Event | Parameters | Description |
|-----------------|------------|-----------------------------------|
| <i>onChange</i> | | After the scrollbar value changes |

The following table lists the common properties, events, and methods of the ScrollBar class:

| Property | | Event | | Method |
|---------------------|----------------------|-------------------------|--------------------------|--------------------|
| <i>before</i> | <i>name</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>parent</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>form</i> | <i>printable</i> | <i>onLeftDblClick</i> | <i>onOpen</i> | |
| <i>height</i> | <i>speedTip</i> | <i>onLeftMouseDown</i> | <i>onRightDblClick</i> | |
| <i>helpFile</i> | <i>statusMessage</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>helpId</i> | <i>tabStop</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>hWnd</i> | <i>top</i> | <i>onMiddleDblClick</i> | <i>when</i> | |
| <i>id</i> | <i>visible</i> | | | |
| <i>left</i> | <i>width</i> | | | |
| <i>mousePointer</i> | | | | |

Description

The class Slider, class SpinBox
 ScrollBar object is maintained primarily for compatibility. Use a Slider instead. **See Also**

class Shape

A simple colored geometric shape.

Syntax [*<oRef>* =] new Shape(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Shape object.

<container> The container—typically a Form object—to which you’re binding the Shape object.

<name expC> An optional name for the Shape object. If not specified, the Shape class will auto-generate a name for the object.

Properties The following tables list the properties of interest of the Shape class. (No particular events or methods are associated with this class.)

| Property | Default | Description |
|--------------------|---------------------|---|
| <i>className</i> | SHAPE | Identifies the object as an instance of the Shape class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | WindowText / Window | The pen and fill color of the shape |
| <i>penStyle</i> | Solid | The pen style used to draw the outline of the shape (0=Solid, 1=Dash, 2=Dot, 3=DashDot, 4=DashDotDot) |
| <i>penWidth</i> | 1 | Width of the outline in pixels |
| <i>shapeStyle</i> | Circle | The type of shape to draw (0=Round Rectangle, 1=Rectangle, 2=Ellipse, 3=Circle, 4=Round square, 5=Square) |

The following table lists the common properties, events, and methods of the Shape class:

| Property | | Event | | Method |
|---------------|------------------|---------------------|-----------------|-------------------|
| <i>before</i> | <i>pageno</i> | <i>canRender</i> | <i>onOpen</i> | <i>move()</i> |
| <i>form</i> | <i>parent</i> | <i>onDesignOpen</i> | <i>onRender</i> | <i>release()</i> |
| <i>height</i> | <i>printable</i> | | | |
| <i>left</i> | <i>top</i> | | | |
| <i>name</i> | <i>visible</i> | | | |
| | <i>width</i> | | | |

Description Use a Shape object to draw a basic geometric shape on a form.

A Shape has no *hWnd* because it is drawn on the surface of the form; it is not a genuine Windows control. Despite its position in the form’s z-order, a Shape can never be drawn on top of another component (other than a Line or Shape).

See Also class Image, class Line, class Rectangle

class Slider

A horizontal or vertical slider for choosing magnitude.

Syntax [*<oRef>* =] new Slider(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Slider object.

<container> The container—typically a Form object—to which you’re binding the Slider object.

<name expC> An optional name for the Slider object. If not specified, the Slider class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the Slider class.

| Property | Default | Description |
|------------------------|--------------|--|
| <i>className</i> | SLIDER | Identifies the object as an instance of the Slider class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnFace | The color of the slider |
| <i>enableSelection</i> | false | Whether to display the selection range |
| <i>endSelection</i> | | The value of the end of the selection range |
| <i>rangeMax</i> | | The maximum value |
| <i>rangeMin</i> | | The minimum value |
| <i>startSelection</i> | | The value of the start of the selection range |
| <i>tics</i> | Auto | How to display the tic marks (0=Auto, 1=Manual, 2=None) |
| <i>ticsPos</i> | Bottom Right | Where to display the tic marks (0=Both, 1=Bottom Right, 2=Top Left) |
| <i>value</i> | | The current value |
| <i>vertical</i> | false | Whether the slider is vertical or horizontal |

| Event | Parameters | Description |
|-----------------|------------|-----------------------------------|
| <i>onChange</i> | | After the slider position changes |

| Method | Parameters | Description |
|--------------------------|------------|--|
| <i>clearTics()</i> | | Clears all manually-set tic marks |
| <i>setTic()</i> | <expN> | Manually sets a tic mark at the specified position |
| <i>setTicFrequency()</i> | <expN> | Sets the automatic tic mark interval |

The following table lists the common properties, events, and methods of the Slider class:

| Property | | Event | Method |
|---------------------|----------------------|------------------------|---|
| <i>before</i> | <i>name</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> <i>move()</i> |
| <i>borderStyle</i> | <i>pageno</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> <i>release()</i> |
| <i>enabled</i> | <i>parent</i> | <i>onHelp</i> | <i>onMouseMove</i> <i>setFocus()</i> |
| <i>form</i> | <i>printable</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> |
| <i>height</i> | <i>speedTip</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> |
| <i>helpFile</i> | <i>statusMessage</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> |
| <i>helpId</i> | <i>tabStop</i> | | <i>when</i> |
| <i>hWnd</i> | <i>top</i> | | |
| <i>id</i> | <i>visible</i> | | |
| <i>left</i> | <i>width</i> | | |
| <i>mousePointer</i> | | | |

Description Use a slider to let users vary numeric values visually. Unlike spinboxes, sliders don't accept keyboard input or use a step value. Instead, the user drags the slider pointer to increase or decrease the value.

As the user moves the slider pointer, the value is continually updated to reflect the position of the pointer. For example, a slider that varies a numeric value

between 1 and 100 sets the value to 50 when the slider pointer is at the center of the slider.

To set a range for the slider, set *rangeMin* to the minimum value and *rangeMax* to the maximum value.

You may also designate a separate selection region inside the slider with the *startSelection*, *endSelection*, and *enableSelection* properties.

You have complete control over the tick marks that appear in the slider.

See also class ScrollBar, class SpinBox

class SpinBox

An entryfield with a spinner for entering numeric or date values.

Syntax [*<oRef>* =] new SpinBox(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created SpinBox object.

<container> The container—typically a Form object—to which you’re binding the SpinBox object.

<name expC> An optional name for the SpinBox object. If not specified, the SpinBox class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the SpinBox class.

| Property | Default | Description |
|-----------------------|---------------------|---|
| <i>border</i> | true | Whether the SpinBox object is surrounded by the border specified by <i>borderStyle</i> |
| <i>className</i> | SPINBOX | Identifies the object as an instance of the SpinBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | WindowText / Window | The color of the text in the SpinBox object when the object has focus |
| <i>colorNormal</i> | WindowText / Window | The color of the text in the SpinBox object when the object does not have focus |
| <i>dataLink</i> | | The Field object that is linked to the SpinBox object |
| <i>function</i> | | A text formatting function |
| <i>picture</i> | | Formatting template |
| <i>rangeMax</i> | | The maximum value |
| <i>rangeMin</i> | | The minimum value |
| <i>rangeRequired</i> | false | Whether the range values are enforced even when no change has been made |
| <i>selectAll</i> | true | Whether the entryfield contents are initially selected when the SpinBox object gets focus |

| Property | Default | Description |
|----------------------|---------------|--|
| <i>spinOnly</i> | false | Whether the value may be changed using the spinner only or typing is allowed |
| <i>step</i> | 1 | The value added or subtracted when using the spinner |
| <i>validErrorMsg</i> | Invalid input | The message that is displayed when the <i>valid</i> event fails |
| <i>validRequired</i> | false | Whether to fire the <i>valid</i> event even when no change has been made |
| <i>value</i> | | The value currently displayed in the SpinBox object |

| Event | Parameters | Description |
|-----------------|------------|--|
| <i>onChange</i> | | After the spinner is clicked |
| <i>valid</i> | | When attempting to remove focus. Must return <i>true</i> , or focus remains. |

| Method | Parameters | Description |
|-------------------|------------|---|
| <i>copy()</i> | | Copies selected text to the Windows Clipboard |
| <i>cut()</i> | | Cuts selected text and to the Windows Clipboard |
| <i>keyboard()</i> | <expC> | Simulates typed user input to the SpinBox object |
| <i>paste()</i> | | Copies text from the Windows Clipboard to the current cursor position |
| <i>undo()</i> | | Reverses the effects of the most recent <i>cut()</i> , <i>copy()</i> , or <i>paste()</i> action |

The following table lists the common properties, events, and methods of the SpinBox class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>enabled</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>name</i> | <i>onLeftDbClick</i> | <i>onOpen</i> | |
| <i>fontItalic</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbClick</i> | |
| <i>fontName</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontSize</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontStrikeout</i> | <i>speedTip</i> | <i>onMiddleDbClick</i> | <i>when</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description

Use a spinbox to let users enter values by typing them in the textbox or by clicking the spinner arrows.

By setting *spinOnly* to *true*, you can control the rate at which users change numeric or date values. For example, one spin box might change an interest rate in increments of hundredths, while another might change a date value in week

increments. Set the size of each increment with the *step* property; for example, if you set *step* to 5, each click on an arrow changes a numeric value by 5 or a date value by 5 days.

To restrict entries to those within a particular range of values, set the *rangeMin* property to the minimum value and *rangeMax* to the maximum value, then set *rangeRequired* to *true*.

See Also class Slider

class TabBox

A set of folder-style (trapezoidal) bottom-tabs.

Syntax [*<oRef>* =] new TabBox(<container> [,<name expC>])

<oRef> A variable or property—typically of <container>—in which to store a reference to the newly created TabBox object.

<container> The container—typically a Form object—to which you’re binding the TabBox object.

<name expC> An optional name for the TabBox object. If not specified, the TabBox class will auto-generate a name for the object.

Properties The following tables list the properties and events of interest in the TabBox class. (No particular methods are associated with this class.)

| Property | Default | Description |
|-----------------------|-----------------|--|
| <i>anchor</i> | Bottom | How the TabBox object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>className</i> | TABBOX | Identifies the object as an instance of the TabBox class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorHighlight</i> | BtnText/BtnFace | The color of the selected tab |
| <i>colorNormal</i> | BtnFace | The color of the background behind the tabs |
| <i>curSel</i> | | The number of the currently selected tab |
| <i>dataSource</i> | | The tab names |

| Event | Parameters | Description |
|--------------------|------------|-----------------------------------|
| <i>onSelChange</i> | | After a different tab is selected |

The following table lists the common properties, events, and methods of the TabBox class:

| Property | | Event | | Method |
|----------------------|----------------------|-------------------------|--------------------------|-------------------|
| <i>before</i> | <i>id</i> | <i>onDesignOpen</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>enabled</i> | <i>left</i> | <i>onGotFocus</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>fontBold</i> | <i>mousePointer</i> | <i>onHelp</i> | <i>onMouseMove</i> | <i>setFocus()</i> |
| <i>fontItalic</i> | <i>name</i> | <i>onLeftDbtClick</i> | <i>onOpen</i> | |
| <i>fontName</i> | <i>pageno</i> | <i>onLeftMouseDown</i> | <i>onRightDbtClick</i> | |
| <i>fontSize</i> | <i>parent</i> | <i>onLeftMouseUp</i> | <i>onRightMouseDown</i> | |
| <i>fontStrikeout</i> | <i>printable</i> | <i>onLostFocus</i> | <i>onRightMouseUp</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | <i>onMiddleDbtClick</i> | <i>when</i> | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description A TabBox contains a number of tabs that users can select.

By setting the *pageno* property of a TabBox control to 0 (the default), you can implement a tabbed multi-page form where the user can easily switch pages by selecting tabs. Use the *pageno* property of a control to determine on which page the control appears, and use the *curSel* property and *onSelChange* event of the TabBox to switch between pages.

See Also class Notebook

class Text

Non-editable HTML text on a form.

Syntax [*<oRef>* =] new Text(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created Text object.

<container> The container—typically a Form object—to which you're binding the Text object.

<name expC> An optional name for the Text object. If not specified, the Text class will auto-generate a name for the object.

Properties The following tables list the properties and methods of interest in the Text class. (No particular events are associated with this class.)

| Property | Default | Description |
|------------------------|----------|--|
| <i>alignHorizontal</i> | Left | Determines how the text displays within the horizontal plane of its rectangular frame (0=Left, 1=Center, 2=Right, 3=Justify) |
| <i>alignment</i> | Top left | Combines the <i>alignHorizontal</i> , <i>alignVertical</i> , and <i>wrap</i> properties (maintained for compatibility) |

| Property | Default | Description |
|------------------------------|------------------------|--|
| <i>alignVertical</i> | Top | Determines how the text displays in the vertical plane of its rectangular frame (0=Top, 1=Center, 2=Bottom, 3=Justify) |
| <i>anchor</i> | None | How the Text object is anchored in its container (0=None, 1=Bottom, 2=Top, 3=Left, 4=Right, 5=Center, 6=Container) |
| <i>border</i> | false | Whether the Text object is surrounded by the border specified by <i>borderStyle</i> |
| <i>className</i> | TEXT | Identifies the object as an instance of the Text class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | BtnText/BtnFace | The color of the text |
| <i>fixed</i> | false | Whether the Text object’s position is fixed or if it can be “pushed down” or “pulled up” by the rendering or suppression of other objects. (See page 17-24.) |
| <i>function</i> | | A text formatting function |
| <i>leading</i> | 0 | The distance between consecutive lines; if 0 uses the font’s default leading. (See page 17-29.) |
| <i>marginHorizontal</i> | | The horizontal margin between the text and its rectangular frame. (See page 17-30.) |
| <i>marginVertical</i> | | The vertical margin between the text and its rectangular frame. (See page 17-31.) |
| <i>picture</i> | | Formatting template |
| <i>prefixEnable</i> | true | Whether to interpret the ampersand (&) character in the <i>text</i> as the accelerator prefix. |
| <i>rotate</i> | 0 | The text orientation, in increments of 90 degrees (0=0, 1=90, 2=180, 3=270). (See page 17-39.) |
| <i>suppressIfBlank</i> | false | Whether the Text object is suppressed (not rendered) if it is blank. (See page 17-41.) |
| <i>suppressIfDuplicate</i> | false | Whether the Text object is suppressed (not rendered) if its value is the same as the previous time it was rendered. (See page 17-41.) |
| <i>text</i> | <same as <i>name</i> > | The value of the Text object; the text that appears |
| <i>tracking</i> | 0 | The space between characters; if 0 uses the font’s default. (See page 17-42.) |
| <i>trackJustifyThreshold</i> | 0 | The maximum amount of added space between words on a fully justified line; 0 indicates no limit. (See page 17-42.) |
| <i>transparent</i> | false | Whether the Text object has the same background color or image as its container |
| <i>variableHeight</i> | false | Whether the Text object’s height can increase based on its value. (See page 17-42.) |

| Property | Default | Description |
|-----------------------------|---------|--|
| <i>verticalJustifyLimit</i> | 0 | The maximum additional space between lines that can be added to attempt to justify vertically. If the limit is exceeded the Text object is top justified. A value of 0 means no limit. (See page 17-43.) |
| <i>wrap</i> | true | Whether to word-wrap the text in the Text object |

| Method | Parameters | Description |
|------------------------|---------------------|--|
| <i>getTextExtent()</i> | <i><expC></i> | Returns the width of the specified string using the Text object's font |

The following table lists the common properties, events, and methods of the Text class:

| Property | | Event | | Method |
|----------------------|---------------------|------------------------|--------------------------|------------------|
| <i>before</i> | <i>id</i> | <i>canRender</i> | <i>onMiddleMouseDown</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>left</i> | <i>onDesignOpen</i> | <i>onMiddleMouseUp</i> | <i>release()</i> |
| <i>fontBold</i> | <i>mousePointer</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> | |
| <i>fontItalic</i> | <i>name</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> | |
| <i>fontName</i> | <i>pageno</i> | <i>onLeftMouseUp</i> | <i>onRender</i> | |
| <i>fontSize</i> | <i>parent</i> | <i>onMiddleDbClick</i> | <i>onRightDbClick</i> | |
| <i>fontStrikeout</i> | <i>printable</i> | | <i>onRightMouseDown</i> | |
| <i>fontUnderline</i> | <i>top</i> | | <i>onRightMouseUp</i> | |
| <i>form</i> | <i>visible</i> | | | |
| <i>height</i> | <i>width</i> | | | |
| <i>hWnd</i> | | | | |

Description Use a Text component to display information in a form or report. The *text* property of the component may contain any text, including HTML tags.

The *text* property may be an expression codeblock, which is evaluated every time it is rendered.

See also class Editor

class TreeItem

An item in a TreeView.

Syntax [*<oRef>* =] new TreeItem(*<parent>* [, *<name expC>*])

<oRef> A variable or property—typically of *<parent>*—in which to store a reference to the newly created TreeItem object.

<parent> The parent object—a TreeView object for top-level items, or another TreeItem—to which you're binding the TreeItem object.

<name expC> An optional name for the TreeItem object. If not specified, the TreeItem class will auto-generate a name for the object.

Properties The following tables list the properties and methods of interest in the TreeItem class. (No particular events are associated with this class.)

| Property | Default | Description |
|----------------------|----------|--|
| <i>bold</i> | false | Whether the text label is bold |
| <i>checked</i> | false | Whether the item is visually marked as checked |
| <i>className</i> | TREEITEM | Identifies the object as an instance of the TreeItem class (Property discussed in Chapter 5, “Core language.”) |
| <i>expanded</i> | false | Whether the item’s children are shown or hidden |
| <i>firstChild</i> | | The first child tree item |
| <i>handle</i> | | The Windows tree item handle (similar to <i>hWnd</i>) |
| <i>image</i> | | Image displayed between checkbox and text label when item does not have focus |
| <i>level</i> | | The tree level of the item |
| <i>nextSibling</i> | | The next tree item with the same parent |
| <i>noOfChildren</i> | | The number of child tree items |
| <i>prevSibling</i> | | The previous tree item with the same parent |
| <i>selectedImage</i> | | Image displayed between checkbox and text label when item has focus |
| <i>text</i> | | The text label of the tree item |

| Method | Parameters | Description |
|-----------------------------|------------|--|
| <i>ensureVisible()</i> | | Expands the tree and scrolls the tree view if necessary to make the tree item visible |
| <i>select()</i> | | Makes the tree item the selected item in the tree |
| <i>setAsFirstVisible()</i> | | Expands the tree and scrolls the tree view if necessary to try to make the tree item the first (topmost) visible tree item |
| <i>sortChildren()</i> | | Sorts the child tree items |

The following table lists the common properties, events, and methods of the TreeItem class:

| Property | Event | Method |
|---------------|-------|-------------------|
| <i>name</i> | none | <i>release()</i> |
| <i>parent</i> | | |

Description Each item in a tree view can contain text, an icon image that can change when the the item is selected, and a checkbox. You can replace the checkbox images with a different pair of images to represent any two-state condition.

A `TreeItem` object can contain other `TreeItem` objects as child objects in a subtree, which can be expanded or collapsed.

See also `class TreeView`

class TreeView

An expandable tree.

Syntax [*<oRef>* =] new `TreeView`(*<container>* [, *<name expC>*])

<oRef> A variable or property—typically of *<container>*—in which to store a reference to the newly created `TreeView` object.

<container> The container—typically a `Form` object—to which you’re binding the `TreeView` object.

<name expC> An optional name for the `TreeView` object. If not specified, the `TreeView` class will auto-generate a name for the object.

Properties The following tables list the properties, events, and methods of interest in the `TreeView` class.

| Property | Default | Description |
|--------------------------|---------------------|---|
| <i>allowEditLabels</i> | true | Whether the text labels of the tree items are editable |
| <i>allowEditTree</i> | true | Whether items can be added or deleted from the tree |
| <i>checkBoxes</i> | true | Whether each tree item has a checkbox |
| <i>checkedImage</i> | | The image to display when a tree item is checked instead of a checked check box |
| <i>className</i> | TREEVIEW | Identifies the object as an instance of the <code>TreeView</code> class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | WindowText / Window | The color of the text labels and background |
| <i>disablePopup</i> | false | Whether the tree view’s popup menu is disabled |
| <i>firstChild</i> | | The first child tree item |
| <i>firstVisibleChild</i> | | The first tree item that is visible in the tree view area |
| <i>hasButtons</i> | true | Whether + and - icons are displayed for tree items that have children |
| <i>hasLines</i> | true | Whether lines are drawn between tree items |
| <i>image</i> | | Default image displayed between checkbox and text label when a tree item does not have focus |
| <i>imageScaleToFont</i> | true | Whether tree item images automatically scale to match the text label font height |
| <i>imageSize</i> | | The height of tree item images in pixels |

| Property | Default | Description |
|-----------------------|---------|--|
| <i>indent</i> | | The horizontal indent, in pixels, for each level of tree items |
| <i>linesAtRoot</i> | true | Whether a line connects the tree items at the first level |
| <i>selected</i> | | The currently selected tree item |
| <i>selectedImage</i> | | Default image displayed between checkbox and text label when a tree item has focus |
| <i>showSelAlways</i> | true | Whether to highlight the selected item in the tree even when the tree view does not have focus |
| <i>toolTips</i> | true | Whether to display text labels as tooltips if they are too long to display fully in the tree view area as the mouse passes over them |
| <i>trackSelect</i> | true | Whether to highlight and underline tree items as the mouse passes over them |
| <i>uncheckedImage</i> | | The image to display when a tree item is not checked instead of an empty check box |

| Event | Parameters | Description |
|------------------------|-------------|---|
| <i>canChange</i> | | Before selection moves to another tree item; return value determines if selection can leave current tree item |
| <i>canEditLabel</i> | | When attempting to edit text label; return value determines whether editing is allowed |
| <i>canExpand</i> | | When attempting to expand or collapse a tree item; return value determines whether expand/collapse occurs |
| <i>onChange</i> | | After the selection moves to another tree item |
| <i>onCheckBoxClick</i> | | After a checkbox in a tree item is clicked |
| <i>onEditLabel</i> | <text expC> | After the text label in a tree item is edited; may optionally return a different label value to save |
| <i>onExpand</i> | | After a tree item is expanded or collapsed |

| Method | Parameters | Description |
|-----------------------------|------------|--|
| <i>count()</i> | | Returns the total number of tree items in the tree |
| <i>releaseAllChildren()</i> | | Deletes all tree items in the tree |
| <i>sortChildren()</i> | | Sorts child tree items |
| <i>visibleCount()</i> | | Returns the number of tree items visible in the tree view area |

The following table lists the common properties, events, and methods of the `TreeView` class:

| Property | | Event | | Method |
|----------------------|----------------------|------------------------|--------------------------|--------------------|
| <i>before</i> | <i>hWnd</i> | <i>onDesignOpen</i> | <i>onMiddleDbClick</i> | <i>move()</i> |
| <i>borderStyle</i> | <i>id</i> | <i>onGotFocus</i> | <i>onMiddleMouseDown</i> | <i>release()</i> |
| <i>enabled</i> | <i>left</i> | <i>onHelp</i> | <i>onMiddleMouseUp</i> | <i>setFocus()</i> |
| <i>fontBold</i> | <i>mousePointer</i> | <i>onLeftDbClick</i> | <i>onMouseMove</i> | |
| <i>fontItalic</i> | <i>name</i> | <i>onLeftMouseDown</i> | <i>onOpen</i> | |
| <i>fontName</i> | <i>pageno</i> | <i>onLeftMouseUp</i> | <i>onRightDbClick</i> | |
| <i>fontSize</i> | <i>parent</i> | <i>onLostFocus</i> | <i>onRightMouseDown</i> | |
| <i>fontStrikeout</i> | <i>printable</i> | | <i>onRightMouseUp</i> | |
| <i>fontUnderline</i> | <i>statusMessage</i> | | | |
| <i>form</i> | <i>tabStop</i> | | | |
| <i>height</i> | <i>top</i> | | | |
| <i>helpFile</i> | <i>visible</i> | | | |
| <i>helpId</i> | <i>width</i> | | | |

Description A TreeView displays a collapsible multi-level tree. There are three ways to create the tree:

- Explicitly add items with code, like the code generated by the Form Designer.
- Interactively through the TreeView’s runtime user interface (right-clicking or pressing certain keys).
- Data-driven code that reads a table and dynamically creates the tree items.

The TreeView object acts as the root of the tree. It contains the first level of TreeItem objects, which can contain their own TreeItem objects, thus forming a tree.

Unlike the deeper levels of the tree, you cannot collapse the first level of a tree. Therefore, you may want to use only one item at the first level of the tree to make the entire tree collapsible.

See also class TreeItem

abandonRecord()

Abandons changes to the current record.

Syntax <oRef>.abandonRecord()
<oRef> An object reference to the form.

Property of Form

Description Use *abandonRecord()* for form-based data handling with tables in work areas. When using the data access objects, *abandonRecord()* has no effect; use the rowset’s *abandon()* method instead.

Form-based data buffering lets you manage the editing of existing records and the appending of new records. The temporary record created by *beginAppend()*

and editing changes to the current record are not written to the table until there is navigation off the record, or until *saveRecord()* is called. Each work area has its own separate edit buffer. For example, if you *beginAppend()* in two separate work areas, you must call *abandonRecord()* while each work area is selected to abandon the changes.

Before calling *abandonRecord()*, you can use *isRecordChanged()* to determine if changes have been made. If so, you may want to confirm the action before proceeding.

Example See *isRecordChanged()*.

See Also *beginAppend()*, *isRecordChanged()*, *saveRecord()*

activeControl

Contains a reference to the object that currently has focus.

Property of Form

Description Use the *activeControl* property to reference the object that currently has focus.

An object gets focus in three ways:

- The user tabs to the object.
- The user clicks the object.
- The *setFocus()* method of the object is executed.

See Also *before, first, id, nextObj*

alias

Determines the table that is accessed by a Browse object.

Property of Browse

Description Use the *alias* property to identify a table to display in a browse object. For example, when the form is based on a .QBE query that opens two or more files in a parent-child relation, you use *alias* to determine which table appears in the Browse object.

Aliases are used for tables open in work areas, not data access objects. When the form uses data access objects, use a Grid control, which can *dataLink* directly to a Rowset object.

See Also *fields, view*

alignHorizontal

Determines the horizontal alignment of text in a Text component.

Property of Text

Description *alignHorizontal* determines the way the text displays within the horizontal plane of its rectangular frame. Set it to one of the following:

| Value | Alignment |
|-------|-----------|
| 0 | Left |
| 1 | Center |
| 2 | Right |
| 3 | Justify |

See also *alignVertical*

alignment [Image]

Determines the size and position of the graphic inside an Image object.

Property of Image

Description If a graphic is smaller than the Image object that displays it, it can be stretched to fill the Image object or positioned inside the Image object with empty space around it. Assign one of the following settings to the *alignment* property of an Image object to determine how the graphic is aligned.

| Setting | Description |
|-------------------------|--|
| 0 (Stretch) | Enlarge graphic to fill the entire Image object |
| 1 (Top Left) | In the top left corner of the Image object |
| 2 (Center) | Centered in the Image object |
| 3 (Keep Aspect Stretch) | Maintains the original height/width (aspect) ratio when stretching the graphic until it fills either dimension of the Image object |
| 4 (True Size) | No changes to the graphic |

If the graphic is larger than the Image object, both Stretch and Keep Aspect Stretch will reduce the graphic to fit the Image object so that the entire image is visible. Top Left and Center will both display whatever fits in the Image object.

True Size does not change the graphic at all. The Image object is dynamically resized to display the graphic. This is the fastest option, because *Visual* dBASE doesn't have to do any stretching or shrinking.

See Also *height, width*
alignment is also a property of the Text class.

alignment [Text]

Positions text in a Text object.

Property of Text

Description The Text object's *alignment* property is maintained primarily for compatibility. It is an enumerated property that can have the following values:

| Setting | Description |
|-------------------|---|
| 0 (Top Left) | Adjacent to the top edge and the left edge |
| 1 (Top Center) | Adjacent to the top edge and centered horizontally |
| 2 (Top Right) | Adjacent to the top edge and the right edge |
| 3 (Center Left) | Centered vertically and adjacent to left edge |
| 4 (Center) | Centered horizontally and vertically |
| 5 (Center Right) | Centered vertically and adjacent to right edge |
| 6 (Bottom Left) | Adjacent to the bottom edge and the left edge |
| 7 (Bottom Center) | Centered horizontally and adjacent to the bottom edge |
| 8 (Bottom Right) | Adjacent to the bottom edge and the right edge |
| 9 (Wrap Left) | Same as Top Left |
| 10 (Wrap Center) | Same as Top Center |
| 11 (Wrap Right) | Same as Top Right |

The Text object's *alignHorizontal* and *alignVertical* properties control the alignment in the horizontal and vertical plane separately. They also include options to justify the text. When either property is set, the *alignment* property is also changed to match (justify becomes top or left). When the *alignment* property is set, the other two properties are changed to match.

Text wrapping is controlled by the *wrap* property.

See also *alignHorizontal, alignVertical, wrap*
alignment is also a property of the Image class.

alignVertical

Determines the vertical alignment of text in a Text component.

Property of Text

Description *alignVertical* determines the way the text displays within the vertical plane of its rectangular frame. Set it to one of the following:

| Value | Alignment |
|-------|-----------|
| 0 | Top |
| 1 | Middle |
| 2 | Bottom |
| 3 | Justify |

See also *alignHorizontal*

allowAddRows

Whether rows can be added directly through a Grid object.

Property of Grid

Description *allowAddRows* determines whether moving down past the last row in a grid starts the append of a new row. It has no control over adding row in other ways, like by calling the rowset's *beginAppend()* method. If the rowset switches to Append mode, the grid will synchronize itself with the rowset and display the new row.

Set *allowAddRows* to *false* to prevent the accidental appending of new rows when navigating through a grid.

See also *allowEditing*

allowColumnMoving

Whether the user may rearrange the columns in a grid with the mouse.

Property of Grid

Description By default, *allowColumnMoving* is *true*, which means that the user can rearrange the columns in a grid by clicking and dragging the column headings. Set *allowColumnMoving* to *false* to disable this ability.

See also *allowColumnSizing*, *dragScrollRate*

allowColumnSizing

Whether the user may resize the columns in a grid with the mouse.

Property of Grid

Description By default, *allowColumnSizing* is *true*, which means that the user can resize the columns in a grid by clicking and dragging between the columns headings. Set *allowColumnSizing* to *false* to disable this ability.

See also *allowColumnMoving*, *allowRowSizing*

allowEditing

Whether a grid is read-only.

Property of Grid

Description By default, *allowEditing* is *true*. Set *allowEditing* to *false* to make the grid read-only.

See also *allowAddRows*

allowEditLabels

Whether the text labels of the tree items are editable.

Property of TreeView

Description When *allowEditLabels* is *true*, the user can edit the *text* of the tree items in a tree by pressing *F2* or clicking on a tree item a second time with a tree item selected. Set *allowEditLabels* to *false* to prevent the user from editing all the items in a tree, or use the *canEditLabel* event to conditionally allow or prevent editing.

See also *allowEditTree*, *canEditLabel*, *onEditLabel*

allowEditTree

Whether items can be added or deleted from the tree.

Property of TreeView

Description When *allowEditTree* is *true*, the user can add another leaf node to a tree item by pressing *Ins*, or delete a tree item by pressing *Del* when the tree item is selected. Set *allowEditTree* to *false* to prevent the user from inserting or deleting items in the tree.

See also *allowEditLabels*

allowRowSizing

Whether the user may resize the rows in a grid with the mouse.

Property of Grid

Description By default, *allowRowSizing* is *true*, which means that the user can resize the rows in a grid by clicking and dragging between the row indicator in the left column. Set *allowRowSizing* to *false* to disable this ability.

See also *allowColumnSizing*, *cellHeight*

anchor

Specifies whether an object stays in the same relative position when its container is resized.

Property of ActiveX, Browse, Container, Editor, Image, Grid, NoteBook, OLE, ReportViewer, TabBox, Text

Description Use *anchor* to specify whether a control should maintain its location and resize itself to match its container, which is usually the form. Anchored controls claim space in the order in which they are created (the z-order). Once an anchored control claims space in its container, that space cannot be used by another anchored control.

anchor is an enumerated property with the following values:

| Value | Description |
|-------|---------------------|
| 0 | None, do not anchor |
| 1 | Bottom |
| 2 | Top |
| 3 | Left |
| 4 | Right |
| 5 | Center |
| 6 | Container |

When anchored to the bottom or top, the *width* of the control resizes to match the width of its container. When anchored to the left or right, the *height* of the control resizes to match the height of its container. Center and container anchors behave identically: the control fills the center of the container, sizing itself to fill all the space not claimed by another anchored control; if there are no other anchored controls in the container, the control resizes to fill its container.

See Also *form*, *height*, *width*

append

Whether records can be added directly through a Browse object.

Property of Browse

Description *append* determines whether moving down past the last record in a browse starts the append of a new record. It has no control over adding records in other ways, like by calling the form's *beginAppend()* method. If a new record is added, the browse will show it.

Set *append* to *false* to prevent the accidental appending of new records when navigating through a browse.

See Also *modify*

autoCenter

Determines if a form is automatically centered when it is opened.

Property of Form

Description Use *autoSize* to automatically center a form when it is opened. If *autoCenter* is *true*, MDI forms are centered in the MDI frame window; SDI forms are centered on-screen. If *autoCenter* is *false*, the form is positioned according to its *top* and *left* properties.

See also *autoSize, left, mdi, top*

autoDrop

Determines if the drop-down list drops automatically when the combobox gets focus.

Property of ComboBox

Description Set *autoDrop* to *true* to make the drop-down list portion of a combobox drop automatically when the combobox gets focus. Whenever the combobox loses focus, its drop-down list is always closed, no matter how it was opened.

autoDrop has no effect when the *style* of the combobox is Simple (0).

See also *dropDownHeight, dropDownWidth, style*

autoSize

Determines if a form is automatically sized to contain its objects when the form is opened.

Property of Form

Description Use *autoSize* to determine how a form is sized and proportioned.

If you set the *autoSize* property of a form to *true*, the form is automatically adjusted to contain its objects when it is opened. If you set *autoSize* to *false*, the form assumes its assigned *height* and *width* when it is opened.

When you set the *autoSize* property of a form to *true*, the default dimensions are ignored. The user can still move or resize the form, but if the form is closed and reopened it is automatically resized again to contain its objects.

See Also *autoCenter*, *height*, *width*

background

A form's background image.

Property of Form

Description Set the *background* property to the file name of a bitmap you want tiled in the background of your form. See class Image for the list of bitmap formats supported by Visual dBASE.

You may use any Visual dBASE-supported bitmap format.

Setting a background image supersedes the background color designated by the form's *colorNormal* property.

See also class Image, *colorNormal*, *transparent*

before

The next object in the z-order of the form.

Property of All form components

Description An object's *before* property contains a reference to the next object in the z-order, in other words, the object the current object comes before. The z-order is the order in which controls are created on the form. It is the same order in which they are created; the same order as they are listed in the .WFM file. If objects

`beginAppend()`

overlap, the one that is later in the z-order is drawn on top, with the exception of Line and Shape objects, which are always drawn on the form surface.

The form's *first* property contains a reference to the first control in the z-order. The z-order is a closed loop. The *before* property of the last control in the z-order points back to the first control.

The form's tab order is related to the z-order. The objects are in the same order, but only those objects that can receive focus are in the tab order. All visual components in the form are somewhere in the z-order. Non-visual components, such as Query objects, are not in the z-order.

The *before* property is read-only. You must reorder the objects in the form's class definition, by editing the code in the .WFM file or using the Form designer to visually reorder the objects.

Example Suppose you have a form that contains a bunch of spinboxes for measurements. You want to be able to set them all to zero with the click of a button.

```
function resetButton_onClick()
  local obj
  obj = form.first           // First control in z-order
  do
    if obj.className == "SPINBOX"
      obj.value := 0
    endif
    obj := obj.before       // Next control in z-order
  until obj == form.first  // Until you get back to first control
```

Compare this loop with the example for *elements*.

See Also *activeControl, elements, first, nextObj*

beginAppend()

Creates a temporary buffer in memory for a record that is based on the structure of the current table, letting the user input data to the record without automatically adding the record to the table.

Syntax `<oRef>.beginAppend()`
<oRef> An object reference to the form.

Property of Form

Description Use *beginAppend()* for form-based data handling with tables in work areas. When using the data access objects, *beginAppend()* has no effect; use the rowset's *beginAppend()* method instead.

beginAppend() creates a single record buffer in the current table, without actually adding the record to the table until *saveRecord()* is issued. While this buffer exists, the user can input data to the record with controls such as an entry

field or a check box. Use *saveRecord()* to append the record to the currently active table, and use *abandonRecord()* to discard the record. Calling *beginAppend()* instead of *saveRecord()* will write the new record and empty the buffer again so you can add another record. Use *isRecordChanged()* to determine if the record has been changed since the *beginAppend()* was issued.

When appending records with *beginAppend()* the new record will not be saved when you call *saveRecord()* unless there have been changes to the record; the blank new record is abandoned. This prevents the saving of blank records in the table. (If you want to create blank records, use APPEND BLANK). You can check there have been changes by calling *isRecordChanged()*. If *isRecordChanged()* returns *true*, you should validate the record with form-level or row-level validation before writing it to the table.

Using *beginAppend()* has different results than using either BEGINTRANS() and APPEND BLANK or APPEND AUTOMEM. With these commands, if you cancel the append operation, you have a record marked as deleted added to the table. If you use *abandonRecord()* to cancel the *beginAppend()* operation, a new record is never added to the table.

See Also *abandonRecord()*, *isRecordChanged()*, *saveRecord()*
beginAppend() is also a method of the Rowset class (page 14-43)

bgColor

The background color of an object.

Property of Grid

Description You may specify any single color for the background color. For a list of valid colors, see *colorNormal*.

See also *colorNormal*

bold

Whether the text of an object is bold.

Property of TreeItem

Description Set *bold* to *true* to make the text of a TreeItem object bold.

See also *fontBold*, *text*

border

Determines whether an object is surrounded with a border.

Property of Editor, Entryfield, OLE, Rectangle, SpinBox, Text

Description The *border* property is maintained primary for compatability. Every object that has a *border* property also has a *borderStyle* property. One of the choices for *borderStyle* is None, while *border* can be either *true* or *false*. Both these properties apply.

If you pick an actual border with *borderStyle*, *border* determines whether that border is displayed. If you choose the None *borderStyle*, no border will appear, even if *border* is *true*.

See Also *borderStyle*

borderStyle

Determines the border around the object.

Property of Most form components

Description *borderStyle* determines the display style of an object’s rectangular frame. Set it to one of the following:

| Value | Alignment |
|-------|-------------|
| 0 | Default |
| 1 | Raised |
| 2 | Lowered |
| 3 | None |
| 4 | Single |
| 5 | Double |
| 6 | Drop shadow |
| 7 | Client |
| 8 | Modal |
| 9 | Etched in |
| 10 | Etched out |

The border is drawn inside the bounds of the object; therefore for thick borders like Drop shadow, there is noticeably less space in the object for the actual contents.

The border is not drawn if *border* is *false*. If *borderStyle* is None, no border appears even if *border* is *true*.

See also *border*

bottom

The vertical position of the lower end of a Line object.

Property of Line

Description Use the *bottom* property in combination with the *right*, *left*, and *top* properties to determine the position and length of a line object.

The unit of measurement in a form or report is controlled by its *metric* property. The default metric for forms is characters, and for reports it's twips.

See Also *left, metric, right, top, width*

buttons

Whether a notebook's tabs appear as buttons

Property of NoteBook

Description Set *buttons* to *true* if you want the notebook tabs to appear as separate buttons instead of tabs attached to the notebook page.

See also *visualStyle*

canChange

Event fired before selection moves to another tree item; return value determines if selection can leave current tree item.

Parameters none

Property of TreeView

Description Use *canChange* to prevent focus from moving to another item in the tree unless certain conditions are met. The *canChange* event handler can either return a value of *true*, which allows the focus to move, or *false*, which prevents the focus change.

The event handler usually uses the tree view's *selected* property to get the currently selected tree item. Note that if no tree item is selected, the property contains *null*, so your event handler must check for that. In particular, when deleting a tree item, the focus must move to another tree item, and the currently selected item has just been deleted, and therefore *selected* will be *null*.

Example The following *canChange* event handler makes sure that the tree item's text label does not have any spaces in it:

```
function TREEVIEW1_canChange
if not empty( this.selected ) // Make sure there is a currently selected item
if " " $ this.selected.text
msgbox( "Spaces are not allowed", "Alert", 48 )
return false
endif
endif
return true
```

Note that you would probably also use the *onEditLabel* event to warn the user of an invalid text label at the time they change it.

See also *onChange*, *select()*, *selected*
canChange is also an event of the Field class (page 14-97).

canClose

Event fired when an attempt is made to close the form; return value determines if the form can be closed.

Parameters none

Property of Form

Description Use *canClose* to prevent a form from closing until certain conditions are met. The *canClose* event handler can either return a value of *true*, which allows the form to close, or *false*, which prevents the form from closing.

When a form is closed by calling *close()* or clicking the Close icon, pending changes in the data buffer are saved. When attempting to close, the form fires the current control's *valid* event, if any, so there's no need to verify individual controls if they have *valid* event handlers. However, you should always perform form- or row-level validation to check controls that you have not visited.

Example The following *canClose* event handler fires the form's row-level validation method:

```
function form_canClose()
return form.recValid() // If row is bad, don't close
```

See Also *close()*, *onClose*
canClose is also a method of the Query class (page 14-54)

canEditLabel

Event fired when attempting to edit text label; return value determines if editing is allowed.

Parameters none

Property of TreeView

Description Use *canEditLabel* to conditionally allow editing of a tree item's text label. The *canEditLabel* event handler can either return *true* to allow editing, or *false* to prevent it.

Set *allowEditLabels* to *false* to prevent all label editing. In that case, *canEditLabel* will never fire.

Example The following *canEditLabel* event handler prevents editing in the first level of the tree:

```
{| | this.selected.level > 1}
```

See also *allowEditLabels*, *onEditLabel*

canExpand

Event fired when attempting to expand or collapse a tree item; return value determines whether expand/collapse occurs.

Parameters none

Property of TreeView

Description Use *canExpand* to conditionally allow the expansion or collapsing of a tree item's subtree. The *canExpand* event handler can either return *true* to allow the action, or *false* to prevent it.

canExpand only affects the user interface. You can still expand or collapse a tree item programmatically by setting the item's *expanded* property, in which case *canExpand* does not fire.

Example Suppose you want to display customer and their orders in a tree view named *custTree*. To make the form load faster, you initially add only the first order for each customer (so that customer with orders will have the + button), then load all other orders on-demand through the *canExpand* event. You call the following method in the form's overridden *open()* method so that the tree is initialized before the form opens:

```
function initTree
  local tCust
```

```

if form.rowset.first()
do
  tCust = new TreeItem( form.custTree, ;
                        "C" + form.rowset.fields[ "CUST_ID" ].value )
  tCust.demandLoaded = false // Create new properties
  tCust.bookmark      = form.rowset.bookmark()
  with tCust
    text = form.rowset.fields[ "LAST_NAME" ].value.rightTrim() + "," + ;
          form.rowset.fields[ "FIRST_NAME" ].value.rightTrim()
    if not form.orders1.rowset.endOfSet
      with new TreeItem( tCust, ;
                        "O" + form.orders1.rowset.fields[ "ORDER_NUM" ].value )
        text = form.orders1.rowset.fields[ "ORDER_NUM" ].value + " " + ;
              form.orders1.rowset.fields[ "ORDER_DATE" ].value
      endwith
    endif
  endwith
until not form.rowset.next()
endif

```

Unique names are generated for each level of tree items and passed as the second parameter to the `TreeItem` class constructor; duplicate names are not allowed. The customer and order rowsets are linked with *masterRowset* so that navigation in the customer rowset automatically navigates to the corresponding orders. The *text* property is assigned inside a `WITH` block in case other stock properties are assigned in the future, which would also go inside the `WITH` block. The custom *demandLoaded* and *bookmark* properties for the top-level tree item must be created outside the `WITH` block; you can't create properties in a `WITH` block. Here is the *canExpand* event handler:

```

function CUSTTREE_canExpand
local t, r
t = this.selected // TreeItem being expanded
r = form.orders1.rowset // Detail rowset
if not t.expanded and not t.demandLoaded
  form.rowset.goto( t.bookmark )
  do while r.next() // Start with second detail row (if any)
    with new TreeItem( t, "O" + r.fields[ "ORDER_NUM" ].value )
      text = r.fields[ "ORDER_NUM" ].value + " " + ;
            r.fields[ "ORDER_DATE" ].value
    endwith
  enddo
  t.demandLoaded := true
endif
return true

```

The event handler creates some short-hand variables for the tree item being expanded and the order rowset. It then checks to see if the tree item is being expanded (it is if its *expanded* property is *false* to begin with) and has not been demand-loaded yet. If so, it goes to the saved bookmark and immediately tries to go to the second matching order. By calling *next()* at the top of the `DO WHILE` loop, nothing happens if there is only one matching order, and that's all that is needed to loop through all the matching orders.

After adding the rest of the orders, the custom *demandLoaded* property is set to *true* so that this code is skipped the next time this customer's tree item is expanded. Finally, the event handler always returns *true* to allow the expansion (or collapse, if that's what triggered the event).

Note that this technique would not work in a situation where the first detail row might change while the user is viewing the tree. In that case, when the detail rows are demand-loaded, the row that was loaded when the tree was initialized would be loaded again, causing an error with the duplicate name. But if, for example, the orders are stored chronologically, this could not happen.

See also *expanded, hasButtons, onExpand*

canNavigate

Event fired when an attempt is made to navigate in a table; return value determines if the record pointer moves.

Parameters **<workarea expN>** The work area number where the navigation is attempted.

Property of Form

Description The form's *canNavigate* event is used mainly for form-based data handling with tables in work areas. It also fires when attempting navigation in the form's primary rowset.

Use *canNavigate* to prevent navigation until certain conditions are met. Navigation saves pending changes in the data buffer, so you should call row- or form-level validation in the *canNavigate* to make sure data should be saved.

Because *canNavigate* fires while still on the current record, you may also use it to perform some action just before you leave. In this case, the *canNavigate* would always return *true* to allow the navigation.

When using tables in work areas, *canNavigate* will not fire unless the form is open and has controls *dataLinked* to fields. For example, if you USE a table, create and open an empty Form, assign an *canNavigate* event handler, and SKIP in the table, the *canNavigate* will not fire simply because the form is open.

When attempting navigation in the form's primary rowset, the form's *canNavigate* fires before the rowset's *canNavigate*, and the *<workarea expN>* parameter is zero. If the form's *canNavigate* returns *false*, nothing further happens; the rowset's *canNavigate* does not fire, and no navigation occurs.

Example The following *canNavigate* event handler fires the form's row-level validation method, but only if the navigation occurred in the current table; for example not in some table being used for a lookup.

```
function form_canNavigate( nWorkArea )
  if nWorkArea == workarea()
    return form.recValid() // If row is bad, don't move
```

`cellHeight`

```
else
    return true
endif
```

See Also *onNavigate*, *rowset*

canNavigate is also a method of the Rowset class (page 14-56)

cellHeight

The height of each cell in the grid.

Property of Grid

Description The *cellHeight* property reflects the height of the cells in the body of the grid.

See also *allowRowSizing*

checkboxes

Whether each tree item has a checkbox.

Property of TreeView

Description When *checkboxes* is *true*, each tree item has a checkbox to the left of the text label and optional icon image. This checkbox is linked to the tree item's *checked* property. Whenever a checkbox is checked or unchecked, the tree's *onCheckBoxClick* event fires.

Use the *checkedImage* and *uncheckedImage* image properties to specify alternate images instead of the standard checkbox. Set *checkboxes* to *false* to hide and disable the checkboxes in the tree.

See also *checked*, *checkedImage*, *image*, *onCheckBoxClick*, *uncheckedImage*

checked

Whether the item is visually marked as checked.

Property of TreeItem

Description TreeItem objects may be visually checked and unchecked by the user, or by assigning a value to the *checked* property. If *checked* is *true*, the tree item's checkbox is checked, or its *checkedImage* is displayed. If *checked* is *false*, the tree item's checkbox is unchecked, or its *uncheckedImage* is displayed.

See also *checkboxes, checkedImage, onCheckBoxClick, uncheckedImage*

checkedImage

The image to display when a tree item is checked instead of a checked check box.

Property of TreeView

Description Use *checkedImage* to display a specific icon instead of the standard checked checkbox for the tree items in the tree that are checked. *uncheckedImage* optionally specifies the icon to display for tree items that are not checked. The tree must have its *checked* property set to *true* to enable checking; each tree item has a *checked* property that reflects whether the item is checked.

The *checkedImage* property is a string that can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies an icon resource and the DLL file that holds it.
- FILENAME <filename>
specifies an ICO icon file.

See also *checkboxes, checked, imageScaleToFont, imageSize, uncheckedImage*

classId

The ID string of an ActiveX control.

Property of ActiveX

Description To use an ActiveX control in a form, set the *classId* property to the control's ID string.

See also *nativeObject*

clearTics()

Clears manually-set tic marks in a Slider object.

Syntax <oRef>.clearTics()
<oRef> The Slider object whose tics to clear.

Property of Slider

Description Call *clearTics()* to clear all tic marks set by *setTic()*

See also *setTic()*, *tics*, *ticsPos*

clientEdge

Whether an object appears to have a beveled inside edge.

Property of Form

Description Set *clientEdge* to *true* to bevel a form's inside edge.

See also *background*, *borderStyle*

close()

Closes a form.

Syntax <oRef>.close()

<oRef> An object reference to the form to close.

Property of Form

Description Use *close()* to close an open form or report.

When you try to close a form, *Visual* dBASE does the following:

- 1 Fires the *valid* event of the current object. If it returns a value of *false*, the form doesn't close.
- 2 Fires the *onLostFocus* event of the current object.
- 3 Fires the *canClose* event of the form. If it returns *false*, the form doesn't close.
- 4 Fires the *onLostFocus* event of the form.
- 5 Removes the form and the objects it contains from the screen.
- 6 Fires the *onClose* event of the form.
- 7 Removes the form from memory if there are no other object references pointing to the form.

When a form is closed with *close()* or by clicking the Close icon, any pending changes in the record buffer are saved, as if *saveRecord()* was called. Closing a form by pressing *Esc* abandons changes before closing (if *escExit* is *true*).

If the form definition is not removed from memory, you can open the form again with *open()*.

See also *canClose*, *open()*, *readModal()*

close() is also a method of the Database (page 14-59) and File (page 11-7) classes.

colorHighlight

Sets the color of the object that has focus.

Property of Browse, Entryfield, ListBox, SpinBox, TabBox

Description Use the *colorHighlight* and *colorNormal* properties of an object so that users can differentiate visually when an object has focus and when it doesn't. You may choose from the same color settings as the *colorNormal* property.

The *colorHighlight* of most controls defaults to an empty string, meaning that it is colored no differently when it has focus. This way, you may set a particular color in the control's *colorNormal* property, without having to override a default *colorHighlight* color as well.

The color scheme in a TabBox is different. There, the *colorNormal* designates the color of the background behind the tabs, and the *colorHighlight* is the color of the selected tab. The unselected tabs are a fixed color, WindowText/Window.

See also *colorNormal*

colorNormal

The color of an object.

Property of Most form objects

Description Use the *colorNormal* and *colorHighlight* properties of an object so that users can differentiate visually when an object has focus and when it doesn't.

For some controls, in particular background colors, you specify a single color. For other controls, you specify two color settings with *colorNormal*: a foreground color (for text), and a background color. Color settings must be separated with a forward slash (/). Each color may be one of the following five color types, in any combination:

- Windows-named color
- Basic 16-color color code
- Hexadecimal RGB color triplet
- User-defined color name
- JavaScript color name

Color settings are not case-sensitive.

Windows-named colors are taken from the settings in the Display Properties. If the colors are changed in the Display Properties while a form is open, the form and any controls that use these values will change automatically. You can use

any of the following Windows-named color settings for either the foreground or background color:

Table 15.1 Windows-named colors

| Color | Corresponding Display Properties Appearance Color |
|---------------------|---|
| ActiveBorder | Active window border |
| ActiveCaption | Active title bar |
| AppWorkspace | Application background |
| Background | Desktop |
| BtnFace | 3D objects |
| BtnHighlight | A shade lighter than 3D objects |
| BtnShadow | A shade darker than 3D objects |
| BtnText | 3D objects font |
| CaptionText | Active title bar font |
| GrayText | preset gray—not available |
| Highlight | Selected items |
| HighlightText | Selected items font |
| InactiveBorder | Inactive window border |
| InactiveCaption | Inactive title bar |
| InactiveCaptionText | Inactive title bar font |
| InfoText | ToolTip |
| InfoBk | ToolTip font |
| Menu | Menu |
| MenuText | Menu font |
| Scrollbar | A shade lighter than 3D objects |
| Window | Window |
| WindowFrame | preset drop shadow—not available |
| WindowText | Window font |

The following one- and two-letter basic color codes (with an optional + or * for brightness) are provided primary for compatibility with earlier versions of dBASE.

Table 15.2 Basic 16-color color codes

| Color name | Foreground code | Background code |
|------------|-----------------|-----------------|
| Black | N | N |
| Dark Blue | B | B |
| Green | G | G |
| Cyan | GB or BG | GB or BG |
| Dark Red | R | R |
| Purple | RB or BR | RB or BR |
| Brown | RG or GR | RG or GR |

Table 15.2 Basic 16-color color codes

| Color name | Foreground code | Background code |
|--------------|-----------------|-----------------|
| Light Gray | W | W |
| Dark Gray | N+ | N* |
| Blue | B+ | B* |
| Bright Green | G+ | G+ |
| Bright Cyan | GB+ or BG+ | GB* or BG* |
| Red | R+ | R* |
| Magenta | RB+ or BR+ | RB* or BR* |
| Yellow | RG+ or GR+ | RG* or GR* |
| White | W+ | W* |

Hexadecimal RGB (Red Green Blue) color triplets are expressed backwards in *Visual* dBASE; that is, Blue, Green, Red. You may specify one of approximately 16 million colors using a triplet. The color will be displayed as well as the settings of the display allow.

You may create your own RGB combinations and give them a name with the `DEFINE COLOR` command.

Finally, *Visual* dBASE supports JavaScript-standard color names. The following table lists those colors and their corresponding RGB values.

Table 15.3 JavaScript color names and RGB values

| Color | Red | Green | Blue |
|----------------|-----|-------|------|
| aliceblue | F0 | F8 | FF |
| antiquewhite | FA | EB | D7 |
| aqua | 00 | FF | FF |
| aquamarine | 7F | FF | D4 |
| azure | F0 | FF | FF |
| beige | F5 | F5 | DC |
| bisque | FF | E4 | C4 |
| black | 00 | 00 | 00 |
| blanchedalmond | FF | EB | CD |
| blue | 00 | 00 | FF |
| blueviolet | 8A | 2B | E2 |
| brown | A5 | 2A | 2A |
| burlywood | DE | B8 | 87 |
| cadetblue | 5F | 9E | A0 |
| chartreuse | 7F | FF | 00 |
| chocolate | D2 | 69 | 1E |
| coral | FF | 7F | 50 |
| cornflowerblue | 64 | 95 | ED |

Table 15.3 JavaScript color names and RGB values

| Color | Red | Green | Blue |
|----------------|-----|-------|------|
| cornsilk | FF | F8 | DC |
| crimson | DC | 14 | 3C |
| cyan | 00 | FF | FF |
| darkblue | 00 | 00 | 8B |
| darkcyan | 00 | 8B | 8B |
| darkgoldenrod | B8 | 86 | 0B |
| darkgray | A9 | A9 | A9 |
| darkgreen | 00 | 64 | 00 |
| darkkhaki | BD | B7 | 6B |
| darkmagenta | 8B | 00 | 8B |
| darkolivegreen | 55 | 6B | 2F |
| darkorange | FF | 8C | 00 |
| darkorchid | 99 | 32 | CC |
| darkred | 8B | 00 | 00 |
| darksalmon | E9 | 96 | 7A |
| darkseagreen | 8F | BC | 8F |
| darkslateblue | 48 | 3D | 8B |
| darkslategray | 2F | 4F | 4F |
| darkturquoise | 00 | CE | D1 |
| darkviolet | 94 | 00 | D3 |
| deeppink | FF | 14 | 93 |
| deepskyblue | 00 | BF | FF |
| dimgray | 69 | 69 | 69 |
| dodgerblue | 1E | 90 | FF |
| firebrick | B2 | 22 | 22 |
| floralwhite | FF | FA | F0 |
| forestgreen | 22 | 8B | 22 |
| fuchsia | FF | 00 | FF |
| gainsboro | DC | DC | DC |
| ghostwhite | F8 | F8 | FF |
| gold | FF | D7 | 00 |
| goldenrod | DA | A5 | 20 |
| gray | 80 | 80 | 80 |
| green | 00 | 80 | 00 |
| greenyellow | AD | FF | 2F |
| honeydew | F0 | FF | F0 |
| hotpink | FF | 69 | B4 |
| indianred | CD | 5C | 5C |
| indigo | 4B | 00 | 82 |
| ivory | FF | FF | F0 |

Table 15.3 JavaScript color names and RGB values

| Color | Red | Green | Blue |
|----------------------|-----|-------|------|
| khaki | F0 | E6 | 8C |
| lavender | E6 | E6 | FA |
| lavenderblush | FF | F0 | F5 |
| lawngreen | 7C | FC | 00 |
| lemonchiffon | FF | FA | CD |
| lightblue | AD | D8 | E6 |
| lightcoral | F0 | 80 | 80 |
| lightcyan | E0 | FF | FF |
| lightgoldenrodyellow | FA | FA | D2 |
| lightgreen | 90 | EE | 90 |
| lightgrey | D3 | D3 | D3 |
| lightpink | FF | B6 | C1 |
| lightsalmon | FF | A0 | 7A |
| lightseagreen | 20 | B2 | AA |
| lightskyblue | 87 | CE | FA |
| lightslategray | 77 | 88 | 99 |
| lightsteelblue | B0 | C4 | DE |
| lightyellow | FF | FF | E0 |
| lime | 00 | FF | 00 |
| limegreen | 32 | CD | 32 |
| linen | FA | F0 | E6 |
| magenta | FF | 00 | FF |
| maroon | 80 | 00 | 00 |
| mediumaquamarine | 66 | CD | AA |
| mediumblue | 00 | 00 | CD |
| mediumorchid | BA | 55 | D3 |
| mediumpurple | 93 | 70 | DB |
| mediumseagreen | 3C | B3 | 71 |
| mediumslateblue | 7B | 68 | EE |
| mediumspringgreen | 00 | FA | 9A |
| mediumturquoise | 48 | D1 | CC |
| mediumvioletred | C7 | 15 | 85 |
| midnightblue | 19 | 19 | 70 |
| mintcream | F5 | FF | FA |
| mistyrose | FF | E4 | E1 |
| moccasin | FF | E4 | B5 |
| navajowhite | FF | DE | AD |
| navy | 00 | 00 | 80 |
| oldlace | FD | F5 | E6 |
| olive | 80 | 80 | 00 |

Table 15.3 JavaScript color names and RGB values

| Color | Red | Green | Blue |
|---------------|-----|-------|------|
| olivedrab | 6B | 8E | 23 |
| orange | FF | A5 | 00 |
| orangered | FF | 45 | 00 |
| orchid | DA | 70 | D6 |
| palegoldenrod | EE | E8 | AA |
| palegreen | 98 | FB | 98 |
| paleturquoise | AF | EE | EE |
| palevioletred | DB | 70 | 93 |
| papayawhip | FF | EF | D5 |
| peachpuff | FF | DA | B9 |
| peru | CD | 85 | 3F |
| pink | FF | C0 | CB |
| plum | DD | A0 | DD |
| powderblue | B0 | E0 | E6 |
| purple | 80 | 00 | 80 |
| red | FF | 00 | 00 |
| rosybrown | BC | 8F | 8F |
| royalblue | 41 | 69 | E1 |
| saddlebrown | 8B | 45 | 13 |
| salmon | FA | 80 | 72 |
| sandybrown | F4 | A4 | 60 |
| seagreen | 2E | 8B | 57 |
| seashell | FF | F5 | EE |
| sienna | A0 | 52 | 2D |
| silver | C0 | C0 | C0 |
| skyblue | 87 | CE | EB |
| slateblue | 6A | 5A | CD |
| slategray | 70 | 80 | 90 |
| snow | FF | FA | FA |
| springgreen | 00 | FF | 7F |
| steelblue | 46 | 82 | B4 |
| tan | D2 | B4 | 8C |
| teal | 00 | 80 | 80 |
| thistle | D8 | BF | D8 |
| tomato | FF | 63 | 47 |
| turquoise | 40 | E0 | D0 |
| violet | EE | 82 | EE |
| wheat | F5 | DE | B3 |
| white | FF | FF | FF |
| whitesmoke | F5 | F5 | F5 |

Table 15.3 JavaScript color names and RGB values

| Color | Red | Green | Blue |
|-------------|-----|-------|------|
| yellow | FF | FF | 00 |
| yellowgreen | 9A | CD | 32 |

Example Both of the following strings represent the color orange and can be used as the *colorNormal* property:

```
0x00a5ff
orange
```

See also *background*, *colorHighlight*, *DEFINE COLOR* (page 16-15), *transparent*

columnCount

The number of columns in the grid.

Property of Grid

Description *columnCount* is a read-only property that contains the number of columns in the grid; either the number of columns that are automatically created when no GridColumn objects are specified, or the number of GridColumn objects explicitly added.

See also *columns*, *currentColumn*

columns

An array of objects for each column in the grid.

Property of Grid

Description A grid's *columns* array contains explicitly created GridColumn objects, one for each column. If no GridColumn objects are created, the grid automatically creates columns, but the *columns* array is empty.

See also *columnCount*, *currentColumn*

copy()

Copies selected text to the Windows clipboard.

Syntax <oRef>.copy()

<oRef> An object reference to the control from which to copy the text.

`count()`

Property of Browse, ComboBox, Editor, Entryfield, SpinBox

Description Use *copy()* when the user has selected text and wants to copy it to the Windows clipboard.

When calling this method from a pushbutton's *onClick* event, the pushbutton should have its *speedBar* property set to *true*, so that it doesn't get focus. Otherwise, the edit control loses focus when the button is clicked, and there's nothing to copy.

If you have assigned a menubar to the form, you can use a menu item assigned to the menubar's *editCopyMenu* property instead of using the *copy()* method of individual objects on the form.

See also *cut()*, *editCopyMenu*, *paste()*, *undo()*

count()

Returns the number of prompts in a listbox, or the number of items in a tree.

Syntax `<oRef>.count()`

<oRef> An object reference to the listbox or tree whose items to count.

Property of ListBox, TreeView

Description Use a listbox's *count()* method when you can't anticipate the number of prompts a listbox might have at runtime. For example, when you specify "FILE *.*" for the *dataSource* property, the number of prompts depends on the number of files in the current directory.

When using an array as the *dataSource* for a listbox, you can check the array's *size* property to get the number of items.

The tree view's *count()* method returns the total number of items in the entire tree, even if they are not displayed or hidden in a collapsed subtree. Use the *visibleCount()* method to count the items that are visible in the tree view.

See also *dataSource*, *multiple*, *selected()*, *visibleCount()*

cuaTab

Determines cursor behavior when you press *Tab* while a control has focus.

Property of Browse, Editor, Grid

Description When *cuaTab* is *true*, pressing *Tab* moves to the next control in the form's tab order. When *cuaTab* is *false*, pressing *Tab* moves to the next field in a Grid or Browse object or moves the cursor to the next tab stop in an Editor object.

The same applies to pressing *Shift+Tab*, except that the movement is in reverse.

See also *tabStop*

currentColumn

The number of the column that has focus in the grid.

Property of Grid

Description Use the *currentColumn* property as an index into the *columns* array to refer to the GridColumn object that represents the column that currently has focus.

See also *columnCount*, *columns*

curSel

Determines which prompt is selected in a component.

Property of ListBox, NoteBook, TabBox

Description Use *curSel* to get or set which prompt in a ListBox, NoteBook, or TabBox is highlighted. The prompts are determined by the component's *dataSource* property. The first prompt is prompt number 1.

Assigning a value to *curSel* fires the control's *onSelChange* event, as if the change was made manually.

Example Suppose a form displays status information about an account on page 1, and background information on page 2, with a TabBox to choose pages. Clicking the Add button switches the form to page 2 so the user can add the data for the new account. By setting the *curSel* of the tabbox, the tabbox is updated, and *onSelChange* for the tabbox changes the *pageno* of the form.

```
function addButton_onClick()
    form.rowset.beginAppend()
    form.pageTabbox.curSel := 2
```

See the example for *onSelChange* for the tabbox's *onSelChange* event handler.

See also *count()*, *dataSource*, *selected()*

cut()

Cuts selected text and places it on the Windows Clipboard.

Syntax <oRef>.cut()

<oRef> An object reference to the control from which to cut the text.

Property of Browse, ComboBox, Editor, Entryfield, SpinBox

Description Use *cut()* when the user has selected text and wants to remove it from the edit control and place it on the Windows clipboard.

When calling this method from a pushbutton's *onClick* event, the pushbutton should have its *speedBar* property set to *true*, so that it doesn't get focus. Otherwise, the edit control loses focus when the button is clicked, and there's nothing to cut.

If you have assigned a menubar to the form, you can use a menu item assigned to the menubar's *editCutMenu* property instead of using the *cut()* method of individual objects on the form.

See also *copy()*, *editCutMenu*, *paste()*, *undo()*

dataLink

The Field object that is linked to the component.

Property of Many form components

Description You link a form component to a table's field by assigning a reference to the *dataLink* property of the component. The reference you assign is to the Field object that represents the field in an open query. This assignment is called *dataLinking*. When a form component and Field object are linked in this way, they are said to be *dataLinked*.

Exception: a Grid object is *dataLinked* to a rowset, not a field.

For compatibility with earlier versions of *Visual dBASE*, you may also assign the field name in a string. This technique is used for form-based data handling with tables open in work areas only.

Both field and component objects have a *value* property. (Fields in a table open in a work area do not have any properties, but they have a value; the concept is the same.) When they are *dataLinked*, changes in one object's *value* property are echoed in the other. The form component's *value* property reflects the value displayed in the component at any given moment. If the component's value is changed, it is copied into the field after the component loses focus.

The *value* property for all fields in a rowset are set when you first open a query and updated as you navigate from row to row. The *value* properties for components *dataLinked* to those fields are also updated at the same time, unless the rowset's *notifyControls* property is set to *false*. You can also force the components to be updated by calling the rowset's *refreshControls()* method, which is useful if you have set a field's *value* property through code.

Form-based data events such as *onNavigate* will not work unless the form has controls *dataLinked* to fields. For example, if you USE a table, create and open an empty Form, assign an *onNavigate* event handler, and SKIP in the table, the *onNavigate* will not fire simply because the form is open.

The *dataLink* property is similar to the *dataSource* property used for Image objects, except that data displayed through the *dataLink* property can be changed, while data displayed through the *dataSource* property is always read-only.

A component's *dataLink* is automatically set when you use the Form wizard or use a field in the Field palette.

Example The following is the statement in a .WFM file that assigns the *dataLink* property to a field in a rowset:

```
dataLink = form.student1.rowset.fields["LAST_NAME"]
```

Note that this is a link to the field object itself, not the object's *value* property.

The equivalent link for a field in a table open in a work area would look like:

```
dataLink = "STUDENT->LAST_NAME"
```

See also *dataSource* [Image], *value*

dataSource [options]

The options that are displayed in a ComboBox, ListBox, Notebook, or TabBox object.

Property of ComboBox, ListBox, Notebook, TabBox

Description Use the *dataSource* property to set the options that are displayed in a ComboBox, ListBox, Notebook, or TabBox object. The *dataSource* property for a ComboBox or ListBox is a string in one of the following five forms:

- ARRAY <array>
creates prompts from elements in an array object.
- FIELD <field name>
creates prompts from all the values in a field in a table file.
- FILE [<filename skeleton>]
creates prompts from file names in the current default directory that match the optional filename skeleton.

`dataSource [options]`

- **STRUCTURE**
creates prompts from all the field names in the currently selected table.
- **TABLES**
creates prompts from the names of all tables in the currently selected database. For the default database, this is all the .DBF and .DB files in the current directory.

For a NoteBook or TabBox, only the *ARRAY dataSource* is allowed. The *dataSource* string in general is not case-sensitive, except that if you specify a literal array, the array contents will appear as specified.

Adding elements to an array after it has been assigned as a component's *dataSource* may not automatically update the component's options. Files added to the directory after the *dataSource* property has been set to a file mask will not automatically appear either.

To update the *dataSource*, you need to reassign the *dataSource* property. In most cases, you can simply reassert the property by assigning its current value to itself. For example, if you had originally specified all the GIF files in the current directory, the *dataSource* property assignment would look like this:

```
with (this.fileCombobox)
  dataSource = "FILENAME *.GIF"
endwith
```

To update the file list when you press an Update button on your form, the button's *onClick* would look like this:

```
function updateButton_onClick()
  form.fileCombobox.dataSource += ""
```

You don't have to specify what the *dataSource* string is again, since it's already contained in the *dataSource* property. The += operator adds an empty string to reassign the value, which reasserts the *dataSource*. This makes your code easier to maintain, since the *dataSource* string is specified in only one place.

When using an array in the *dataSource* string, you can use a literal array, for example,

```
array {"Chocolate", "Strawberry", "Vanilla"}
```

Or you can use a reference to an array object, for example,

```
array aFlavors
```

If you use a reference, that array must exist at the time the *dataSource* property is assigned. Since the *dataSource* property contains that string (in this example, `array aFlavors`), if you reassert the *dataSource* property as shown above, an updated version of the named array must exist. In this example, the array *aFlavors* must be accessible in the method *updateButton_onClick()*.

For this reason, when using an updatable array as the *dataSource* property, the array is usually created as a property of the form. This makes the array equally accessible from the component that uses the array and from any other component that tries to reassert the *dataSource* property. In this example, the

array *aFlavors* would be created as a property of the form, and the *dataSource* string would contain:

```
array form.aFlavors
```

The reference *form.aFlavors* is valid from the event handler of any component on the form.

Example The following *onOpen* event handler reads the contents of a field in a table into an array to be used as the ComboBox object's options. A table of ice cream flavors has already been opened in a query named *flavors1*.

```
function flavorCombobox_onOpen()
  form.aFlavors = new Array()
  if form.flavors1.rowset.first()
    do
      form.aFlavors.add( form.flavors1.rowset.fields[ "Name" ].value )
      until not form.flavors1.rowset.next()
    endif
  this.dataSource := "array form.aFlavors"
```

Later, if someone adds a new flavor, they can add it to the array and update the ComboBox object immediately. (The flavor will be added to the table once it's approved by the flavor committee.)

```
function addFlavorButton_onClick()
  form.aFlavors.add( this.form.newFlavorText.value ) // Add new flavor
  form.flavorCombobox.dataSource += " " // Reassert by adding empty string
```

See also *curSel*, *selected()*

dataSource [Image]

The bitmap that is displayed in an Image object.

Property of Image

Description An Image object can display either a static file from disk, a resource image, or a bitmap stored in a table. Set the *dataSource* property to either one of the following:

- A string containing the word *FILENAME*, a space, and the name of a file. The string is not case-sensitive.
- A string of the form "*RESOURCE <resource id> <dll name>*", which specifies a bitmap resource and the DLL file that holds it.
- A string containing the form *BINARY*, a space, and the name of a binary field in a table open in a work area that contains bitmapped images.
- A reference to a field object in an open query that contains bitmapped images.

If you assign a field object (or a field in a work area) as the *dataSource*, the Image object will automatically update as you navigate from row to row, unless the rowset's *notifyControls* property is set to *false*.

The *dataSource* property is similar to the *dataLink* property used for Field objects, except that data displayed through the *dataLink* property can be changed, while data displayed through the *dataSource* property is always read-only.

An Image object's *dataSource* is automatically set when you use the Form Wizard or use a bitmap image field in the Field Palette.

Example The following string would set the *dataSource* of an Image object to the file LOGO.GIF in the current directory:

```
filename LOGO.GIF
```

See also *dataLink*

dataSource is also a property of the ComboBox, ListBox, NoteBook, and TabBox classes (page 15-85).

default

Determines if a pushbutton is the form's default pushbutton.

Property of PushButton

Description Use the *default* property to make a pushbutton the default pushbutton when the user submits a form by pressing *Enter*. This behavior is used primarily for dialog boxes, with the either the OK or Cancel button being the default, whichever is more appropriate. Setting the *default* property of a pushbutton to *true* gives the pushbutton a visual highlight that identifies it as the default.

Setting the *default* property to *true* causes two things to happen when the user presses *Enter* when the focus is not on a pushbutton:

- The *onClick* subroutine of the default pushbutton executes.
- The *id* property of the default pushbutton is passed to the form's *onSelection* event handler.

However, if the user clicks on any pushbutton, the *onClick* event handler of that pushbutton executes. The *id* value of that pushbutton is passed to the *onSelection* event, even if the *default* property of another pushbutton is *true*.

If you give more than one pushbutton a *default* value of *true*, the last pushbutton to get the value is the default.

See also *onClick*, *onSelection*

description

A short description for an ActiveX control.

Property of ActiveX

Description An ActiveX object's *description* property contains a short description of the ActiveX control it represents. The description is provided by the control and is read-only.

designView

Designates a .QBE query or table that is used when designing a form.

Property of Form

Description Use *designView* to facilitate creating and *dataLinking* a form that uses form-based data handling with tables in work areas. The value in *designView* is ignored at runtime. When using the data access objects, do not use *designView* or *view*.

There are two main instances in which you may want to use *designView* instead of *view*.

- If you know which tables will be open when the form is opened at runtime, use *designView* to avoid opening the tables again with *view* when the form is opened.
- If you don't know which tables will be open when the form is opened at runtime, but need certain tables open to design the form, use *designView* to automatically open the tables at design time, regardless of the tables needed at runtime.

If you specify a *view* property for a form, you should not also specify a *designView* property.

See also *dataLink*, *view*

disabledBitmap

Specifies the graphic image to display in a pushbutton when the pushbutton is disabled.

Property of PushButton

Description Use *disabledBitmap* to indicate visually when a pushbutton is not available for use. A pushbutton is disabled when its *enabled* property is set to *false*.

The *disabledBitmap* setting can take one of two forms:

- **RESOURCE** <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- **FILENAME** <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

When you specify a character string for the pushbutton with *text* and an image with *disabledBitmap*, the image is displayed with the grayed-out character string

See also class Image, *downBitmap*, *enabled*, *focusBitmap*, *textLeft*, *upBitmap*

disablePopup

Whether the tree view's popup menu is disabled.

Property of TreeView

Description The tree has a right-click popup menu that allows the user to insert, delete, and edit items. Set *disablePopup* to *true* to disable this popup menu.

Even if the popup is disabled, the user can still edit the items by clicking twice or pressing *F2*, and insert and delete items by pressing *Ins* and *Del*. Set *allowEditLabels* and *allowEditTree* to *false* to prevent these actions.

See also *allowEditLabels*, *allowEditTree*

doVerb()

Starts an action in an OLE server application.

Syntax <oRef>.doVerb(<verb expN> [, <title expC>])

<oRef> The OLE control that contains the linked or embedded object.

<verb expN> The numeric value of the OLE verb.

<title expC> An optional text string to display in the title bar of the server window.

Property of OLE

Description Use *doVerb()* to initiate an action from an OLE document stored in an OLE field and to specify what action to take.

Every OLE object accepts one or more verbs. Each verb determines which actions are taken, and each is represented by a number.

Example Most sound applications accept one of two verbs:

- 0 (Play) plays a sound stored in an OLE field.
- 1 (Edit) opens the Sound Recorder to edit the sound.

Double-clicking the OLE control plays the sound, but to edit the sound, use a button that calls the OLE control's *doVerb()* method:

```
function editSound_onClick()  
    form.soundOLE.doVerb( 1 )
```

See also *oleType*

downBitmap

Specifies the graphic image to display in a pushbutton when the user presses the button.

Property of PushButton

Description Use *downBitmap* to give visual confirmation when the user clicks a pushbutton. When the user releases the mouse button or moves the pointer off the pushbutton, the image and/or text specified by *focusBitmap* and *text* is displayed.

The *downBitmap* setting can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

When you specify a character string for the pushbutton with *text* and an image with *downBitmap*, the image is displayed with the character string.

See also class Image, *disabledBitmap*, *enabled*, *focusBitmap*, *textLeft*, *upBitmap*

dragScrollRate

The delay time between each column scroll when dragging columns.

Property of Grid

Description The *dragScrollRate* property controls how fast the grid scrolls horizontally when rearranging columns in the grid by dragging. The delay time is measured in milliseconds.

See also *allowColumnMoving*

dropDownHeight

The number of lines displayed in the list portion of the combobox.

Property of ComboBox

Description Use *dropDownHeight* to specify how much information will appear when a user drops down a list from a combo box.

See also *autoDrop*, *dropDownWidth*, *style*

dropDownWidth

The width of the list portion of the combobox.

Property of ComboBox

Description The width of the drop-down list of a combobox may be different than the width of the text entry portion. If *dropDownWidth* is zero (the default), the width of the drop-down list is sized to match the width of the control. Otherwise, the combobox's *dropDownWidth* setting is used.

The *dropDownWidth* is expressed in the form's current *metric* units (the same as the *width*).

See also *autoDrop*, *dropDownHeight*, *width*

editorControl

The editable control that comprises the body of the grid in the column.

Property of GridColumn

Description The *editorControl* property contains an object reference to the editable control in the grid column. The type of control is determined by the *editorType* property.

See also *class ColumnCheckBox*, *class ColumnComboBox*, *class ColumnEntryfield*, *class ColumnSpinBox*, *editorType*, *headingControl*

editorType

The type of editing control in the grid column.

Property of GridColumn

Description *editorType* is an enumerated property that determines the type of editable control used for the data in the column. It may have one of the following values:

| Value | Description | Control |
|-------|-------------|-----------------------------|
| 0 | Default | Depends on column data type |
| 1 | Entryfield | ColumnEntryfield |
| 2 | CheckBox | ColumnCheckBox |
| 3 | SpinBox | ColumnSpinBox |
| 4 | ComboBox | ColumnComboBox |

You may access the control through the *editorControl* property.

See also class ColumnCheckBox, class ColumnComboBox, class ColumnEntryfield, class ColumnSpinBox, *editorControl*, *headingControl*

elements

An array containing object references to all the components in a form.

Property of Form

Description The *elements* array contains an object reference for each visual component in a form. Other types of objects, like data access objects, are not included.

You can determine the number of components in the form by checking the *elements* array's *size* property. Each element in the array can be addressed by its element number or by the *name* of the component.

The *elements* array is not a member of the Array class, but rather an ObjectArray class with specific capabilities for managing a list of objects. It does not support most of the methods of the Array class. The *elements* array is not meant to be changed directly, although it is safe to scan to get the object references for the components in the form.

Example Suppose you have a form that contains a bunch of spinboxes for measurements. You want to be able to set them all to zero with the click of a button.

```
function resetButton_onClick()
  local nObj
  for nObj = 1 to form.elements.size
    if form.elements[ nObj ].className == "SPINBOX"
      form.elements[ nObj ].value := 0
    endif
  endfor
```

Compare this loop with the example for *before*.

See also *before*, *name*

enabled

Determines if an object can get focus and operate.

Property of Most form components

Description When you set the *enabled* property of an object to *true*, the user can select and use the object. When you set the *enabled* property to *false*, the object is dimmed and the user can't select or use the object. This is a visual indication that the object cannot get focus. The object is removed from the tab sequence and clicking the object has no effect.

Example Suppose you have a checkbox to echo output to a file and an entryfield for the file name. When the checkbox is unchecked, you disable the entryfield:

```
function outputFileCheckbox_onChange()  
    form.outputFileEntryfield.enabled := this.value
```

See also *visible, when*

enabled is also a property of the Timer class (page 8-12)

enableSelection

Whether the selection range is displayed in the Slider object.

Property of Slider

Description If *enableSelection* is *true*, the selection range set by the slider's *startSelection* and *endSelection* properties is displayed inside the slider as a colored area with matching tic marks. You may use the selection range to show the current or preferred range.

If *enableSelection* is *false*, the *startSelection* and *endSelection* properties have no effect.

See also *endSelection, startSelection*

endSelection

The high end of the selection range in a Slider object.

Property of Slider

Description *endSelection* contains the high value in the selection range. It should be equal to or higher than *startSelection*, and between the *rangeMin* and *rangeMax* values of the slider.

The selection is not displayed unless the slider's *enableSelection* property is *true*.

See also *enableSelection, rangeMax, rangeMin, startSelection*

ensureVisible()

Makes the tree item visible in the tree view.

Syntax `<oRef>.ensureVisible()`

<oRef> An object reference to the tree item you want to display.

Property of TreeItem

Description Use *ensureVisible()* when you want to make sure that a tree item is visible in the tree view. The tree is expanded and scrolled if necessary to make the item visible. If the tree item is already visible, nothing happens.

See also *select(), setAsFirstVisible()*

escExit

Determines if the user can close a form by pressing *Esc*.

Property of Form

Description Set *escExit* to *false* to prevent the user from closing a form by pressing *Esc*. You can verify that the user wants to close the form by using the form's *canClose* event handler. Closing a form by pressing *Esc* abandons all pending changes in the data buffer, as if *abandonRecord()* was called.

When a form is opened with *readModal()*, it returns *false* when it is closed by pressing *Esc*.

See also *canClose*

evalTags

Whether to evaluate HTML tags in the text.

Property of Editor

Description *Visual* dBASE supports common HTML formatting tags. You may choose to evaluate any tags that appear in the text of the editor and apply the formatting, or to leave the HTML tag as-is so that they can be edited.

Use the Format toolbar to format the text with HTML tags. You may also type in the tags manually, but they will not be interpreted until you toggle *evalTags* to *true*.

The editor's *evalTags* property corresponds to the Apply Formatting option in the editor's popup menu. The Format toolbar is not active if the editor's *evalTags* property is *false*.

See also *popupEnable, showFormatBar()*

expanded

Whether the tree item's children are shown or hidden.

Property of TreeItem

Description A tree item's subtree may be expanded or collapsed visually by the user, or by assigning a value to the *expanded* property. If *expanded* is *true*, the subtree is displayed. If *expanded* is *false*, the subtree is hidden.

See also *noOfChildren*

fields

Specifies the fields to display in a Browse object, and the field options to apply to each field.

Property of Browse

Description The *fields* is a string with the following the following format:

```
<field 1> [<field option list 1>] |
<calculated field 1> = <exp1> [<calculated field option list 1>]
    [, <field 2> [<field option list 2>] |
    <calculated field 2> = <exp1> [<calculated field option list 2>], ...]
```

The fields are displayed in the order they're listed, and the *<field option list>* affects the way each field is displayed. The options are as follows:

| Option | Description |
|-----------------------|---|
| \<column width> | The width of the column within which the field appears |
| \B = <exp 1>, <exp 2> | RANGE option; forces any value entered in <field 1> to fall within <exp 1> and <exp 2>, inclusive. |
| \H = <expC> | HEADER option; causes <expC> to appear above the field column in the browse, replacing the field name |

| Option | Description |
|-----------------------------------|---|
| \P = <expC> | PICTURE option; formats the field according to the <i>picture</i> template <expC> |
| \V = <condition> [\E = <expC>] | VALID option; allows a new field value to be entered only when <condition> evaluates to <i>true</i> ERROR MESSAGE option; \E = <expC> causes <expC> to appear when <condition> evaluates to <i>false</i> |

Calculated fields are read-only and are composed of an assigned field name and an expression that results in the calculated field value, for example:

```
Commission = Rate * Saleprice
```

Options for calculated fields affect the way these fields are displayed. These options are as follows:

| Option | Description |
|-----------------|--|
| \<column width> | The width of the column within which the calculated field is displayed |
| \H = <expC> | Causes <expC> to appear above the calculated field column in the browse, replacing the calculated field name |

Example The following *fields* string:

```
CONTCCITY->CITY_NAME\H="Contract City"\15,CONTCCITY->CONTACT\H="Contact Person"\25
```

Displays two fields in the browse from the Contccity table:

- The City_name field, with the heading “Contract City” in a column 15 characters wide
- The Contact field, with the heading “Contact Person” in a column 25 characters wide

See also *alias*, *dataLink*, *dataSource*, SET FIELDS

fields is also a property of the Rowset class (page 14-69)

filename

The name of the file that contains the desired report.

Property of ReportViewer

Description Set the *filename* property to the file that contains the report class definition. Reports are stored in files with a .REP extension.

The .REP file is executed when you assign the *filename* property, even if the form is not open. Normally, report parameters (if any) are assigned to the *params* array before setting the *filename* property; if they are assigned after

setting the *filename* property, you must call the ReportViewer object's *reExecute()* method to regenerate the report.

See also *params*, *reExecute()*

filename is also a property of the DataModRef class (page 14-69)

first

A reference to the first component in a containing object's z-order.

Property of Form, Container, Notebook

Description Use the *first* property to reference the first component in the containing object's z-order. For more information on component z-order within containing objects, see *before*.

If the *first* component can receive focus, it gets focus initially when you open a form. Otherwise, the next component in the z-order is tried until one is found that can receive focus.

first is a read-only property.

Example See *before*.

See also *before*

firstChild

The first child tree item.

Property of TreeItem, TreeView

Description *firstChild* is a read-only property that contains an object reference to the object's first child tree item. If the object has no children, *firstChild* is *null*.

Example The following method calls itself recursively to traverse the items in a tree, applying code to each tree item encountered:

```
function traverseTree( tRoot, kProcess )
  local t
  t = tRoot.firstChild           // Start with first item in the tree
  do while t # null              // if any, and repeat until done
    kProcess( t )                // Process each tree item
    class::traverseTree( t, kProcess ) // Call recursively for sub-tree
    t := t.nextSibling           // Next item at same level
  enddo
```


The following code calls the method to traverse the entire tree *treeview1* and display each item's basic properties:

```
function dumpTreeButton_onClick
  class::traverseTree( form.treeview1, {|t|; ? t.level, t.checked, t.text} )
```

See also *firstVisibleChild*, *nextSibling*, *prevSibling*

firstVisibleChild

The first tree item that is visible in the tree view area.

Property of TreeView

Description *firstVisibleChild* is a read-only property that contains an object reference to the first tree item that is visible at the top of the tree view area. Note that this is not necessarily a top-level tree item.

See also *firstChild*, *setAsFirstVisible()*

focus

When to give focus to the notebook tabs when they are clicked.

Property of NoteBook

Description The *focus* property determines whether the notebook tabs (or buttons if *buttons* is *true*) get focus when they are clicked. It is an enumerated property with the following possible values:

| Setting | Description |
|--------------------|-----------------------------------|
| 0 (Normal) | Tab does not focus if tab changes |
| 1 (On Button Down) | Tab always gets focus |
| 2 (Never) | Tab never gets focus |

You can always give focus to the notebook tabs with the *Tab* and *Shift+Tab* keys if its *tabStop* property is *true*. If a tab already has focus, clicking will keep focus in the tabs, no matter what the *focus* property is.

See also *buttons*, *tabStop*

focusBitmap

Specifies the graphic image to display in a pushbutton when the pushbutton has focus.

Property of PushButton

Description Use *focusBitmap* to indicate visually when a pushbutton is selected.

The *focusBitmap* setting can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

When you specify a character string for the pushbutton with *text* and an image with *focusBitmap*, the image is displayed with the character string.

See also class Image, *disabledBitmap*, *downBitmap*, *enabled*, *textLeft*, *upBitmap*

fontBold

Specifies whether the component displays its text in bold type.

Property of Many form components

Description Set *fontBold* to *true* if you want the component to display its text in boldface.

See also *fontItalic*, *fontName*, *fontSize*, *fontStrikeout*, *fontUnderline*

fontItalic

Specifies whether the component displays its text in italics.

Property of Many form components

Description Set *fontItalic* to *true* if you want the component to display its text in italics.

See also *fontBold*, *fontName*, *fontSize*, *fontStrikeout*, *fontUnderline*

fontName

The typeface of the component's text.

Property of Many form components

Description Set *fontName* to the name of the typeface you want to apply to the text in the component.

See also *fontBold, fontItalic, fontSize, fontStrikeout, fontUnderline*

fontSize

The point size of the component's text.

Property of Many form components

Description Set *fontSize* to the point size in which you want the text of the component to be displayed. There are approximately 72 points per inch. Common text is from 8 to 12 points. Default: 10.

See also *fontBold, fontItalic, fontName, fontStrikeout, fontUnderline*

fontStrikeout

Specifies whether the component displays its text struck through.

Property of Many form components

Description Set *fontStrikeout* to *true* if you want the component to display its text struck through.

See also *fontBold, fontItalic, fontName, fontSize, fontUnderline*

fontUnderline

Specifies whether the component displays its text underlined.

Property of Many form components

Description Set *fontUnderline* to *true* if you want the component to display its text underlined.

See also *fontBold, fontItalic, fontName, fontSize, fontStrikeout*

form

The form or report that contains the component.

Property of All form components.

Description A component's *form* property is a reference to the form or report that contains it. It is set automatically when the component is created and cannot be changed.

Use the *form* property in component event handlers and methods to generically refer to the object that contains the component.

In a form, a component's *form* and *parent* property refer to the same thing—the form—if the component is placed directly on the form. However, if for example you place a component in a Notebook control on a form, then the component's *parent* is the Notebook control, not the form; and the Notebook control's *parent* is the form. In a report, components are contained deeper in the object hierarchy and their *parents* are never the report.

By using the *form* property, you can immediately get back to the top of the object hierarchy and refer to its properties, events, or methods; or refer to other objects in the form or report.

In addition to the *form* property, a local variable named *form* is also created for any event handler for form, report, or a visual component. The *form* variable is not created for events in other kinds of classes. The variable and the *form* property both refer to the same object. In most event handlers, the *form* variable is used.

Example The following is an *onClick* event handler for a button that puts the form's primary rowset—a data access component—in Append mode to add a new row.

```
function addButton_onClick()
    this.form.rowset.beginAppend() // Use form property to get to other objects in form
```

The following function uses the *form* variable instead of the *form* property, and behaves identically:

```
function addButton_onClick()
    form.rowset.beginAppend()
```

See also *name, parent*

frozenColumn

The name of the column in which the cursor is confined.

Property of Browse, Grid

Description Use *frozenColumn* to confine the cursor in a single column of the grid. The other columns in the grid are displayed, but you cannot put the cursor in them.

frozenColumn expects the name of the column, which is a string. To release the cursor from the column, assign an empty string to *frozenColumn*.

See also *lockedColumns*

function

Formats text in an object.

Property of Entryfield, SpinBox, Text

Description Use a formatting *function* to format the display and entry of data. While a *picture* gives you character-by-character control, a *function* formats the entire value.

Visual dBASE recognizes the following *function* symbols:

| Symbol | Description |
|--------|---|
| (| Encloses negative numbers in parentheses. |
| ! | Converts letters to uppercase. |
| ^ | Displays numbers in exponential form. |
| \$ | Inserts a dollar sign or the symbol defined with SET CURRENCY TO instead of leading spaces. |
| A | Restricts entry to alphabetic characters. |
| B | Left-aligns a numeric entry. |
| C | Displays CR (credit) after a positive number. |
| D | Displays and accepts entry of a date in the current SET DATE format. |
| E | Displays and accepts entry of a date in European (DD/MD/YY) format. |
| I | Centers the entry. |
| J | Right-aligns the entry. |
| L | Displays numbers with leading zeros. |
| R | Inserts literal characters into the display without including them in the field. |
| T | Removes leading and trailing spaces from character values. |
| V<n> | ? and ?? command only: Wraps a character string within a width specified by <n>. |
| X | Displays DB (debit) after a negative number. |
| Z | Displays zeros as a blanks. |

Example Suppose you want a character field to be right-aligned, and all the letters typed to be converted into uppercase. Set the *function* of the control to "J!".

See also *picture*

getTextExtent()

Returns the length of a text string based on the current font settings of the Text control.

Syntax <oRef>.getTextExtent(<expC>)

<oRef> The Text object used to calculate the text size.

<expC> The string to measure

Property of Text

Description *getTextExtent()* calculates the *width* required to display <expC> in the Text object, using the object's current font settings. It returns a value in the form's current *metric*.

See also *fontBold, fontItalic, fontName, fontSize, metric, scaleFontBold, scaleFontName, scaleFontSize*

gridLineWidth

The width of the grid lines in a Grid object.

Property of Grid

Description *gridLineWidth* controls the width, in pixels, of the lines that separate the cells in a Grid object.

See also *hasColumnLines, hasRowLines*

group

Creates component groups in the form's tab order.

Property of CheckBox, PushButton, RadioButton

Description Use *group* to determine if an object is part of a group within which the user can move focus with the arrow keys. Pressing *Tab* does not move the focus within the group; it moves to the next object outside the group. All objects in a group must be of the same class.

Radiobuttons must be used in groups of two or more. Only one radio button in the group may be selected at any time.

To create a group of radio buttons, assign the same string to each RadioButton object's *group* property. The value of the string doesn't really matter; it has no other function besides grouping the radio buttons, and is not related to any other property.

You may also use *true* and *false* to create groups. Set the *group* property of the first object in the group to *true*. For all other objects that belong to the group, set *group* to *false*. The next object with a *group* setting of *true* begins another group.

Example Suppose you're creating an order entry screen. For the options "Cash, Check, or Charge," you use three RadioButton objects with the *group* "payment". For the

options “Phone, Fax, or E-mail,” you use three other `RadioButton` objects with the *group* “orderedBy”.

See also *tabStop*

handle

The Windows tree item handle

Property of `TreeItem`

Description Each tree item has an internal handle. This handle is similar to the *hWnd* handle for each control on a form. This handle may be used for low-level API calls to manipulate the individual items in the tree view

See also *hWnd*

hasButtons

Whether + and - icons are displayed for tree items that have children.

Property of `TreeView`

Description When *hasButtons* is *true*, tree items that have child items have a + or - icon to indicate whether the subtree is collapsed or expanded. Clicking the icon expands and collapses the tree.

Set *hasButtons* to *false* to prevent these icons from appearing. The user can still expand and collapse a subtree by pressing *Shift+minus* and *Shift+plus* on the numeric keypad. To prevent these keys from expanding or collapsing the tree, use the *canExpand* event.

See also *canExpand*, *hasLines*

hasColumnHeadings

Whether column headings are displayed.

Property of `Grid`

Description Set *hasColumnHeadings* to *false* to suppress column headings in a grid. If *hasIndicator* is also *false*, the grid will contain data cells only.

See also *hasIndicator*

hasColumnLines

Whether column (vertical) grid lines are displayed.

Property of Grid

Description Set *hasColumnLines* to *false* to suppress the vertical lines that separate columns in the grid.

See also *gridLineWidth*, *hasRowLines*

hasIndicator

Whether the indicator column is displayed.

Property of Grid

Description The indicator column is the left-most column in the grid, and contains an icon indicating the current column. The icon changes when a row is being appended.

Set *hasIndicator* to *false* to suppress the indicator column. If *hasColumnHeadings* is also *false*, the grid will contain data cells only.

See also *hasColumnHeadings*

hasLines

Whether lines are drawn between tree items.

Property of TreeView

Description Set *hasLines* to *false* to display items in the tree view without the normal connecting branch lines. To disable lines at the root level only, leave *hasLines* *true* and set *linesAtRoot* to *false*.

See also *hasButtons*, *linesAtRoot*

hasRowLines

Whether row (horizontal) grid lines are displayed.

Property of Grid

Description Set *hasRowLines* to *false* to suppress the horizontal lines that separate rows in the grid.

See also *gridLineWidth*, *hasColumnLines*

hasVScrollHintText

Whether the relative row count is displayed as the grid is scrolled vertically.

Property of Grid

Description When *hasVScrollHintText* is *true*, a relative row count, like “12 of 600” is continuously updated and displayed next to the vertical scrollbar as it is scrolled.

Set *hasVScrollHintText* to *false* to suppress the message.

See also *vScrollBar*

headingControl

The control that displays the grid column heading.

Property of Grid

Description The *headingControl* property contains an object reference to the *ColumnHeadingControl* object that contains the column heading. The editable control in the column is referenced through the *editorControl* property.

See also class *ColumnHeadingControl*, *editorControl*

height

The height of an object.

Property of Form components; report components: *Band*, *PageTemplate*, *StreamFrame*.

Description The unit of measurement in a form or report is controlled by its *metric* property. The default metric for forms is characters, and for reports it's twips.

See also *expandable*, *left*, *move()*, *top*, *width*

helpFile

Identifies a Windows Help file (.HLP) that contains context-sensitive Help topics.

Property of Most form objects.

Description Use *helpFile* in combination with *helpId* to provide context-sensitive help from a Windows Help file. Context-sensitive help appears for the object that has focus when the user presses *F1*, or chooses Help | Context Sensitive Help from the default menu.

After creating the Windows Help file, follow these steps:

- 1 Assign the name of the Help file to the form's *helpFile* property. This assigns the default Help file for all help topics.
- 2 Assign a context ID or index string for the form's default Help topic to the form's *helpId* property.
- 3 For individual controls that have their own help, assign the appropriate value to the control's *helpId* property.
- 4 If an individual control has a topic in a different Help file, assign that file to the control's *helpFile* property.

Warning If you assign the *F1* key as the *shortCut* key to your own menu item, pressing *F1* executes the *onClick* for that menu item; it does not display context-sensitive help. Context-sensitive help is also disabled if you assign an *onHelp* event handler.

See also *helpId*, *onHelp*

helpId

Specifies the Help context ID or index entry for an object.

Property of Most form objects

Description Use *helpId* in combination with *helpFile* to assign context-sensitive help to a control. Context-sensitive help appears for the object that has focus when the user presses *F1*, or chooses Help | Context Sensitive Help from the default menu.

Warning If you assign the *F1* key as the *shortCut* key to your own menu item, pressing *F1* executes the *onClick* for that menu item; it does not display context-sensitive help. Context-sensitive help is also disabled if you assign an *onHelp* event handler.

The *helpId* is a string that contains either:

- A context ID number, preceded by the “#” symbol, for example:

#2002

Choosing help displays the topic with that context ID number. If that context ID is not found, Help displays an error.

- A help index string, for example

Deleting accounts

Choosing help searches the index for that string. If only one topic is found that uses that index string, it is displayed. If there are multiple matches, Help displays a Topics Found dialog, letting the user choose which topic to view. If the string is not found in the index, the Help index is displayed, with the *helpId* property as the current search value.

As with *helpFile*, you may set a default *helpId* in the form. If the control that has focus does not have its own *helpId* property, the form's value is used.

See also *helpFile, onHelp*

hScrollBar

Determines when an object has a horizontal scroll bar.

Property of Grid

Description The *hScrollBar* property determines when and if a control displays a horizontal scrollbar. It may have any of four settings:

| Value | Description |
|--------------|--|
| 0 (Off) | The object never has a horizontal scroll bar. |
| 1 (On) | The object always has a horizontal scroll bar. |
| 2 (Auto) | Displays a horizontal scroll bar only when needed. |
| 3 (Disabled) | The horizontal scroll bar is visible but not usable. |

See also *vScrollBar*

hWnd

The Windows object handle for the form object.

Property of Most form objects

Description Use *hWnd* when you need to pass the handle of an form object to a Windows API function or other external DLL.

The *hWnd* of a form is the not the immediate parent of the controls in it. For that, use the *hWndClient* property.

The *hWnd* property is read-only.

See also EXTERN, *hWndClient*

hWndClient

The Windows object handle of the window that contains a form's controls.

Property of Form

Description *hWndClient* is the handle for the parent window that contains a form's controls. In contrast, *hWnd* is the handle for the form itself; the parent of the *hWndClient* window, and the grandparent of the controls.

See also *hWnd*

icon

Specifies an icon file (.ICO) or resource that displays when a form is minimized.

Property of Form

Description Use *icon* to specify an image to be used when a form is minimized. The *icon* property is a string that can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies an .ICO file.

See also *minimize*, *windowState*

id

Identifies an object with a numeric value.

Property of Most form components

Description Use *id* to give a unique supplementary identifier to an object.

In most cases, you use an object's *name* or compare object references directly to determine the identity of an object. *id* is used primarily with the *onSelection* event, which is an antiquated and rarely-used event.

The *id* property defaults to -1.

See also *onSelection*

image

Image displayed between checkbox and text label when a tree item does not have focus.

Property of TreeItem, TreeView

Description The tree view may display images to the left of the text label of each tree item. If the tree has checkboxes, the image is displayed between the checkbox and the text label.

The *image* property of the TreeView object specifies the default icon image for all tree items when they do not have focus. You may designate specific icons for each TreeItem object to override the default. Use the *selectedImage* property to specify icons for when a tree item has focus. If any individual item in the tree has its *image* or *selectedImage* property set, space is left in all tree items for an icon, even if they don't have one.

The *image* property is a string that can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies an icon resource and the DLL file that holds it.
- FILENAME <filename>
specifies an ICO icon file.

See also *imageScaleToFont*, *imageSize*, *selectedImage*

imageScaleToFont

Whether tree item images automatically scale to match the text label font height.

Property of TreeView

Description When *imageScaleToFont* is *true*, the *image*, *selectedImage*, *checkedImage*, *uncheckedImage*, and default checkbox images all scale to match the height of the text label font, controlled by the *fontName* and *fontSize* properties.

See also *checkedImage*, *image*, *imageSize*, *selectedImage*, *uncheckedImage*

imageSize

The height of tree item images in pixels.

Property of TreeView

Description *imageSize* reflects the height of the *image*, *selectedImage*, *checkedImage*, *uncheckedImage*, and default checkbox images in a tree view. You may assign a size if *imageScaleToFont* is *false*.

See also *checkedImage*, *image*, *imageScaleToFont*, *selectedImage*, *uncheckedImage*

indent

The horizontal indent, in pixels, for each level of tree items.

Property of TreeView

Description The *indent* property reflects the amount of indent, in pixels, for each level of tree items, as indicated by the tree item's *level* property. Note that indentation at the root level is also affected by branch lines at the root, which are controlled by the *hasLines* and *linesAtRoot* properties.

See also *hasLines*, *level*, *linesAtRoot*

inDesign

Whether the object was instantiated normally or by a visual designer.

Property of Form, Report

Description The Form and Report designers create a special instance of the form or report when designing them. Some actions that occur when the form or report is executed also take place when it is designed, such as the activation of queries. However, other things are missing; the Header is not executed and parameters that are usually passed are not present.

The *inDesign* property is *true* when the object was created for design instead of execution. Use it to take shortcut actions to allow the design of the object without error.

See also *onDesignOpen*

integralHeight

Whether a partial row at the bottom of the grid is displayed.

Property of Grid

Description Set *integralHeight* to *true* to show complete rows only. If *true* and the row at the bottom of the grid is clipped, the entire row is hidden. When *false* (the default), the partial row is shown.

See also *cellHeight*

isRecordChanged()

Returns a logical value that indicates whether data in the current record buffer has been modified.

Syntax `<oRef>.isRecordChanged(<keystroke expC>)`
<oRef> An object reference to the form.

Property of Form

Description Use *isRecordChanged()* for form-based data handling with tables in work areas. When using the data access objects, *isRecordChanged()* has no effect; check the rowset's *modified* property instead.

Form-based data buffering lets you manage the editing of existing records and the appending of new records. Editing changes to the current record are not written to the table until there is navigation off the record, or until *saveRecord()* is called. Each work area has its own separate edit buffer. *isRecordChanged()* returns *true* if the any fields in the currently selected work area have changed; otherwise it returns *false*.

Example The following example shows the *onClick* event handler for a Cancel button. It checks if any changes have been made to the current record. If there has, it asks for confirmation before abandoning the changes.

```
function cancelButton_onClick()
  if form.isRecordChanged()
    if msgbox( "Are you sure you want to lose these changes?", "Cancel", 4+32 ) # 6
      return // Did not choose Yes, don't cancel (or anything else)
    endif
  else
    form.abandonRecord()
```

See also *abandonRecord()*, *beginAppend()*, *saveRecord()*

key

Event fired when the user types a keystroke in a control; return value may alter or cancel the keystroke.

Parameters **<char expN>** The ASCII value of the character typed.

- <position expN> The position of the new character in the string.
- <shift expL> Whether the Shift key was down.
- <ctrl expL> Whether the Ctrl key was down.

Property of ComboBox, Entryfield

Description Use *key* to evaluate and possibly modify each character that the user enters in a control, or to perform some action for each keystroke.

The *key* event handler must return a numeric or a logical value. A numeric value is interpreted as the ASCII code of a character, which automatically replaces the character input by the user. A logical value is interpreted as a decision to accept or reject the character input by the user.

See also *keyboard()*, **KEYBOARD** (page 16-24), *picture*

keyboard()

Stuffs a character string into an edit control, simulating typed user input.

- Syntax <oRef>.keyboard(<keystroke expC>)
- <oRef> The control to receive the keystrokes.
 - <keystroke expC> A string, which may include key codes.

Property of Browse, ComboBox, Editor, Entryfield, SpinBox

Description Use *keyboard()* when you want to simulate typing keystrokes into a control. The control does not have to be the one that has focus.

Note If you want to set a value in a control, it's better to assign the *value* property directly.

Use curly braces ({ }) in <keystroke expC> to indicate cursor keys or characters by ASCII code. The following key labels may be used inside the curly braces:

| | | | |
|--------------------------|-----------------|----------------|---------------------------|
| Alt+0 through Alt+9 | Ctrl+Home | End | PgDn |
| Alt+A through Alt+Z | Ctrl+LeftArrow | Enter | PgUp |
| Backspace | Ctrl+PgDn | Esc | RightArrow |
| Backtab | Ctrl+PgUp | F1 through F10 | Shift+F1 through Shift+F9 |
| Ctrl+A through Ctrl+Z | Ctrl+RightArrow | Home | Space or Spacebar |
| Ctrl+End | Del | Ins | Tab |
| Ctrl+F1 through Ctrl+F10 | DnArrow | LeftArrow | UpArrow |

You may specify a character by its ASCII code by enclosing the value in the curly braces. If the value inside the curly braces is not a recognized key label or ASCII value, the curly braces and whatever is between them are ignored.

Calling *keyboard()* immediately fires the control's *key* event, if any. In contrast, the **KEYBOARD** command stuffs keystrokes in the main typeahead buffer. The control that has focus then picks up the keys from the typeahead buffer as usual.

See also *key*, **KEYBOARD** (page 16-24)

left

The position of the left edge of an object relative to its container.

Property of All form objects.

Description The unit of measurement in a form or report is controlled by its *metric* property. The default metric for forms is characters, and for reports it's twips.

See also *height*, *move()*, *right*, *top*, *width*

level

The tree level of the item.

Property of *TreeItem*

Description The tree item's read-only *level* property contains the nesting level of the item. The top level in the tree is level number one.

See also *firstChild*, *noOfChildren*

lineNo

The current line in an Editor object.

Property of *Editor*

Description Use *lineNo* to move the cursor to a specified line in an Editor object, or to determine what line the cursor is on.

When you set *lineNo*, the cursor moves to the beginning of the specified line.

See also *wrap*

linesAtRoot

Whether a line connects the tree items at the first level.

Property of TreeView

Description Set *linesAtRoot* to *false* to disable the connecting branch lines at the first level of the tree. To disable all branch lines, set *hasLines* to *false*.

See also *hasLines*

linkFileName

Identifies which OLE document file (if any) is linked with the current OLE field when that field is displayed in an OLE viewer.

Property of OLE

Description Use *linkFileName* to identify which OLE document file is linked with the current OLE field when that field is displayed in an OLE viewer. *linkFileName* is a read-only property.

See also *oleType*, *serverName*

lockedColumns

The number of columns that remain locked on the left side of the grid as it is scrolled horizontally.

Property of Browse, Grid

Description *lockedColumns* specifies the number of contiguous columns on the left side of the grid that do not move when you are scrolling the grid. The number must be between zero and the number of columns in the grid. When the grid is scrolled horizontally, the number of columns you locked will remain displayed in the same position. Note that if you allow column moving, the user can rearrange the columns; whatever columns end up on the left are locked.

Set *lockedColumns* to zero to unlock all columns.

See also *allowColumnMoving*, *frozenColumn*

maximize

Determines if a form can be maximized when it's not MDI.

Property of Form

Description Set *maximize* to *false* to disable the maximize icon and the Maximize option in the system menu. You must set *maximize* before you open the form. If both *maximize* and *minimize* are *false*, their icons do not appear in the title bar. If either one is *true*, they both appear, with one of them disabled.

minimize has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and has its maximize icon enabled.

See also *mdi, minimize, moveable, sizeable, sysMenu*

maxLength

Specifies the maximum number of characters allowed in an entryfield.

Property of Entryfield

Description When an Entryfield control is *dataLinked* to a field, the control automatically reads the length of the field into its *maxLength* property to set the maximum number of characters allowed.

You may set *maxLength* manually to override this setting, or when using an entryfield that is not *dataLinked* to a field. If you set the *maxLength* longer than the field, the entry is allowed, but the field value will be truncated when the value is stored in the table.

See also *picture*

mdi

Determines if a form conforms to the Multiple Document Interface (MDI) standard.

Property of Form

Description MDI is a Windows specification for opening multiple document windows within the application window. Most word processors are MDI applications. In *Visual dBASE*, all windows are forms. MDI forms have the following characteristics:

- Like application windows, they are moveable and sizeable.
- They are listed on the Windows menu of the application.
- They have a title bar, and in that title bar are the system menu, minimize, maximize, and close icons.

- When one MDI form is maximized, all other MDI forms in the same application are maximized.
- When they are active, their menus replace the menus in the main menu bar.
- They cannot be modal.
- The shortcut keystroke to close the form is *Ctrl+F4*.

The opposite of MDI is SDI (Single Document Interface), where each document is in its own application window. The Windows Explorer is an SDI application. SDI forms have the following features:

- They each have complete control over their appearance; whether they are movable, sizeable, have a title bar, or any control icons enabled.
- Each form is listed separately in the Windows Taskbar.
- Their menus appear in the form.
- They can be modeless or modal.
- They can be set to always display on top of other windows, or appear as palette windows.
- The shortcut keystroke to close the form is *Alt+F4*.

A form's *mdi* property determines whether a form is MDI or SDI. When *mdi* is *true*, the following properties are ignored:

- *maximize*
- *minimize*
- *moveable*
- *sizeable*
- *smallTitle*
- *sysMenu*
- *topMost*

Those properties default to the corresponding values for an MDI form.

Because an MDI form cannot be modal, you cannot open an MDI form with the *readModal()* method.

See also *maximize, minimize, moveable, sizeable, smallTitle, sysMenu, topMost*

memoEditor

A reference to a control's memo editor object.

Property of Entryfield

Description When editing a memo field with a *dataLinked* entryfield, you may open an Editor object in another form by double-clicking the entryfield, or by calling the entryfield's *showMemoEditor()* method.

When the editor is open, the entryfield's *memoEditor* property contains a reference to that Editor object. You may use the *memoEditor* property to manipulate the editor, or close the form that contains it. The memo editor is a standard Editor object, *anchored* to a form, so the form is the *memoEditor* object's *form* and *parent* object.

See also *showMemoEditor()*

menuFile

Assigns a menubar to a form.

Property of Form

Description Use *menuFile* to designate the menu that is displayed when the form has focus. If a form's *menuFile* property is empty, a default menu is displayed when the form has focus.

Menubars created by the Menu designer are stored in .MNU files, which is the default extension for file names assigned to *menuFile*. Assigning a file to *menuFile* executes the named file with the form as the parameter. The default bootstrap code for a .MNU file creates a menu named *root* as a child of the form. The file assigned to *menuFile* is automatically loaded as a procedure file. The procedure file's reference count is decremented when the form is released; if that was the last form that used that menu file, it is automatically unloaded.

See also class MenuBar (page 16-6), *popupMenu*

metric

The units of measurement for the position and size of an object.

Property of All form objects.

Description *metric* is an enumerated property that can have the following values:

| Value | Description |
|-------|-----------------|
| 0 | Chars (default) |
| 1 | Twips |
| 2 | Points |
| 3 | Inches |
| 4 | Centimeters |
| 5 | Millimeters |
| 6 | Pixels |

The Chars *metric* is based on the average height and width for characters in a specific base font. The base font is set through the form's *scaleFontBold*, *scaleFontName*, and *scaleFontSize* properties.

All position and size properties such as *top*, *left*, *height*, and *width* are expressed in the form's current *metric* units. The form's *metric* cannot be changed once the form is opened. If the form is closed, changing the *metric* scales the position and size properties of all the components on the form.

When a control is saved as a custom control, the *metric* of the form is saved with the control definition. This way, when the control is dropped on another form, assigning the original *metric* of the control will resize the control appropriately on the new form.

See also *scaleFontBold*, *scaleFontName*, *scaleFontSize*

minimize

Determines if a form can be minimized when it's not MDI.

Property of Form

Description Set *minimize* to *false* to disable the minimize icon and the Minimize option in the system menu. The form can still be minimized in other ways, such as choosing Minimize All Windows from the context menu of the Windows Taskbar. You must set *minimize* before you open the form. If both *maximize* and *minimize* are *false*, their icons do not appear in the title bar. If either one is *true*, they both appear, with one of them disabled.

minimize has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and has its minimize icon enabled.

See also *maximize*, *mdi*, *moveable*, *sizeable*, *sysMenu*

modify

Determines if the user can alter data in a Browse or Editor object.

Property of Browse, Editor

Description Set *modify* to *false* when you want to make a control read-only.

See also *append*














mousePointer

Changes the appearance of the mouse pointer.

Property of Most form objects

Description Use *mousePointer* to provide a visual cue when the user moves the mouse pointer over an object. For example, one pointer style might mean an object is disabled, while another pointer style might mean the object is ready for input.

You can specify the following settings for *mousePointer*:

| | | | |
|------------------|---|------------------|---|
| 0 (Default) | N/A | 7 (Size S) |  |
| 1 (Arrow) |  | 8 (Size NWSE) |  |
| 2 (Cross) |  | 9 (Size E) |  |
| 3 (I- Beam) |  | 10 (UpArrow) |  |
| 4 (Icon) |  | 11 (Wait) |  |
| 5 (Size) |  | 12 (No) |  |
| 6 (Size NESW) |  | 13 (Hand) |  |

See also *onMouseMove*, *speedTip*

move()

Repositions and resizes an object.

Syntax `<oRef>.move(<left expN> [, <top expN> [, <width expN> [, <height expN>]]])`

<oRef> The object to move or resize.

<left expN> The new *left* property.

<top expN> The new *top* property.

<width expN> The new *width* property. To change the size of the image, you must specify both the *<left expN>* and the *<top expN>*.

<height expN> The new *height* property.

Property of Most form objects

Description Use *move()* to reposition and/or resize an object in one step. You could assign the four properties directly, but doing so would require four separate steps, and the object would have to be moved and/or resized after each step. Using *move()* is faster.

If you want to resize the object without moving it, pass the current *left* and *top* properties as parameters to *move()*, along with the new width and height.

If you're using *move()* to resize an image, the object's *alignment* property should be set to either Stretch (0) or Keep Aspect Stretch (3).

Example The following are two *onClick* event handlers for buttons that zoom and unzoom a bitmap image.

```
function zoomButton_onClick()
  with form.mapImage
    move( left, top, 60, 20 )
  endwhile

function unzoomButton_onClick()
  with form.mapImage
    move( left, top, 30, 10 )
  endwhile
```

See also *alignment, height, left, onMove, top, width*

moveable

Determines if a form can be moved when it's not MDI.

Property of Form

Description Set *moveable* to *false* to prevent the form from being moved in the usual manner. Dragging the title bar has no effect, and the Move option in the system menu is disabled. However, if *sizeable* is *true*, you can move the edges of the form and in-effect move the form.

sizeable has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and is always *resizable*.

See also *mdi, onMove, sizeable, sysMenu*

multiple

Specifies whether a *ListBox* object allows selection of more than one item at a time, or whether a *NoteBook* object can have more than one row of tabs.

Property of *ListBox, Notebook*

- Description** Set *multiple* to *true* if you want to allow the selection of more than one item at one time in a `ListBox` object. The selections—whether there’s one, many, or none—are returned by the `ListBox` object’s *selected()* method.
- If a `NoteBook` object’s *multiple* property is *false*, all its tabs are displayed in a single row. If there are more tabs than will fit in the width of the notebook, scroll arrows appear. If you set *multiple* to *true*, the tabs are stacked, taking up as many rows as needed, decreasing that amount of space below the tabs. The notebook’s *visualStyle* property has more effect when *multiple* is *true*.
- See also** *selected()*, *visualStyle*

multiSelect

Whether multiple rows are visually selected.

- Property of** `Grid`
- Description** *multiSelect* is like *rowSelect*, except that you can select multiple rows. Use the *selected()* method to get the bookmarks for the rows that have been selected.
- See also** *rowSelect*, *selected()*

name

The name of the form property that is used to refer to a component.

- Property of** All form components
- Description** A component’s *name* property reflects the name of the property of the form that is used to refer to the component.
- For example, if pushing one button makes another button visible, the code looks like this:

```
function oneButton_onClick()
    form.anotherButton.visible = true
```

In *oneButton*’s event handler, *form* refers to the form that contains the button, and *anotherButton* is a property of the form that contains an object reference to the `PushButton` object *anotherButton*.

When the form was created in the Form designer, the *name* property of the `PushButton` object was set to *anotherButton*. When the form is saved into a .WFM file, the resulting code for the button looks like this:

```
this.anotherButton = new PushButton(this)
with (this.anotherButton)
    left = 10
    top = 0
```

```
width = 8
endwith
```

The name of the button is never assigned to the *name* property. Instead, the name of the button is determined by the name of the form property that contains the reference to the object. This is true for any form component that has a *name* property.

To change the name of a component in the .WFM file, change the name of the property in the initial assignment statement and the WITH statement below it.

When you read a component's *name* property, Visual dBASE returns the name of the property that the component's *parent* (the form unless the component is in a Container or NoteBook object) uses to refer to the object. The *name* is always all-uppercase.

If you assign a value to a component's *name* property, you actually change the name of the form property that contains the component's object reference. While this is allowed, there aren't many reasons you would want to do that—avoid it.

See also *elements, form, id, parent*

nativeObject

The object that contains the native properties, events, and methods of the ActiveX control.

Property of ActiveX

Description An ActiveX object's *nativeObject* property contains a reference to an object that contains the properties, events, and methods, of the actual ActiveX control. Placing the native properties in a separate object prevents name conflicts between the properties of the Visual dBASE ActiveX object, and any ActiveX control it represents.

The *nativeObject* object is empty until the *classId* property is set.

Example Suppose the ActiveX control on your form has a Launch() method. This method is called through the *nativeObject* property; for example:

```
function launchButton_onClick()
  form.someActiveX.nativeObject.Launch()
```

See also *classId*

nextObj

The object that is going to get focus during a focus change.

Property of Form

Description *nextObj* contains a reference to the control that is going to get focus during a focus change, for example when you click on another control or press *Tab* or *Shift+Tab*. If no focus change is pending, *nextObj* is *null*.

Example Use *nextObj* in *valid* event handlers to determine if validation is needed before moving to the other control. For example, the following event handler determines if the selected control is a Cancel button:

```
function somedata_valid()
  if form.nextObj == form.cancelButton
    return true // Don't bother with validation code
  else
    // Validate as usual
  endif
```

Note that this approach only works if the Cancel button is not the next button in the tab order. Otherwise pressing *Tab* would make *nextObj* the Cancel button, but simply tabbing to that button doesn't mean the user will click it. In that case, you would want to validate the data. You can remove the Cancel button from the tab order so that the user must click the button or press the accelerator key for the button, which would cancel the form.

See also *activeControl*, *before*

nextSibling

The next tree item with the same parent.

Property of TreeItem

Description The read-only *nextSibling* property contains an object reference to the next tree item (down) that has the same parent. If the tree item is the last one, *nextSibling* is *null*.

Use *nextSibling* to loop forward through the items in a tree (or subtree).

Example See the example for *firstChild* which uses *nextSibling* to loop through all the items in a tree.

See also *firstChild*, *noOfChildren*, *parent*, *prevSibling*

noOfChildren

The number of child tree items.

Property of TreeItem

Description The read-only *noOfChildren* property contains the number of children a tree item has. It goes down one level only; it does not count grandchildren.

See also *firstChild, level, parent*

oleType

Returns a number that reveals whether an OLE field is empty, contains an embedded document, or contains a link to a document file.

Property of OLE

Description Use *oleType* to determine the state of an OLE field. It is a read-only property that may have one of the following three values:

| Value | Description |
|-------|-------------------|
| 0 | Empty |
| 1 | Document link |
| 2 | Embedded document |

See also *linkFileName, serverName*

onAppend

Event fired when a record is added to a table.

Parameters none

Property of Browse, Form

Description The form's (or browse's) *onAppend* event is used mainly for form-based data handling with tables in work areas. It also fires when the *onAppend* event of the form's primary rowset fires.

Use *onAppend* to make your application respond each time the user adds a record. *onAppend* fires after the new record is saved. If the record is saved because the user navigated to another record, *onAppend* fires after arriving at the other record, before *onNavigate*.

onAppend will not work unless the form is open and has controls *dataLinked* to fields. For example, if you USE a table, create and open an empty Form, assign an *onAppend* event handler, and APPEND BLANK, the *onAppend* will not fire simply because the form is open.

See also *onAppend, onChange, onNavigate*

onAppend is also an event of the Rowset class (page 14-96).

onChange

When the contents of the component have been changed.

Parameters none

Property of Many form objects

Description *onChange* fires when the user changes data, which includes the following actions:

- Inserts or removes a checkmark in a checkbox
- Selects a different radio button in the radiobutton group
- Selects a different item in a tree
- Changes a value in an entryfield
- Changes a value in the text box portion of a combo box or a spin box
- Clicks the spinner on a spinbox
- Moves the scroll thumb in a scrollbar object
- Changes a value in a field and moves to another row in a browse

The *onChange* event of an OLE object fires each time the record pointer moves from one record to another. The *onChange* event of a form fires after moving to another record, if the previous record was changed, but only if the form is open and has controls *dataLinked* to fields.

Example The following *onChange* event handler for a radiobutton sets the *wrap* property of an editor control on the same form to match:

```
function wrapCheckbox_onChange()
  form.editor1.wrap := this.value
```

In this example, three radiobuttons, whose *text* properties happen to match the names of the index tags of the current table, use the same *onChange* event handler. When a different radiobutton is selected, *onChange* fires twice: once for the radiobutton that was deselected, and once for the button that is selected. The *value* of the radiobutton is checked to set the index for the radiobutton that was clicked.

```
function indexRadios_onChange
  if this.value
    set order to ( this.text )
  endif
```

See also *valid*

onChange is also an event of the Field class (page 14-97).

onChar

Event fired when a “printable” key or key combination is pressed while the control has focus.

- Parameters** **<char expN>** The ASCII code of the key or key combination
- <repeat count expN>** The number of times the keystroke is repeated based on how long the key is held down.
- <key data expN>** A double-byte value that contains information about the key released, stored in separate bit fields. The bits of this parameter contain the following information:

| Bit numbers | Description |
|-------------|---|
| 0–7 | Keyboard scan code (OEM dependent) |
| 8 | Extended key (1 if <i>true</i>) such as right <i>Alt</i> and <i>Ctrl</i> , and numbers on numeric keypad |
| 9–12 | Reserved |
| 13 | Context code: 1 if <i>Alt</i> was pressed during keystroke |
| 14 | Previous key state: 1 if key was held down |
| 15 | Transition state: 1 if key is being released, 0 if key is pressed (usually 0) |

Property of PaintBox

Description If you have created a PaintBox object to develop a custom edit control, use *onChar* to do something when the object has focus and the user presses a key; that is, when they type a normal character.

onChar is similar to *onKeyDown*. However, *onChar* doesn't fire for non-printable keys, such *Caps Lock*, while *onKeyDown* fires for any key pressed.

See also *onKeyDown*, *onKeyUp*

onCheckBoxClick

Event fired after a checkbox in a tree item is clicked.

Parameters none

Property of TreeView

Description *onCheckBoxClick* fires after the user has clicked a tree item's checkbox. Check the *checked* property of the tree's currently *selected* tree item to see whether the checkbox is now checked or not.

See also *checkboxes*, *checked*

onClick

After a button is clicked.

Parameters none

Property of Menu, PushButton

Description Use *onClick* to execute code when you click a button or choose a menu item.

Example The following *onClick* event handler for an Add button puts the form's primary rowset in Append mode to allow entry of a new row.

```
function newButton_onClick()
    form.rowset.beginAppend()
```

The *onClick* of a menu item often executes a method of the form. In this example, a menu item calls the *onClick* event handler of the equivalent button on the form.

```
function addMenu_onClick()
    form.newButton.onClick()
```

This example demonstrates a Next Page button in a report.

```
function nextPageButton_onClick()
    form.endPage := ++form.startPage
    form.render()
```

See also *onChange*

onClose

After the form has been closed.

Parameters none

Property of Form, OLE

Description Use *onClose* to perform any extra manual cleanup, if necessary, when you close a form or report. Normally, *Visual* dBASE automatically discards anything in the form when you close it. You might use *onClose* if you created an object in the *onOpen* that you did not bind to the form or report.

Before executing the *onClose* event handler, *Visual* dBASE does the following:

- 1 Executes the *canClose* event handler (if any) of the form. If it returns *false*, the form does not close; nothing further happens.
- 2 Executes the *valid* event handler (if any) of the object that currently has input focus. If it returns a value of *false*, the form does not close; nothing further happens.
- 3 Executes the *onLostFocus* event handler (if any) of the object that currently has input focus.
- 4 Executes the *onLostFocus* event handler (if any) of the form.

The *onClose* events of an OLE controls execute when the parent form is closed, after the *onClose* of that form.

See also *close()*, *onOpen*

onClose is also an event of the Query and StoredProc classes (page 14-97).

onDesignOpen

After a form or component is loaded in the Form designer.

- Parameters** *<from palette expL>* Whether the component was added from the palette. If *true*, the component has just been created. If *false*, the component has been reloaded into the Form designer (when editing an existing form).
- Property of** All form objects.
- Description** Use *onDesignOpen* to execute code whenever a form or component is loaded into the Form Designer, either when it is first created (for components only), or when it is subsequently loaded into the Form Designer.

onEditLabel

Event fired after the text label in a tree item is edited; may optionally return a different label value to save.

- Parameters** *<text expC>* The not-yet-posted text label.
- Property of** TreeView
- Description** *onEditLabel* fires after the user has pressed *Enter* or clicked away to submit their label change.
- If the *onEditLabel* event handler returns a character string, that string is saved as the *text* property of the tree item instead of *<text expC>*. If the event handler returns any other type, or returns nothing, *<text expC>* is used as-is.

Example The following *onEditLabel* event handler converts all labels to camel-case by converting the string to proper-case and removing all spaces:

```
function TREEVIEW1_onEditLabel(text)
    local cRet
    cRet = proper( text )
    do while " " $ cRet
        cRet := stuff( cRet, at( " ", cRet ), 1, "" )
    enddo
    return cRet
```

See also *allowEditLabels*, *canEditLabel*

onExpand

Event fired after a checkbox in a tree item is clicked.

Parameters none

Property of TreeView

Description *onExpand* fires after the user has expanded or collapsed a tree item's subtree through the user interface, usually by clicking the + or - icon. Check the *expanded* property of the tree's currently *selected* tree item to see whether the subtree is now expanded or not.

See also *canExpand*, *hasButtons*

onFormSize

Event fired whenever the parent form of a PaintBox object is resized.

Parameters none

Property of PaintBox

Description *onFormSize* fires whenever the parent form of a paintbox object is resized, restored, or maximized. This lets you reposition or resize the object based on the form's new size. For example, you could use *onFormSize* to implement behavior similar to the *anchor* property, keeping the bottom of the PaintBox object positioned at the bottom of the form.

onFormSize is similar to *onPaint*. However, *onPaint* is triggered when the parent form is opened and when items covering the paintbox object are moved away, while *onFormSize* is not.

See also *onPaint*

onGotFocus

Event fired when a component gains focus.

Parameters none

Property of Form and all form components that get focus

Description *onGotFocus* fires whenever the form or component gains focus.

See also *onLostFocus*

onHelp

Event fired when the user presses *F1* while an object has focus, instead of context-sensitive help.

Parameters none

Property of Most form objects

Description Use *onHelp* to override the built-in context-sensitive help system (based on the *helpFile* and *helpId* properties) and execute your own code when the user presses *F1*. For example, you might use *onHelp* if you have not yet written a Help file, if the help you want to give is very simple, or you want *Visual dBASE* to drive the help (as you would with an online assistant).

As with context-sensitive help, if you assign an *onHelp* event handler to a form, that is the default handler for all the controls in the form. Each control may then have its own *onHelp* if necessary; otherwise, the form's *onHelp* is fired when the user presses *F1*.

See also *helpFile*, *helpId*

onKeyDown

Event fired when any key is pressed while the control has focus.

Parameters **<virtual key expN>** The Windows virtual-key code of the key released. For a list of virtual-key codes, see the Win32 Programmer's Reference (search for "Virtual-key Codes" in the index).

<repeat count expN> The number of times the keystroke is repeated based on how long the key is held down.

<key data expN> A double-byte value that contains information about the key released, stored in separate bits. The bits of this parameter contain the following information:

| Bit numbers | Description |
|-------------|---|
| 0-7 | Keyboard scan code (OEM dependent) |
| 8 | Extended key (1 if <i>true</i>) such as right <i>Alt</i> and <i>Ctrl</i> , and numbers on numeric keypad |
| 9-12 | Reserved |
| 13 | Context code: always 0 for <i>onKeyDown</i> |
| 14 | Previous key state: 1 if key was held down |
| 15 | Transition state: always 0 for <i>onKeyDown</i> |

Property of PaintBox

Description Use *onKeyDown* and *onKeyUp* for complete control of keystrokes while a *PaintBox* object has focus. Each key is treated separately, with none of their normal relationships, and pressing and releasing the key are two separate actions. For example, holding down *Shift* and pressing the *A* key is normally interpreted as a capital “A”. With *onKeyDown* and *onKeyUp*:

- *onKeyDown* fires when the *Shift* is pressed
- *onKeyDown* continues to fire as the *Shift* is held down
- *onKeyDown* fires when the *A* is pressed
- *onKeyDown* continues to fire if the *A* is held down
- *onKeyUp* fires when the *A* is released
- Releasing a key stops the repeat action of *onKeyDown* for the *Shift* key
- *onKeyUp* fires when the *Shift* is released

To know that this was a capital “A”, you would have to keep track of the fact that the *Shift* key was down when the *A* key was pressed.

A similar event, *onChar* is used when you want the *PaintBox* to respond to normal “printable” characters. For example, *onChar* would fire just once, getting the ASCII code for the capital “A”. *onKeyDown* and *onKeyUp* deal with Windows virtual-key codes, which are not the same as the key character value in many cases.

See also *onChar*, *onKeyUp*

onKeyUp

Event fired when any key is released while the control has focus.

Parameters **<virtual key expN>** The Windows virtual-key code of the key released. For a list of virtual-key codes, see the Win32 Programmer’s Reference (search for “Virtual-key Codes” in the index).

<repeat count expN> The number of times the keystroke is repeated based on how long the key is held down; always 1 for *onKeyUp*.

<key data expN> A double-byte value that contains information about the key released, stored in separate bits. The bits of this parameter contain the following information:

| Bit numbers | Description |
|-------------|---|
| 0–7 | Keyboard scan code (OEM dependent) |
| 8 | Extended key (1 if <i>true</i>) such as right <i>Alt</i> and <i>Ctrl</i> , and numbers on numeric keypad |
| 9–12 | Reserved |
| 13 | Context code: always 0 for <i>onKeyUp</i> |
| 14 | Previous key state: always 1 for <i>onKeyUp</i> |
| 15 | Transition state: always 1 for <i>onKeyUp</i> |

Property of PaintBox

Description Use *onKeyUp* with *onKeyDown* for complete control of keystrokes while a PaintBox object has focus. For more information, see *onKeyDown*.

See also *onChar*, *onKeyDown*

onLeftDbIcIck

Event fired when the user double-clicks a form or an object.

Parameters **<flags expN>** A single-byte value that tells you which other keys and mouse buttons were pressed when the user double-clicked the button.

<col expN> The horizontal position of the mouse when the user double-clicked the button.

<row expN> The vertical position of the mouse when the user double-clicked the button.

Property of Most form objects

Description Use *onLeftDbIcIck* to perform an action when the user double-clicks with the left mouse button. *onLeftDbIcIck* can also trap *Shift*, *Ctrl*, middle mouse button, or right mouse button presses if they occur at the same time the user double-clicks the button.

The state of each of the three mouse buttons and the *Shift* and *Ctrl* keys is stored in a separate bit in the *<flags expN>* parameter, as follows:

| Bit number | Flag for |
|------------|---------------------|
| 0 | Left mouse button |
| 1 | Right mouse button |
| 2 | <i>Shift</i> |
| 3 | <i>Ctrl</i> |
| 4 | Middle mouse button |

To check if the key or button was down, use the `BITSET()` function with the *<flags expN>* as the first parameter, and corresponding the bit number as the second parameter. `BITSET()` will return *true* if the key or button was down, and *false* if it was not.

The *<col expN>* and *<row expN>* parameters contain values that are relative to the object that fired the event. For example, the upper left corner of a button is always row 0, column 0, even if that button is in the bottom corner of the form.

All other *onLeft-*, *onRight-*, and *onMiddle-* mouse events operate in the same way, and receive the same parameters.

When you double-click a button, its button events fire in the following order:

- 1 mouse down
- 2 mouse up
- 3 mouse double click
- 4 mouse up

Example Suppose you have a secret function that you want to activate by double-clicking a Text object while holding down the *Shift*, *Ctrl*, and right mouse button:

```
function someText_onLeftDbClick( flags, col, row )
    if bitset( flags, 1 ) and bitset( flags, 2 ) and bitset( flags, 3 )
        // Do secret function
    endif
```

See also BITSET() (page ???), *onLeftMouseDown*, *onLeftMouseUp*, *onMiddleDbClick*, *onRightDbClick*

onLeftMouseDown

Event fired when the user presses the left mouse button while the pointer is over a form or an object.

Description Use *onLeftMouseDown* to perform an action when the user presses the left mouse button. Other than the initiating action, this event is identical to *onLeftDbClick*.

See also *onLeftDbClick*, *onLeftMouseUp*, *onMiddleMouseDown*, *onRightMouseDown*

onLeftMouseUp

Event fired when the user releases the left mouse button while the pointer is over a form or an object.

Description Use *onLeftMouseUp* to perform an action when the user releases the left mouse button. Other than the initiating action, this event is identical to *onLeftDbClick*.

See also *onLeftDbClick*, *onLeftMouseDown*, *onMiddleMouseUp*, *onRightMouseUp*

onLostFocus

Event that fires when a component loses focus.

Parameters none

Property of Form and all form components that get focus

- Description** *onLostFocus* fires whenever the component loses focus.
onLostFocus differs from *valid*, which specifies a condition that must evaluate to *true* before the object can lose focus.
- See also** *onChange, onGotFocus, valid*

onMiddleDbIcIck

Event fired when the user double-clicks with the middle mouse button while the pointer is on a form or an object.

- Description** Use *onMiddleDbIcIck* to perform an action when the user double-clicks with the middle mouse button. Other than the initiating action, this event is identical to *onLeftDbIcIck*.
- See also** *onLeftDbIcIck, onMiddleMouseDwn, onMiddleMouseUp, onRightDbIcIck*

onMiddleMouseDwn

Event fired when the user presses the middle mouse button while the pointer is over a form or an object.

- Description** Use *onMiddleMouseDwn* to perform an action when the user presses the middle mouse button. Other than the initiating action, this event is identical to *onLeftDbIcIck*.
- See also** *onLeftDbIcIck, onLeftMouseDwn, onMiddleMouseDwn, onMiddleMouseUp, onRightMouseDwn*

onMiddleMouseUp

Event fired when the user releases the middle mouse button while the pointer is over a form or an object.

- Description** Use *onMiddleMouseUp* to perform an action when the user releases the middle mouse button. Other than the initiating action, this event is identical to *onLeftDbIcIck*.
- See also** *onLeftDbIcIck, onLeftMouseUp, onMiddleDbIcIck, onMiddleMouseDwn, onRightMouseUp*

onMouseMove

Event fired when the user moves the mouse over a form or control.

- Parameters**
- <flags expN>** A single-byte value that tells you which keys and mouse buttons were pressed while the user moved the mouse. You can interpret this value with the `BITSET()` function, which examines individual bits in numeric values. For more information, see *onLeftDbClick*.
 - <col expN>** The horizontal position of the mouse inside the bounds of the object.
 - <row expN>** The vertical position of the mouse inside the bounds of the object.

Property of Most form objects

Description Use *onMouseMove* to perform actions when the user moves the mouse over an object. A control's *onMouseMove* fires when the mouse moves over that control. The form's *onMouseMove* fires when the mouse moves over an area of the form where there is no control.

The *<col expN>* and *<row expN>* parameters contain values that are relative to the object that fired the event. For example, the upper left corner of a button is always row 0, column 0, even if that button is in the bottom corner of the form.

Example The following *onMouseMove* event handler makes a button harder to click.

```
function PUSHBUTTON1_onMouseMove(flags, col, row)
    static jump = {|| 2 + rand() * 2}
    local nLeft, nTop
    // Calculate horizontal movement
    nLeft = this.left + jump() * sign( rand() - 0.5 )
    if nLeft < 0
        nLeft = form.width - this.width - jump()
    elseif nLeft > form.width - this.width
        nLeft = jump()
    endif
    // Calculate vertical movement
    if row < this.height / 2
        nTop = this.top + jump() * row
    else
        nTop = this.top - jump() * ( this.height - row )
    endif
    if nTop < 0
        nTop = form.height - this.height - jump()
    elseif nTop > form.height - this.height
        nTop = jump()
    endif
    this.move( nLeft, nTop )
```

See also *onLeftDbClick*, *speedTip*

onMove

Event fired after the user moves the form.

Parameters **<left expN>** The new horizontal position of the upper left corner of the form's client area.

<top expN> The new vertical position of the upper left corner of the form's client area.

Property of Form

Description Use *onMove* to perform actions automatically when a form is moved.

The two parameters passed to the event handler indicate the new position of the client area of the form, the area below the title bar and inside the edges of the form. To get the new position of the entire form, check the form's *left* and *top* properties directly.

See also *onSize*

onNavigate

Event fired when the record pointer in a table in a work area is moved.

Parameters **<workarea expN>** The work area number where the navigation took place.

Property of Browse, Form

Description The form's (or browse's) *onNavigate* event is used mainly for form-based data handling with tables in work areas. It also fires when there is navigation in the form's primary rowset.

Use *onNavigate* to make your application respond each time the user moves from one record to another.

When using tables in work areas, *onNavigate* will not fire unless the form is open and has controls *dataLinked* to fields. For example, if you USE a table, create and open an empty Form, assign an *onNavigate* event handler, and SKIP in the table, the *onNavigate* will not fire simply because the form is open.

When navigating in the form's primary rowset, the form's *onNavigate* fires after the rowset's *onNavigate*, and the *<workarea expN>* parameter is zero.

Example The following *onNavigate* event handler calls a custom form method called *refreshUnlinked()* method to update components on the form that are not *dataLinked* directly to fields so that the components contain the correct information as the user moves from record to record. It does this only if the

navigation occurred in the main work area, the one that's usually selected; not in another work area that, for example, has a lookup table.

```
function Form_onNavigate( nWorkArea )
  if nWorkArea == workarea()
    form.refreshUnlinked()
  endif
```

For example, you might display the current record number in a Text component, which does not get automatically updated.

```
function refreshUnlinked()
  form.recnoText.text := "" + recno() + "/" + reccount()
```

See also *canNavigate*, *rowset*

onNavigate is also an event of the Rowset class (page 14-99).

onOpen

After the form or component has been opened.

Parameters none

Property of All form objects.

Description *onOpen* events fire after a form has been opened. First the *onOpen* event for the form or report fires, then the *onOpen* for each component, if one has been assigned. Use *onOpen* to set up items in the form that cannot be set in the Form designer.

Example The following example is the *onOpen* event handler for the FISH.WFM form. It assigns the popup in FISH.POP as the popup menu for the form. There is no way to set up a popup menu directly in the Form designer.

```
function Form_onOpen
  set procedure to FISH.POP additive
  this.popupMenu := new fishPopup(this,"POPUP")
```

In this example, a CheckBox control toggles the *wrap* property of an editor on the same form. It reads the *wrap* property of the editor when the form is opened. This way, the two properties are in sync, and you don't have to set the checkbox's *value* property whenever you change the editor's *wrap* property.

```
function wrapCheckbox_onOpen()
  this.value := form.editor1.wrap
```

See also *onClose*

onOpen is also an event of the Query and StoredProc classes (page 14-100).

onPaint

Event fired whenever a PaintBox object needs to be redrawn.

Parameters none

Property of PaintBox

Description *onPaint* is called whenever a PaintBox object needs to be redrawn. Events that trigger *onPaint* include:

- the parent form is opened
- a minimized parent form is restored or maximized
- a window or object which has been covering the paintbox object is moved away

See also *onFormSize*

onRightDbClick

Event fired when the user double-clicks with the right mouse button while the pointer is on a form or an object.

Description Use *onRightDbClick* to perform an action when the user double-clicks with the right mouse button. Other than the initiating action, this event is identical to *onLeftDbClick*.

This event will not fire for the form if you have a popup menu assigned to the form's *popupMenu* property.

See also *onLeftDbClick*, *onMiddleDbClick*, *onRightMouseDown*, *onRightMouseUp*, *popupMenu*

onRightMouseDown

Event fired when the user presses the right mouse button while the pointer is on a form or an object.

Description Use *onRightMouseDown* to perform an action when the user presses the right mouse button. Other than the initiating action, this event is identical to *onLeftDbClick*.

If the form has a popup menu assigned to its *popupMenu* property, the sequence of events when you right-click the form is:

- 1 The popup menu appears
- 2 After making a choice or dismissing the menu, the *onRightMouseDown* event fires.

3 If a choice was made from the popup menu, its *onClick* fires.

See also *onLeftDbClick, onLeftMouseDown, onMiddleMouseDown, onRightDbClick, onRightMouseUp, popupMenu*

onRightMouseUp

Event fired when the user releases the right mouse button while the pointer is on a form or an object.

Description Use *onRightMouseUp* to perform an action when the user releases the right mouse button. Other than the initiating action, this event is identical to *onLeftDbClick*.

This event will not fire for the form if you have a popup menu assigned to the form's *popupMenu* property.

See also *onLeftDbClick, onLeftMouseUp, onMiddleMouseUp, onRightDbClick, onRightMouseDown, popupMenu*

onSelChange

Event fired when a selection is changed in a component.

Parameters none

Property of Grid, ListBox, NoteBook, TabBox

Description *onSelChange* is used in components that have a specific set of options to choose from; the tabs in a NoteBook or TabBox, the items in a ListBox, and the rows in a Grid. It fires whenever the focus changes from one option to another.

Example The following is the standard *onSelChange* event handler for a tabbox used to display the pages of a multi-page form.

```
function pageTabbox_onSelChange()  
  form.pageno := this.curSel
```

The *pageno* of the form is changed to the corresponding tab number, reflected in the tabbox's *curSel* property.

See also *curSel, onChange, selected()*

onSelection

Event fired when the user submits a form.

Parameters **<id expN>** The *id* property of the control that had focus when the form was submitted.

Property of Form

Description A form is submitted when the user either:

- Presses *Enter* when the form has focus and no Editor, Grid, or Browse object has focus.
- Presses *Spacebar* when a pushbutton has focus.
- Clicks a pushbutton.

The concept of submitting a form is antiquated and rarely used. You should code the *onClick* event handler for a specific pushbutton, and set the *default* property of a pushbutton to *true* so that pushbutton is clicked when *Enter* is pressed.

See also *default, id, onClick*

onSize

Event fired after the user resizes a form.

Parameters **<nType>** A number that indicates how the user resized the form. It has three possible values:

| Value | Description |
|-------|--|
| 0 | The user resized the form with the mouse or restored the form from a maximized or minimized condition. |
| 1 | The user minimized the form. |
| 2 | The user maximized the form. |

<width> The new width of the client area of the form.

<height> The new height of the client area of the form.

Property of Form

Description Use *onSize* to perform actions when the user resizes a form.

The two parameters passed to the event handler indicate the new size of the client area of the form, the area below the title bar and inside the edges of the form. To get the new size of the entire form, check the form's *width* and *height* properties directly.

Some controls have an *onFormSize* event that fires when the form is resized; any actions specific to those controls should be handled with that event. Other controls have an *anchor* property, and are resized to automatically.

Example Suppose you have a form with a horizontal Line object that goes all the way across the form. You could handle form resizing by making the line excessively long, or you can resize it with the form's *onSize* event handler:

```
function form_onSize( nSizeType, nWidth, nHeight )
    form.line1.right := nWidth
    return
```

See also *anchor, mdi, onFormSize, sizeable, windowState*

open()

Opens a form.

Syntax `<oRef>.open()`
<oRef> The form to open.

Property of Form

Description Use *open()* to open a form.
 The form you open with *open()* is *modeless*, and has the following characteristics:

- While the form is open, focus can be transferred to other forms.
- Execution of the routine that opened the form continues after the form is opened and active.

See also *close(), onOpen, readModal()*
open() is also a method of the File class (page 11-27).

pageCount()

Returns the highest numbered page used in a form.

Syntax `<oRef>.pageCount()`
<oRef> An object reference to the form you want to check.

Property of Form

Description *pageCount()* returns the highest *pageno* used by the controls in the form. (There are actually no pages or page objects in a form.)
 In most cases, you know how many pages there are in a form because you decide on which pages to place the form's controls. Use *pageCount()* if you do

not want to keep track of the highest page manually, or if the form creates objects on different pages dynamically.

Example The following is the *onClick* event handler for a button that displays the next page on a form. If the last page is displayed, the button is disabled.

```
function nextButton_onClick()
  if ++form.pageno >= form.pageCount()    // Goto next page, and if it's the last page
    this.visible := false                  // You can't go any further
  endif
```

See also *pageno*

pageno

The page of the form on which a component appears, or the form's active page.

Property of All form objects.

Description All form objects have a *pageno* property that can be between 0 and 255. The form's *pageno* property indicates the form's active page, the one it is displaying. All the components in the form that have the same *pageno* as the form are displayed on that "page"; the rest are hidden. There are no actual pages or page objects to manage.

When a form's *pageno* property is zero, all components are displayed. If a component's *pageno* property is zero, it appears on all pages. For example, a company logo that appears on every page can be placed on page zero.

The *pageno* property can be changed at any time. Changing a form's *pageno* displays another page of the form. Changing a component's *pageno* moves that component to that page.

In addition to the *pageno* property, you can set a component's *visible* property if you want to hide or display it under particular circumstances.

Example Suppose you have a 12-page survey form. There are buttons to move to the next and previous pages. These buttons are on page zero, so that they appear on every page. The Previous button has its *visible* property initially set to *false*, because the form starts on page 1 and there is no previous page to go to. When you get to page 12, you want to hide the Next button, since there are no more pages.

The *onClick* event handlers for the two buttons would look like:

```
function nextPageButton_onClick()
  if ++form.pageno >= 12                // Goto next page, and if it's the last page
    this.visible := false                // You can't go any further
  endif
  form.prevButton.visible := true        // Always make previous button visible
function prevPageButton_onClick()
```

```

if --form.pageno <= 1          // Goto previous page, and if it's the first page
    this.visible := false      // You can't go any further
endif
form.nextButton.visible := true // Always make next button visible

```

The prefix increment and decrement operators are used so that the page number is changed before it is tested. It's not necessary to see if you should be allowed to change pages; if the button is visible, you can go in that direction. Finally, going in one direction always makes it possible to go the other way.

See also *visible*

params

Parameters passed to a report.

Property of ReportViewer

Description The *params* property contains an associative array that contains parameter names and values, if any, that are passed to the specified .REP file. The parameters are passed in the order they are assigned to the *params* property. Normally, report parameters are assigned to the *params* array before setting the *filename* property; if they are assigned after setting the *filename* property, you must call the ReportViewer object's *reExecute()* method to regenerate the report.

Example Suppose you have a form that uses a ReportViewer to preview a report of the grade point average of all students. You include the option of showing students in a specific grade, by using a SpinBox for the grade number, and a CheckBox to enable or disable the grade restriction. From the CheckBox or SpinBox components' *onChange* event, you call the following form method to redisplay the report with the latest options:

```

function viewReport()
    if form.gradeCheckbox.value    // Use grade
        form.reportViewer1.params[ "grade" ] = form.gradeSpinbox.value
    else
        // No grade, remove the element
        form.reportViewer1.params.removeAll() // Only one element, so just removeAll()
    endif
    form.reportViewer1.reExecute() // Re-execute report with new parameters

```

The .REP file has the following statements in the Header:

```

if argcount() >= 1
    local r
    r = new GPAREport()
    r.streamSource1.rowset.filter := "GRADE = " + argvector(1)
    r.render()
    return
endif

```

`paste()`

If a parameter is passed, the report's StreamSource object's rowset's *filter* property is set so that only the specified grade is shown. The report is rendered, and the RETURN statement prevents the execution of the standard report bootstrap code.

See also *filename*
params is also a property of the Query and StoredProc classes (page 14-103).

paste()

Copies text from the Windows clipboard to the control.

Syntax `<oRef>.paste()`
<oRef> An object reference to the control in which to paste the text.

Property of Browse, ComboBox, Editor, Entryfield, SpinBox

Description Use *paste()* when the user wants to copy text from the Windows clipboard into the specified control.

If you have assigned a menubar to the form, you can use a menu item assigned to the menubar's *editPasteMenu* property instead of using the *paste()* method of individual objects on the form.

See also *copy()*, *cut()*, *editPasteMenu*, *undo()*

patternStyle

Specifies the background hatching pattern.

Property of Rectangle

Description Use *patternStyle* to select a background hatching pattern for a Rectangle object.
You can specify the following settings for *patternStyle*

Table 15.4 Fill patterns








| Value | Description | Example |
|-------|-------------|---|
| 0 | Solid |  |
| 1 | BDiagonal |  |
| 2 | Cross |  |
| 3 | Diagcross |  |
| 4 | FDiagonal |  |

Table 15.4 Fill patterns

| Value | Description | Example |
|-------|-------------|---|
| 5 | Horizontal |  |
| 6 | Vertical |  |

The color of the pattern is determined by the foreground and background colors specified in the rectangle’s *colorNormal* property.

See also *colorNormal*






pen

Specifies the pattern of a Line object.

Property of Line

Description Use *pen* to control the appearance of a Line object when its *width* is 1.
You can specify any of five settings for *pen*:

Table 15.5 Pen patterns

| Value | Description | Example |
|-------|-------------|--|
| 0 | Solid |  |
| 1 | Dash |  |
| 2 | Dot |  |
| 3 | Dash Dot |  |
| 4 | DashDotDot |  |

If the line’s *width* is greater than 1, the *pen* property is ignored and the line is always drawn with a solid pen.

See also *width*

penStyle

Specifies the type of line to be used as the border of a Shape object.

Property of Shape

Description Use *penStyle* to control the appearance of the border of a Shape object when the *penWidth* is 1.

You can specify any of five settings for *penStyle*:

| Value | Description | Example |
|-------|-------------|-----------|
| 0 | Solid | _____ |
| 1 | Dash | - - - - - |
| 2 | Dot | |
| 3 | Dash Dot | - . - . - |
| 4 | DashDotDot | - |

If *penWidth* is greater than 1, the *penStyle* property is ignored and the outline is always drawn with a solid pen.

See also *penWidth, shapeStyle*

penWidth

Specifies the width in pixels of the line used as the border of a Shape object.

Property of Shape

Description Use *penWidth* to specify the thickness of the line used to border a shape object. If you set *penWidth* to a value greater than 1, then *penStyle* is always treated as 0 (Solid).

See also *penStyle, shapeStyle*

phoneticLink

Contains a reference to the control that mirrors the phonetic equivalent of the current value.

Property of Entryfield

Description *phoneticLink* is used in double-byte operating systems to store the single-byte phonetic representation of a value in an entryfield. It contains an object reference or the *name* of the mirror Entryfield object.

picture

A formatting template for text.

Property of Entryfield, SpinBox, Text

Description Specify the *picture* property with a character string called a *template*. A template can consist of

- Picture template characters, which represent and modify individual characters in the text string.
- Function symbols, which usually modify the entire text string. (For information on function symbols, see the *function* property.)
- Literal characters, which are inserted into the text string.

Here are the *picture* template characters:

| | |
|-----------|--|
| 9 | Restricts entry of character data to numbers, and restricts entry of numeric data to numbers and + and - signs |
| # | Restricts entry to numbers, spaces, periods, and signs |
| ! | Converts letters to uppercase |
| \$ | Inserts a dollar sign or the symbol defined with SET CURRENCY TO instead of leading blanks |
| * | Inserts asterisks in place of leading spaces |
| . | Marks the position of the decimal point |
| , | Separates thousands with a comma (or with another character indicated by SET SEPARATOR) |
| A | Restricts entry to alphabetic characters |
| L | Restricts entry to T, t, F, f, Y, y, N, or n, and converts it to uppercase |
| N | Restricts entry to letters and numbers |
| X | Allows any character |
| Y | Restricts entry to Y, y, N, or n, and restricts display to Y and N |

You may include *function* symbols in a template by preceding them with the @ symbol. If you combine template characters and function symbols in the same template, list function symbols first and separate them from the template characters with a space.

If the data is longer than the length of the *picture* string, it is truncated to match.

When displaying a calculated or morphed field, use a *picture* that represents the field's maximum size.

Example The following *picture* is for a phone number field that stores the digits only. By using the R *function*, preceded by the @ symbol in the *picture*, the literal characters in the template are not stored in the value, and are inserted when the value is displayed:

```
@R (999) 999-9999
```

Suppose you're using a morphed field that stores an ID number but displays a name. The name can be a maximum of 30 characters, so you set the *picture* property of Entryfield component that displays the name to 30 "X" characters:

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

See also *function*

picture is also a property of the PdxField class (page 14-104).

popupEnable

Whether an editor's popup menu is available.

Property of Editor

Description An Editor object has a popup menu that contains options to:

- Find and replace text
- Toggle its *wrap* and *evalTags* properties
- Show the Format toolbar

You may set the *popupEnable* property to *false* to prevent this menu from appearing when the user right-clicks the Editor object.

See also *evalTags*, *showFormatBar()*, *wrap*

popupMenu

Specifies a pop-up menu for a form.

Property of Form

Description After creating the Popup object as a child of the form, assign a reference to that Popup object to the form's *popupMenu* property to have the popup appear when the user right clicks on the form. In this way, you may have more than one popup menu defined for a form and change the popup menu that appears as needed.

Note You cannot name your popup *popupMenu*; that would conflict with the name of the existing property.

Example The following example is the *onOpen* event handler for the FISH.WFM form. It assigns the popup in FISH.POP as the popup menu for the form.

```
function Form_onOpen
  set procedure to FISH.POP additive
  this.popupMenu := new fishPopup(this,"POPUP")
```

After opening the FISH.POP file as a procedure file to load the popup class that it contains, the second statement creates the Popup object as a child object of the form with the name "POPUP", and assigns the resulting object to the form's *popupMenu* property.

See also class Popup (page ???), *useTablePopup*

prefixEnable

Determines whether the ampersand (&) character is interpreted as the accelerator key prefix.

Property of Text

Description When *prefixEnable* is *true*, the ampersand character is interpreted as the accelerator key prefix; the ampersand is not displayed, and the character that follows it is used as the accelerator key for the control that follows the Text object in the z-order. The accelerator key appears underlined.

Set *prefixEnable* to *false* to treat the ampersand as a normal character, so that it can be displayed within a Text control.

See also *shortCut*, *text*

prevSibling

The previous tree item with the same parent.

Property of TreeItem

Description The read-only *prevSibling* property contains an object reference to the previous tree item (up) that has the same parent. If the tree item is the first one, *prevSibling* is *null*.

See also *firstChild*, *nextSibling*, *noOfChildren*, *parent*

print()

Prints a form and the objects it contains.

Syntax <oRef>.print([<dialog expL>])

<oRef> An object reference to the form you want to print.

<dialog expL> An optional parameter that determines whether to display the standard print dialog. If omitted, the dialog is displayed by default. If the dialog is not displayed, the form is printed according to the settings of *_app.printer*.

Property of Form

Description Use the *print()* method to print a form on a selected printer. Executing the *print()* method opens the standard Print dialog box. If the user clicks OK, the current page of the form is printed on the selected printer.

You may set the *printable* property of individual controls to *false* to prevent them from printing.

See also *printable*

printable

Whether the component is printed when the form is printed.

Property of Most form components.

Description You may suppress the printing of individual components on the form by setting their *printable* property to *false*.

See also *print()*

rangeMax

Determines the upper limit for values in a component.

Property of Progress, ScrollBar, Slider, SpinBox

Description Use RangeMax in combination with RangeMin to specify a range restriction for values entered into a component. (*rangeMax* sets the upper limit and *rangeMin* sets the lower limit.) For example, an application that lets the user input a percentage would prevent the input of a value less than 0 or greater than 100. The same ranges would apply for a Progress component showing percent complete.

In a SpinBox component, if the value is too high, the value is set to *rangeMax*. SpinBox components allow both numbers and dates; the *rangeMax* must be the same data type. The Progress, Slider, and ScrollBar allow numbers only. *rangeMax* must be greater than *rangeMin*.

Note Range restrictions in a SpinBox have effect only when the *rangeRequired* property is *true*.

See also *rangeMin*, *rangeRequired*, *step*, *value*

rangeMin

Determines the lower limit for values in a component.

Property of Progress, ScrollBar, Slider, SpinBox

Description Use *rangeMin* in combination with *rangeMax* to specify a range restriction for values entered into a component. (*rangeMin* sets the lower limit and *rangeMax* sets the upper limit.) For example, an application that lets the user input a percentage would prevent the input of a value less than 0 or greater than 100. The same ranges would apply for a Progress component showing percent complete.

In a SpinBox component, if the value is too low, the value is set to *rangeMin*. SpinBox components allow both numbers and dates; the *rangeMin* must be the same data type. The Progress, Slider, and ScrollBar allow numbers only. *rangeMin* must be less than *rangeMax*.

Note Range restrictions in a SpinBox have effect only when the *rangeRequired* property is *true*.

See also *rangeMax*, *rangeRequired*, *step*, *value*

rangeRequired

Determines whether the range you specify with the *rangeMax* and *rangeMin* properties is enforced.

Property of SpinBox

Description Set *rangeRequired* to *true* to enforce a range limitation specified by the *rangeMax* and *rangeMin* properties. You may set *rangeRequired* to *false* to temporarily disable range checking.

When range checking is active, existing values are checked when they are displayed in the control. The spinbox also will not allow the entry of a number that is higher than *rangeMax* or lower than *rangeMin*. If the number—an existing number or new number—is out of range, the spinbox will change the number to the range limit; to *rangeMax* if the number is too large, or to *rangeMin* if the number is too small.

See also *rangeMax*, *rangeMin*, *spinOnly*, *step*, *valid*

readModal()

Opens a form as a modal window and returns an optional value.

Syntax `<oRef>.readModal()`
<oRef> The form to open.

Property of Form

`readModal()`

Description Use *readModal()* to open a form as a modal window. A modal window has the following characteristics:

- While the form is open, focus can't be transferred to other forms.
- Execution of the routine that opened the form stops until the form is closed. When the form is closed, control transfers to the statement after the one that opened the form.

The form's *close()* method takes an optional parameter. If the form is closed with a parameter, the value of that parameter is returned by *readModal()*. Otherwise *readModal()* returns *false*.

Many applications use modal forms as dialog boxes, which typically require users to take an action before the dialog box can be closed.

You can't open a form with the *readModal()* method when the MDI property is set to *true*.

To open a form as a *modeless* window, use the *open()* method.

Example The standard bootstrap code generated for a .WFM form file contains code to open the form with *readModal()* if the DO the .WFM with the parameter *true*, for example:

```
do ABOUT.WFM with true
```

By adding a couple of lines in the Header of the .WFM, you can make the form open modally by default. For example:

```
openForm( true )      // Call bootstrap with true
function openForm()    // Convert bootstrap into its own function
// Everything below is generated code
** END HEADER -- do not remove this line
//
// Generated on 09/28/97
//
parameter bModal
local f
f = new AboutForm()
if (bModal)
    f.mdi = false // ensure not MDI
    f.readModal()
else
    f.open()
endif
```

This is appropriate for forms that are always modal and do not have a return value, like About and Print dialogs.

See also *close()*, *mdi*, *open()*

reExecute()

Re-executes the report.

Syntax <oRef>.reExecute()

<oRef> The ReportViewer object that contains the report to re-execute.

Property of ReportViewer

Description Call *reExecute()* to execute the report again with a new set of parameters. To render another page in the existing report, call the report's *render()* method through the ReportViewer object's *ref* property.

Example See *params*

See also *params, ref*

ref

A reference to the Report object in a ReportViewer.

Property of ReportViewer

Description Use the ReportViewer object's *ref* property to access the report displayed.

Example Suppose you have a simple report preview form. To display the next page of the report, you increment the report's *startPage* and re-render() the report through the ReportViewer object's *ref* property.

```
function nextPage_onClick()
  with form.reportViewer1.ref
    endPage = ++startPage
    render()
  endwhile
```

See also *active, filename*
ref is also a property of the DataModRef class (page 14-108).

refresh()

Redraws the form or grid.

Syntax <oRel>.refresh()
 <oRel> The object to refresh

Property of Form, Grid

Description Use a form's *refresh()* method to update the controls on the form to reflect the current state of the data in the buffer when using tables in work areas. To update the data buffer, use the REFRESH command first. When using the data access objects, call the rowset's *refreshControls()* method instead.

Call a grid's *refresh()* method to repaint the grid. The current row in the rowset becomes the top row in the grid, and all visible columns are repainted from the current row down.

See also REFRESH, *refreshAlways*, *refreshControls()*
refresh() is also a method of the Rowset class (page 14-108)

refreshAlways

Whether to update the form after all form-based navigation and updates.

Property of Form

Description Set *refreshAlways* to *false* when performing extensive navigation or data processing during an event. When *refreshAlways* is *true*, Visual dBASE updates the form periodically during processing, which causes flicker and slows the process. When *refreshAlways* is *false*, the form is not updated until the event has completed.

You may force the update for the form during the event by calling *refresh()*.

See also *refresh()*

releaseAllChildren()

Deletes all tree items in the tree.

Syntax <oRef>.releaseAllChildren()
 <oRef> The TreeView object you want to clear.

Property of TreeView

Description Call *releaseAllChildren()* to delete the entire contents of a tree view so that you can start over. To delete an individual tree item and its subtree, call the tree item's *release()* method.

See also *count()*, *firstChild*, *release()*

right

The position of the right edge of an object relative to its container.

Property of Line

Description The unit of measurement in a form or report is controlled by its *metric* property. The default metric for forms is characters, and for reports it's twips.

See also *left*, *width*

rowSelect

Whether the entire row is visually selected.

Property of Grid

Description Set *rowSelect* to *true* to create the visual effect of highlighting the entire row in the grid.

See also *multiSelect*, *selected()*

rowset

The form's primary rowset.

Property of Form

Description For forms that use data objects, the *rowset* property identifies the form's primary rowset.

If the form uses only one query, then the Form designer assigns that query's rowset as the primary rowset. If the form uses a data module, the Form designer assigns the data module's primary rowset as the form's primary rowset.

The primary rowset is where navigation and other actions from the default menu and toolbar take place. Navigation in the primary rowset causes the form's *canNavigate* and *onNavigate* events to fire.

Example A button on a form that goes to the first row in the primary rowset would have an *onClick* event handler like this:

```
function firstButton_onClick()
    form.rowset.first()
```

`saveRecord()`

See also *canNavigate*, class Rowset (page 14-20), *onNavigate*, *view rowset* is also a property of the DataModule, Query, and StoredProc classes (page 14-114).

saveRecord()

Saves changes to the current record in the currently active table.

Syntax `<oRef>.saveRecord()`
<oRef> An object reference to the form.

Property of Form

Description Use *saveRecord()* for form-based data handling with tables in work areas. When using the data access objects, *saveRecord()* has no effect; use the rowset's *save()* method instead.

Form-based data buffering lets you manage the editing of existing records and the appending of new records. Editing changes to the current record are not written to the table until there is navigation off the record, or until *saveRecord()* is called. Each work area has its own separate edit buffer. For example, if you have two controls *dataLinked* to two different work areas, and you change both controls, you must call *saveRecord()* while each work area is selected to commit the changes.

To append a new record, call *beginAppend()*. This empties the record buffer for the currently selected work area. Calling *saveRecord()* writes the new record to the table, leaving you at the newly added record. Calling *beginAppend()* instead of *saveRecord()* will write the new record and empty the buffer again so you can add another record.

When appending records with *beginAppend()* the new record will not be saved when you call *saveRecord()* unless there have been changes to the record; the blank new record is abandoned. This prevents the saving of blank records in the table. (If you want to create blank records, use APPEND BLANK). You can check there have been changes by calling *isRecordChanged()*. If *isRecordChanged()* returns *true*, you should validate the record with form-level or row-level validation before writing it to the table.

To abandon the changed or new record instead of saving it, call *abandonRecord()*.

See also *abandonRecord()*, *isRecordChanged()*, *saveRecord()*

scaleFontBold

Whether the base font used for the Chars *metric* of a form is boldface.

Property of Form

Description The Chars *metric* of a form is based on the average height and width for characters in a specific base font. The base font is set through the form's *scaleFontBold*, *scaleFontName*, and *scaleFontSize* properties.

scaleFontBold determines whether the base font is boldface. Boldface fonts are wider than non-boldface fonts.

See also *metric*, *scaleFontName*, *scaleFontSize*

scaleFontName

The base font typeface used for the Chars *metric* of a form.

Property of Form

Description The Chars *metric* of a form is based on the average height and width for characters in a specific base font. The base font is set through the form's *scaleFontBold*, *scaleFontName*, and *scaleFontSize* properties.

scaleFontName is the name of the base font typeface.

See also *metric*, *scaleFontBold*, *scaleFontSize*

scaleFontSize

The point size of the base font used for the Chars *metric* of a form.

Property of Form

Description The Chars *metric* of a form is based on the average height and width for characters in a specific base font. The base font is set through the form's *scaleFontBold*, *scaleFontName*, and *scaleFontSize* properties.

scaleFontSize is the size of the base font, in points.

See also *metric*, *scaleFontBold*, *scaleFontName*

scrollBar

Determines when an object has a scroll bar.

Property of Browse, Editor, Form

`select()`

Description The *scrollBar* property determines when and if a control displays a scrollbar. It may have any of four settings:

| Value | Description |
|--------------|---|
| 0 (Off) | The object never has scroll bars. |
| 1 (On) | The object always has scroll bars. |
| 2 (Auto) | Displays scroll bars only when needed. |
| 3 (Disabled) | The scroll bars are visible but not usable. |

See also *autoSize*

select()

Makes the tree item the selected item in the tree.

Syntax `<oRef>.select()`

<oRef> An object reference to the tree item you want to select.

Property of TreeItem

Description Use *select()* to programmatically select a tree item. The tree view is scrolled and expanded if necessary to display the selected tree item.

See also *selected, showSelAlways*

selectAll

Determines if the value contained in a control initially appears selected (highlighted) when tabbing to the control.

Property of Entryfield, SpinBox

Description Set *selectAll* to *true* to give the user a shortcut for deleting or replacing the initial value in an entry field or a spin box. The entire value is highlighted when the user tabs to the control. The first character the user enters overwrites the value. Pressing *Del* or *Backspace* deletes the value. Pressing a direction key (such as *Home* or *End*) removes the highlight without erasing the value.

Clicking a control ignores *selectAll*; the cursor goes to the position clicked, and nothing is highlighted.

See also *tabStop*

selected

The currently selected tree item.

Property of TreeView

Description The *selected* property contains an object reference to the currently-selected tree item in the tree view, the one that has focus. If no item is selected, *selected* contains *null*.

selected is usually used in TreeView event handlers, which fire for all items in the tree.

select is read-only. To programmatically select an item, call the item's *select()* method.

See also *canChange*, *select()*, *showSelAlways*

selected()

Returns the currently selected item(s) in an object, or checks if a specified item is selected.

Syntax `<oRef>.selected([<item expN>])`

<oRef> An reference to the object you want to check.

<item expN> The item number to check. If omitted, the currently selected item(s) are returned. This parameter is allowed only for the ListBox control; the Grid method takes no parameters.

Property of Grid, ListBox

Description Calling *selected()* with no parameters returns the control's currently selected item or items. If a listbox's *multiple* property is *false*, *selected()* returns the currently selected item in the listbox. If *multiple* is *true*, *selected()* returns an array containing the currently selected items, one element per selection.

For a grid, *selected()* returns bookmarks to the row or rows (in an array) that are selected when *rowSelect* or *multiSelect* are *true*.

As with any Array object, check its *size* property to determine how many items have been selected. The items in the *selected()* array are listed in the order they were selected.

If you specify *<item expN>*, *selected()* will return the prompt string for that numbered item if the item is selected, or an empty string if the item is not selected.

The ListBox object's *value* property contains the value of the item that currently has focus, whether it's selected or not.

Example The following *onClick* event handler copies all the selected items in the ListBox object *select1* into another ListBox object named *select2*:

```
function makeSelections_onClick()
    form.select2.dataSource := "array form.select1.selected()"
```

See also *count()*, *multiple*, *multiSelect*, *rowSelect*, *value*

selectedImage

Image displayed between checkbox and text label when a tree item has focus.

Property of TreeItem, TreeView

Description The tree view may display images to the left of the text label of each tree item. If the tree has checkboxes, the image is displayed between the checkbox and the text label.

The *selectedImage* property of the TreeView object specifies the default icon image for all tree items to display when that tree item has focus. You may designate specific icons for each TreeItem object to override the default. Use the *image* property to specify icons for when the tree item does not have focus. If any individual item in the tree has its *image* or *selectedImage* property set, space is left in all tree items for an icon, even if they don't have one.

The *selectedImage* property is a string that can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies an icon resource and the DLL file that holds it.
- FILENAME <filename>
specifies an ICO icon file.

See also *image*, *imageScaleToFont*, *imageSize*, *selected*

serverName

Identifies the server application that is invoked when the user double-clicks an OLE viewer object.

Property of OLE

Description *serverName* is a read-only property that contains the name of the OLE server for the current contents of the OLE control. You may use *serverName* to anticipate which server application is activated if the user double-clicks the current OLE viewer object.

See also *linkFileName*, *oleType*

setAsFirstVisible()

Makes the tree item visible as the first (topmost) in the tree view.

Syntax <oRef>.setAsFirstVisible()

<oRef> An object reference to the tree item you want to display.

Property of TreeItem

Description Use *setAsFirstVisible()* when you want to make sure that a tree item is visible in the tree view as close to the top of the tree view area as possible. The tree is expanded and scrolled if necessary to make the item visible.

If the tree item is close to the bottom, the tree is scrolled as high as possible.

See also *ensureVisible()*, *select()*

setFocus()

Sets focus to a component.

Syntax <oRef>.setFocus()

<oRef> A reference to the object to receive focus.

Property of Form and all form components that get focus

Description Calling a component's *setFocus()* method moves the focus to that component (unless it already has focus). When you call a form's *setFocus()* method, the component on the form that last had focus gets the focus.

See also *onGotFocus*

setTic()

Manually sets a tic mark in a Slider object.

Syntax <oRef>.setTic(<expN>)

<oRef> The Slider object whose tic mark to set.

<expN> The location of the tic mark.

Property of Slider

setTicFrequency()

Description To manually set tic marks in a slider, set the slider's *tics* property to Manual. Call *clearTics()* to clear any previously set tic marks. Then call *setTic()* with the location of each tic mark.

The *<expN>* should be between the *rangeMin* and *rangeMax* of the Slider control.

See also *clearTics()*, *rangeMax*, *rangeMin*, *tics*, *ticsPos*

setTicFrequency()

Sets the tic mark interval for automatic tic marks in a Slider object.

Syntax *<oRef>.setTicFrequency(<expN>)*

<oRef> The Slider object whose tic mark to set.

<expN> The frequency of the tic marks.

Property of Slider

Description For automatic tic marks, set the slider's *tics* property to Auto. The default interval is 1. Call *setTicFrequency()* to use another interval value.

See also *tics*, *ticsPos*

shapeStyle

Determines the shape of a Shape object.

Property of Shape

Description Use *shapeStyle* to specify a shape for a Shape object. The shapes you can specify are as follows:

| Value | Shape |
|-------|--------------------------------|
| 0 | Rectangle with rounded corners |
| 1 | Rectangle |
| 2 | Ellipse |
| 3 | Circle |
| 4 | Square with rounded corners |
| 5 | Square |

See also *colorNormal*, *penStyle*, *penWidth*

showFormatBar()

Displays or hides the Format toolbar.

Syntax <oRef>.showFormatBar(<expl>)

<oRef> A Form object.

<expl> *true* to show the toolbar, *false* to hide the toolbar.

Property of Form

Description You may show or hide the Format toolbar when the form has focus by calling *showFormatBar()*.

See also *evalTags*

showMemoEditor()

Displays or hides the memo editor control for an entryfield.

Syntax <oRef>.showMemoEditor(<expl>)

<oRef> The Entryfield that's *dataLinked* to the memo field.

<expl> *true* to show the editor, *false* to hide the editor.

Property of Entryfield

Description When an entryfield control is *dataLinked* to a memo field, double-clicking the control opens a memo editor in a form. You may show or hide this editor by calling *showMemoEditor()*.

See also *memoEditor*

showSelAlways

Whether to highlight the selected item in the tree even when the tree view does not have focus.

Property of TreeView

Description When *showSelAlways* is *true*, the tree view highlights the selected item even when the tree view does not have focus. This highlight is different than when the tree view does have focus, but still visually indicates the selected item.

When *showSelAlways* is *false*, no item is highlighted when the tree view does not have focus. Even though no item is highlighted, the tree view's *selected* property still points to the item that had focus last.

See also *selected*

showSpeedTip

Determines if tool tips defined for a control appear.

Property of Form

Description Set *showSpeedTip* to *false* to suppress the display of all tool tips defined in the controls' *speedTip* property.

See also *speedTip*

sizeable

Determines if the user can resize a form when it's not MDI.

Property of Form

Description Set *sizeable* to *false* to prevent the form from being resized. You must set *sizeable* before you open the form.

sizeable has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and is always *sizeable*.

See also *maximize*, *mdi*, *minimize*, *moveable*, *onSize*, *sysMenu*

smallTitle

Determines if the form has the small palette-style title bar when it's not MDI.

Property of Form

Description Set *sizeable* to *true* to make the form look like a palette window, with the smaller title bar and no minimize, maximize, or system menu icons. You must set *smallTitle* before you open the form.

smallTitle has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and has a normal-sized title bar.

See also *mdi*

sortChildren()

Sorts child tree items.

Syntax <oRef>.sortChildren()

<oRef> The tree object whose children to sort.

Property of TreeItem, TreeView

Description *sortChildren()* sorts the child tree items of a tree view or tree item according to the *text* labels of the tree items. The sort is not case-sensitive and goes one level deep only.

sorted

Determines whether the prompts in a listbox or a combobox are listed in sorted order or in natural order.

Property of ComboBox, ListBox

Description Set *sorted* to *true* when you want the prompts in a list box or a combo box to appear in sorted order (alphabetically, numerically, or chronologically). For example, a list of names is more accessible if it is sorted alphabetically.

The natural order of a list box or a combo box depends on the order in which the prompts are generated. For example, when you specify "FILE *.*" for the *dataSource* property of a list box, the prompts consist of the file names in the default directory. The prompts are created in the order in which the files are listed in the directory, so they are not necessarily arranged alphabetically when you set *sorted* to *false*.

sorted has no effect when the *dataSource* property of the list box or combo box specifies "FIELD" followed by a field name. In this case, the order of prompts in the list box or combo box depends on the record sequence in the table containing the specified field.

See also *dataSource*

speedBar

Determines whether a pushbutton behaves like a toolbar button or a standard pushbutton.

Property of PushButton

Description Set *speedBar* to *true* when you want a pushbutton to behave like a toolbar button. A toolbar button is not included in the tab order of a form, so you can't get to it by pressing *Tab* or *Shift+Tab*; and when clicked, it does not receive focus. For example, navigation controls on a form usually have their *speedBar* property set to *true*. When you navigate from row to row, the control that has focus, typically one *dataLinked* to a field, never loses focus.

See also *tabStop*

speedTip

Specifies the tooltip text that appears when the mouse remains on a control for more than one second.

Property of CheckBox, ComboBox, Editor, Entryfield, Grid, NoteBook, Progress, PushButton, RadioButton, ScrollBar, Slider, SpinBox

Description Use *speedTip* to create a tool tip that appears when the mouse rests on a control. Usually this message gives the user a clue as to the function of the control. To suppress the display of these tool tips, set the *showSpeedTip* property of the form to *false*.

See also *showSpeedTip*, *statusMessage*

spinOnly

Determines if users can enter a value in the text box portion of a spin box.

Property of SpinBox

Description A spinbox lets users enter values in a text box or select values by clicking the arrows on the right edge of the spinbox. When you set the *spinOnly* property to *false*, the text box is enabled. When you set the *spinOnly* property to *true* the text box is disabled, restricting input to a predefined *step* value.

See also *step*

startSelection

The low end of the selection range in a Slider object.

Property of Slider

Description *startSelection* contains the low value in the selection range. It should be equal to or less than *endSelection*, and between the *rangeMin* and *rangeMax* values of the slider.

The selection is not displayed unless the slider's *enableSelection* property is *true*.

See also *enableSelection, rangeMax, rangeMin, endSelection*

statusMessage

The message to display on the status bar while an object has focus.

Property of Form and all form components that can receive focus

Description Use *statusMessage* to provide instructions to the user when the user selects an object. If you set the *statusMessage* of the form, that message is displayed in the status bar when a component's *statusMessage* is blank.

See also *speedTip*

step

Determines how the amount added or subtracted by clicking an arrow in a spinbox.

Property of SpinBox

Description Use *step* to control the rate at which a user can increase or decrease a numeric or date value. For example, a program that expresses large dollar values only in increments of \$500.00 would give a spinbox a *step* value of 500.

See also *rangeMax, rangeMin, spinOnly*

style

Specifies which parts of a combobox are usable and which parts are displayed automatically.

Property of ComboBox

Description Use *style* to determine how the user selects values in a combobox. The user selects a value from a combobox by entering initial characters in an entryfield or by selecting the value directly from the prompt list. The *style* property determines whether the text box is usable and whether the prompt list is displayed automatically.

You can give Style one of three values:

| Style value | Description |
|------------------|--|
| 0 (Simple) | The drop-down list is displayed automatically. |
| 1 (DropDown) | The user has to click the arrow to display the drop-down list. |
| 2 (DropDownList) | The user has to click the arrow to display the drop-down list, and the text box does not accept input; the user must choose from the list. |

Pressing *Alt+DownArrow* when the combobox has focus also displays the drop-down list.

See also *autoDrop, dropDownHeight, dropDownWidth*

sysMenu

Determines if a form has a control menu and close icon when it's not MDI.

Property of Form

Description Set *sysMenu* to *false* to hide the control menu and close icon on a form. You must set *sysMenu* before you open the form.

sysMenu has no effect unless the form's *mdi* property is *false*; if it is *true*, the form follows the MDI specification and always has a control menu and close icon.

See also *maximize, mdi, minimize, moveable, sizeable*

tabStop

Determines if the user can select an object by pressing *Tab* or *Shift+Tab*.

Property of All form components that can receive focus

Description Set the *tabStop* property to *false* when you want to remove an object from the tab order of the parent form. For example, a *TabBox* object to select pages on a form is often not put in the tab order because it's not a data entry control. (For data entry purposes, you could put a button at the end of the tab order to move the user to the next page.)

If you have a *PushButton* component that you don't want in the tab order, you may not want it to receive focus either. If that's the case, set its *speedBar* property to *true* instead, which prevents both tabbing and focus.

See also *before, speedBar*

text

The non-editable text that appears in a component.

Property of Browse, CheckBox, Form, Menu, PushButton, RadioButton, Rectangle, Text, TreeItem

Description The *text* of a CheckBox or RadioButton object is the descriptive text that appears beside the actual check box or radio button. The *text* of a TreeItem object is the item's editable text label. The *text* of a PushButton object is the text that appears on the button face, and the *text* of a Menu object is the menu prompt.

The *text* of a Rectangle object is the caption text that appears at the top of the control. The *text* of a Form object is the text that appears in the form's title bar. The *text* property has no effect in a Browse object; the property exists for compatibility only.

The *text* of a Text object is the content of the object: the actual HTML text that is displayed in the form or report.

You may assign any of the following types of data to the *text* property of a Text component:

- Boolean
- Numeric
- Integer
- Character
- Object
- Null
- DateTime
- Codeblock

If you assign a codeblock to the *text* property, it must return a value. Use either an expression codeblock or a statement codeblock that uses RETURN to return a value. The codeblock is evaluated whenever it is rendered.

Creating accelerator keys Use a *pick character* to create an *accelerator key* to let the user select a menu item or simulate clicking a CheckBox, RadioButton, or PushButton by pressing *Alt* and the pick character. To designate a character as a pick character, precede it with an ampersand (&) in the object's *text* property. The pick character is underlined when the object is displayed. A pick character may also be used for the *text* of a Text or Rectangle object; pressing the key combination gives focus to the first control that follows the Text or Rectangle object in the z-order that can receive focus.

Example If the following string is assigned to a Menu object's *text* property, the menu has the character A as its pick character; pressing *Alt+A* selects the menu item.

```
Select &All
```

See also *prefixEnable*, *textLeft*, *value*

textLeft

Whether the component's text displays to the left or the right of the graphic element.

Property of CheckBox, PushButton, RadioButton

Description The CheckBox, PushButton, and RadioButton objects combine a text label and a graphic element. (The PushButton uses the *upBitmap*, *downBitmap*, *focusBitmap*, and *disabledBitmap* properties to display a bitmap on the button face). Set *textLeft* to *true* if you want to text to display on the left side of the control. By default, *textLeft* is *false*, so the text displays on the right.

See also *disabledBitmap*, *downBitmap*, *focusBitmap*, *text*, *upBitmap*

tics

How to display the tic marks in a Slider object.

Property of Slider

Description *tics* is an enumerated property that can be one of the following values:

| Value | Description |
|-------|-------------|
| 0 | Auto |
| 1 | Manual |
| 2 | None |

If *tics* is Auto, set the tic mark interval with *setTicFrequency()*. For Manual tics, use the *setTic()* method. Use the *ticsPos* property to set where the tic marks are displayed.

See also *setTic()*, *setTicFrequency()*, *ticsPos*

ticsPos

Where to display the tic marks in a Slider object.

Property of Slider

Description *ticsPos* is an enumerated property that can be one of the following values:

| Value | Description |
|-------|--------------|
| 0 | Both |
| 1 | Bottom Right |
| 2 | Top Left |

Tic marks are displayed if the *tics* property is not set to None. If the slider is vertical, the tic marks are displayed on the right or left side of the slider, or both. If the slider is horizontal, the tic marks are displayed on the top or bottom, or both.

Make sure the Slider object is large enough to display the tic marks.

See also *tics*, *vertical*

toggle

Determines if a button acts as a two-state toggle.

Property of PushButton

Description Set *toggle* to *true* to have a PushButton object behave like a two-state toggle button that stays down when clicked. Clicking the button again makes it pop back up.

To check the state of a toggle button, check its *value* property.

See also *downBitmap*, *upBitmap*, *value*

toolTips

Whether to display text labels as tooltips if they are too long to display fully in the tree view area as the mouse passes over them.

Property of TreeView

Description When *toolTips* is *true*, tree item text labels are displayed as tooltips if necessary as the mouse passes over them. The tooltip displays directly over the obscured text label, in the exact position where the text label should appear.

In general, this allows the user to see the entire text label without having to scroll the tree view back and forth horizontally. However, some portion of the tree item must be visible; if the tree item is completely outside the tree view area, the item will not appear simply by pointing the mouse where the item would be (because the mouse is outside the bounds of the TreeView object).

See also *trackSelect*

top

The position of the top edge of an object relative to its container.

Property of All form objects.

Description The unit of measurement in a form or report is controlled by its *metric* property. The default metric for forms is characters, and for reports it's twips.

The container for an MDI form is the main *Visual* dBASE application window, also known as the MDI frame window, below the menu and any toolbars docked on the top of the window. For a non-MDI form, the container is the screen.

See also *height, left, mdi, move()*, *width*

topMost

Specifies whether a form displays on top of all other forms when it's not MDI.

Property of Form

Description Set a form's *topMost* property to *true* to make the form stay in the foreground while focus transfers to other windows. If more than one open form has *topMost* set to *true*, those windows behave normally in relation to each other, while always staying on top of all other windows.

topMost has an effect only when the *mdi* property is *false*.

See also *mdi, windowState*

trackSelect

Whether to highlight and underline tree items as the mouse passes over them.

Property of TreeView

Description Set *trackSelect* to *true* to give the user an extra visual indication of which tree item the mouse is currently over.

See also *toolTips*

transparent

Specifies whether an object's background matches the background color or image of its container.

Property of CheckBox, Container, PaintBox, RadioButton, Text

Description By setting an object's *transparent* property to *true*, its background takes on the the background color or image of its container, making the background appear to be transparent. Note that the background is not actually transparent; if the coontrol overlaps another control, you will still see the background of the container, not the portion of the control that has been overlapped.

See also *background, borderStyle, colorNormal*

uncheckedImage

The image to display when a tree item is not checked instead of an empty check box.

Property of TreeView

Description Use *uncheckedImage* to display a specific icon instead of the standard empty checkbox for the tree items in the tree that are not checked. *checkedImage* optionally specifies the icon to display for tree items that are checked. The tree must have its *checked* property set to *true* to enable checking; each tree item has a *checked* property that reflects whether the item is checked.

The *uncheckedImage* property is a string that can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies an icon resource and the DLL file that holds it.
- FILENAME <filename>
specifies an ICO icon file.

See also *checkBoxes, checked, checkedImage, imageScaleToFont, imageSize*

undo()

Reverses the effects of the last Cut or Paste action.

Syntax <oRef>.undo()

<oRef> An object reference to the control you want to restore.

Property of Browse, ComboBox, Editor, Entryfield, SpinBox

Description Use *undo()* when the user wants reverse the effects of the last Copy, Cut or Paste action.

If you have assigned a menubar to the form, you can use a menu item assigned to the menubar's *editUndoMenu* property instead of using the *undo()* method of individual objects on the form.

See also *copy()*, *cut()*, *editUndoMenu* (page 16-19), *paste()*

upBitmap

Specifies the graphic image to display in a pushbutton when it isn't selected.

Property of PushButton

Description Use *upBitmap* to give visual confirmation that a pushbutton is enabled and the user is not clicking it.

The *upBitmap* setting can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

When you specify a character string for the pushbutton with *text* and an image with *upBitmap*, the image is displayed with the character string.

See also class Image, *disabledBitmap*, *downBitmap*, *enabled*, *focusBitmap*, *textLeft*

useTablePopup

Specifies whether to use the default table navigation popup when no popup has been assigned to a form.

Property of Form

Description Set *useTablePopup* to *true* to have the default table navigation popup displayed when the user right-clicks the form and no popup has been assigned to the form's *popupMenu* property.

If a popup is assigned to *popupMenu*, that popup is always used.

See also *popupMenu*

valid

Event fired when attempting to leave a control; return value determines if focus can be removed.

Parameters none

Property of Editor, Entryfield, SpinBox

Description Use *valid* to validate data. *valid* fires when attempting to leave the control only when data has been changed, unless *validRequired* is *true*; then *valid* always fires.

The *valid* event handler must return *true* or *false*. If it returns *true*, operation continues normally. If it returns *false*, the *validErrorMsg* is displayed in a dialog box and the focus remains on the control.

valid does not fire if the user never visits the control, even if *validRequired* is *true*. Therefore, unless the control is the first or only control on the form that gets focus, you should always use form-level or row-level validation in addition to control-level or field-level validation.

To enforce a simple numeric range in a SpinBox control, use *rangeMax* and *rangeMin* instead of *valid*.

To perform an action when a control loses focus, use *onLostFocus*.

Example The following event handler is assigned as the *valid* event handler for an order number entryfield in a form for entering new orders. The order numbers are preprinted on a paper form. The validation checks if the order number is valid, and if it's valid, whether the order has already been entered.

```
function ORDER_NUM_valid
  local lRet
  lRet = false
  if form.nextObj == form.cancelButton
    lRet := true  // Allow cancel
  elseif empty( this.value ) or ;
    transform( val( this.value ), "@L " + this.picture ) # this.value
    this.validErrorMsg := "Order number must be four digits"
  elseif keymatch( this.value, tagno( "ORDER_NUM" ) )
    this.validErrorMsg := "That order has already been entered"
  else
    lRet := true
  endif
  return lRet
```

The order number field is the first field in the form that gets focus, so the user must get by this validation to continue. If they click the Cancel button—on this form, it's appropriately named *cancelButton*—the validation is bypassed so that the user can cancel the new order. The format of the field value is checked to make sure it is not blank and the correct length. A *picture* is set in the control to enforce digits-only, but it can't check for length; the *valid* routine does. Finally, *KEYMATCH()* checks if that order number has already been entered. If it

passes these checks, the *valid* returns *true*. If either check fails, the *validErrorMsg* is set appropriately and the event handler returns *false* so that the message is displayed and the focus remains in the control.

See also RangeMax, RangeMin, RangeRequired, ValidErrorMsg, ValidRequired, When

validErrorMsg

Specifies the message to display when the *valid* event handler returns *false*.

Property of Entryfield, SpinBox

Description When validating data with *valid*, set the *validErrorMsg* property of the control to a more specific error message than the default, “Invalid input”. You may set a static message for each specific control, for example “Bad account number”; or you may dynamically set the *validErrorMsg* from within the *valid* event with specifics about the particular error, for example “Account number requires six digits” or “Account expired”.

See also *valid*, *validRequired*

validRequired

Determines if the *validevent* fires even if the data is not changed.

Property of Entryfield, SpinBox

Description Set *validRequired* to *true* to validate existing data as well as new data. For *validRequired* to take effect, you must assign a *valid* event handler.

You typically set *validRequired* to *true* when you change a validation condition and need to verify and update existing data. For example, a business might add a digit to its account numbers and change the *valid* event handler of an entry field to require the new digit. If the *validRequired* property is set to *true*, Visual dBASE also detects any existing account numbers that lack the digit and forces the user to make appropriate changes.

valid does not fire if the user never visits the control, even if *validRequired* is *true*. Therefore, unless the control is the first or only control on the form that gets focus, you should always use form-level or row-level validation in addition to control-level or field-level validation.

See also *rangeRequired*, *valid*, *validErrorMsg*

value

The component's current value.

Property of CheckBox, ComboBox, Editor, Entryfield, ListBox, Progress, PushButton, RadioButton, ScrollBar, Slider, SpinBox

Description A component's *value* property reflects its value, which is

- The value that is displayed in a Entryfield, Editor, SpinBox, or ComboBox component
- *true* if a CheckBox component is checked; *false* if it's not checked
- *true* if a RadioButton component is the one in its group that is selected; *false* if it's not selected
- The item that has focus in a ListBox component
- The interpolated number for the current position in a Slider, Progress, or ScrollBar object.
- *true* if a *toggle* PushButton is down; *false* if it's up

Both field and component objects have a *value* property. (Fields in a table open in a work area do not have any properties, but they have a value; the concept is the same.) When they are *dataLinked*, changes in one object's *value* property are echoed in the other. The form component's *value* property reflects the value displayed in the component at any given moment. If the component's value is changed, it is copied into the field.

The *value* property for all fields in a rowset are set when you first open a query and updated as you navigate from row to row. The *value* properties for components *dataLinked* to those fields are also updated at the same time, unless the rowset's *notifyControls* property is set to *false*. You can also force the components to be updated by calling the rowset's *refreshControls()* method, which is useful if you have set a field's *value* property through code.

When reading or writing values to *dataLinked* components, you can use the *value* property of either the visual component or the field object; there's no difference, although you should be consistent. You may choose to program the visual interface, if the underlying data is more likely to change; or you might choose to work with the data access objects, so you don't have to worry about the names of the form components and whether they're correctly *dataLinked*. In general, it's easier and more portable for data access object events to access the fields, so you're more likely to assign to the *value* properties of the fields.

See also *curSel*, *dataLink*, *rangeMax*, *rangeMin*, *toggle*

value is also a property of the Field (page 14-125) and Parameter (page 14-126) classes.

vertical

Determines whether a Slider or ScrollBar object is vertical or horizontal.

Property of ScrollBar, Slider

Description Slider and ScrollBar objects may be horizontal or vertical. If *vertical* is *true* it's vertical; if *vertical* is *false*, it's horizontal.

Changing the *vertical* property does not resize the object. For example, if you have a long horizontal scrollbar, setting *vertical* to *true* results in a short, fat vertical scrollbar.

See also *height, width*

view

Specifies the name of a .QBE query or a table on which a form is based.

Property of Form

Description Use *view* for form-based data handling with tables in work areas. When using the data access objects, do not use *view*.

view determines which tables are automatically opened whenever the form is opened. You may specify a single table, or a .QBE query file. If you specify a single table, Visual dBASE internally issues CLOSE DATABASES to close all tables open in work areas (in the current workset) before opening the specified table in work area 1, in its natural or primary key order.

A .QBE file is a program file that is supposed to open one or more tables in a specific index order, and contains the appropriate Xbase DML commands such as SET RELATION and SET SKIP to create a multi-table view. In .QBE files generated by earlier versions of Visual dBASE, the first command in the file is CLOSE DATABASES, so using a generated .QBE also closes all open tables.

The specified table is opened (or the .QBE is executed) immediately when the *view* property is assigned.

Instead of using the *view* property, you may open the necessary tables yourself. All tables containing fields that are *dataLinked* to controls on the form must be open when the form is instantiated; otherwise the *dataLink* properties will fail (because the specified fields cannot be found), causing an error.

If one form opens another form that is supposed to use the current record in the first form, you don't want to set the *view* property in the second form, because instantiating that second form would close the tables used by the first form. This is a common situation where a form would use the current view, and not have anything assigned to its *view* property.

If a form does not have its own *view*, you may assign a *designView* property to the form so that the necessary tables are opened when you edit the form in the Form designer. The *designView* property has no effect when you actually run the form.

Example The following are example values for *view*:

```
ZIPCODES.DBF      // A single DBF table in the current directory
:IBLOCAL:EMPLOYEE // A table in a database
ORDERS.QBE        // A .QBE file in the current directory
```

See also Alias, DataLink, DataSource, DesignView

visible

Specifies whether a component is visible.

Property of All form components.

Description Use the *visible* property to conditionally hide a component. You may also disable a component without making it invisible by setting its *enabled* property to *false*. This makes the component appear grayed-out. Depending on the application, it may be more appropriate to completely hide something when it's not needed, or to let the user see it but not be able to do anything with it.

See also *enabled*

visibleCount()

Returns the number of tree items visible in the tree view area.

Syntax `<oRef>.visibleCount()`
<oRef> The tree view to check.

Property of TreeView

Description *visibleCount()* returns the number of tree items that are visible in the tree view area. To count all the tree items, whether or not they are visible, use the *count()* method.

See also *count()*, *setAsFirstVisible()*

visualStyle

The style of the tabs in a NoteBook object.

Property of Notebook

Description *visualStyle* is an enumerated property that can be one of the following values:

| Value | Description |
|-------|---------------|
| 0 | Right Justify |
| 1 | Fixed Width |
| 2 | Ragged Right |

The Fixed Width style makes all tabs the same width. The Right Justify and Ragged Right Styles are the same when the notebook’s *multiple* property is *false*; the tabs are sized to the width of their text label.

When *multiple* is *true* and there are enough tabs to create multiple rows, Right Justify makes the tab edges line up on both the left and right sides, while Ragged Right lines up the tabs on the left edge only.

See also *dataSource, multiple*

vScrollBar

Determines when an object has a vertical scroll bar.

Property of Grid

Description The *vScrollBar* property determines when and if a control displays a vertical scrollbar. It may have any of four settings:

| Value | Description |
|--------------|--|
| 0 (Off) | The object never has a vertical scroll bar. |
| 1 (On) | The object always has a vertical scroll bar. |
| 2 (Auto) | Displays a vertical scroll bar only when needed. |
| 3 (Disabled) | The vertical scroll bar is visible but not usable. |

See also *hScrollBar*

when

Event fired when attempting to give focus to an object; return value determines if object gets focus.

Parameters *<form open expl>* *true* for when the *when* event handler is called when the form is opened; *false* from then on.

Property of All form components that can get focus

Description Use *when* to conditionally prevent an object from getting focus without disabling the object. If you set an object's *enabled* property to *false*, the object will appear grayed-out and disabled. This is a visual indication that the object cannot get focus. The object is removed from the tab sequence and clicking the object has no effect.

If an object is enabled, you may define a *when* event handler to determine if an object is available. The event handler must return *true* to allow the object to get focus. If it returns *false*, the object does not get focus. If the object was clicked, focus remains on the object that previously had focus. If *Tab* or *Shift+Tab* was pressed, the focus goes to the next control in the tab sequence.

Using *when* gives you the flexibility of displaying a message or taking some other action when the focus is not allowed. But in most cases, it's better to conditionally disable controls so that they are clearly not available. For example, if you have a checkbox to echo output to a file and an entryfield for the file name, you can disable the entryfield when the checkbox is unchecked in the checkbox's *onChange* event handler.

If the *when* event handler returns *true*, or there is no *when* event handler, *onGotFocus* fires after the object receives focus.

The *when* event for all controls is fired when the form first opens. Use the `<form open expL>` parameter if necessary to distinguish that event from all other normal focus attempts.

See also *enabled*, *onGotFocus*, *valid*

width

The width of an object.

Property of Most form objects

Description Use an object's *width* property in combination with its *height* property to adjust the size of an object.

width is an integer expressed in the current *metric* unit of the form that contains the object. The value includes any border, bevel or shadow effect assigned to the object.

The default *metric* unit is characters, which is the average width of characters in the form's base font (default *scaleFontName* is MS Sans Serif, default *scaleFontSize* is 8 points). Thus, if the parent form's *metric* unit is set to characters, and its *scaleFontName* or *scaleFontSize* properties are changed, the objects on the form are automatically scaled relative to the new form size.

Exception: The *width* property of Line objects is used to set the thickness of the line, and is always measured in pixels.

See also *height, left, move(), top*

windowState

The maximized/minimized state of the form window.

Property of Form

Description Use *windowState* to get or set the display state of a form. A window may be maximized, minimized, or in its non-maximized “normal” (restored) state. The appearance of the window also depends on whether it is an MDI window, as noted in the following table.

| Setting | Effect on an MDI form | Effect on a non-MDI form |
|---------------|---|---|
| 0 (Normal) | If minimized or maximized, the form is restored to its most recent non-maximized size and position within the MDI frame window. | If minimized or maximized, the form is restored to its most recent non-maximized size and position on the screen. |
| 1 (Minimized) | Reduces a normal or maximized form to an iconized title bar at the bottom of the MDI frame window. | Reduces a normal or maximized form to an iconized button on the Windows taskbar. |
| 2 (Maximized) | Enlarges the form to the extent of the MDI frame window. | Enlarges the form to the extent of the screen. |

The MDI frame window is the main *Visual* dBASE application window.

If you assign a value to *windowState* that changes its state, the form’s *onSize* event fires and the form is also given focus. To give focus to the window without changing its state, call the form’s *setFocus()* method.

Example

See also *mdi, moveable, onSize, sizeable*

wrap

Determines if an Editor or Text object wraps text automatically.

Property of Text, Editor

Description Use *wrap* to wrap long lines of text in the editor or in a text object. When *wrap* is *true*, text won’t exceed the width of the object and a horizontal scrollbar will not appear. The Text or Editor object will attempt to break each line at a space.

If *wrap* is *false*, long lines extend past the right edge of the Editor. If *scrollBar* is On or Auto, a horizontal scrollbar is used (where needed) to view long lines of

text. Even without scrollbars, you can use the cursor to move to the end of long lines.

See also *scrollBar*

Application shell

This section covers supporting application elements such as menus, popups, toolbars, standard dialogs, keyboard behavior, and the `_app` object.

`_app`

The global object representing the currently running instance of *Visual* dBASE.

Syntax The `_app` object is automatically created when you start *Visual* dBASE.

Properties The following tables list the properties and events of the `_app` object. (No methods are associated with the `_app` object.)

| Property | Default | Description |
|-----------------------------|--------------|---|
| <code>className</code> | APPLICATION | Identifies the object as an instance of the <i>Visual</i> dBASE application (Property discussed in Chapter 5, “Core language.”) |
| <code>databases</code> | Object array | An array containing references to all database objects used by the Navigator |
| <code>ddeServiceName</code> | VDB | The name used to identify each instance of <i>Visual</i> dBASE when used as a DDE service |
| <code>frameWin</code> | Object | The <i>Visual</i> dBASE MDI frame window |
| <code>insert</code> | true | Whether text typed at the cursor is inserted or overwrites existing text |
| <code>printer</code> | Object | Configuration properties for the default printer |
| <code>session</code> | Object | The default Session object |
| <code>speedBar</code> | true | Whether to display the default toolbar |
| <code>statusBar</code> | true | Whether to display the status bar at the bottom of the MDI frame window |

Description

| Event | Parameters | Description |
|-------------------|--------------|--|
| <i>onInitiate</i> | <topic expC> | When a client application requests a DDE link with <i>Visual</i> dBASE as the server, and no DDETopic object for the specified topic exists in memory. |

Use *_app* to control and get information about the currently running instance of *Visual* dBASE. The *insert* property controls the insert or overwrite behavior of typed text in all forms, the Source Editor, and the Command window. It is toggled by pressing the *Insert* key. You may show or hide the default toolbars and the status bar. To control other aspects of the main application window, use the *_app.frameWin* object.

The *databases* array contains references to all databases opened by the Navigator. The default database is the first element in that array. The *session* property points to the default session. Therefore *_app.databases[1].session* and *_app.session* point to the same object.

To use *Visual* dBASE as a DDE server, set the *ddeServiceName* to a unique identifier if there is more than one instance of *Visual* dBASE running or if you want your application to have a specific DDE service name other than the default "VDB", then assign an *onInitiate* event handler to handle the service request. For more information and an example of using *ddeServiceName*, see Chapter 19, "Extending Visual dBASE with DLLs, OLE and DDE".

The *_app* object is also used to store important global values and other objects used by your application. Dynamically creating properties of *_app* is better than creating public variables. Variables may be inadvertently released or conflict with other variable names. Objects referenced only in variables cannot communicate with each other using object-oriented techniques like objects attached to the same parent object—in this case *_app*—can.

Example To set *Insert* off for your application:

```
_app.insert = false
```

The following example attaches a form manager object (a custom class) to the *_app* object.

```
set procedure to MANAGER.CC additive
_app.manager = new FormManager()
```

See Also *_app.frameWin*

_app.frameWin

The *Visual* dBASE MDI frame window.

Syntax The *_app.frameWin* object is automatically created when you start *Visual* dBASE.

Properties The following table lists the properties of the `_app.frameWin` object. (No events or methods are associated with the `_app.frameWin` object.)

| Property | Default | Description |
|--------------------------|--------------|---|
| <code>className</code> | FRAMEWINDOW | Identifies the object as an instance of an MDI frame window (Property discussed in Chapter 5, “Core language.”) |
| <code>hWnd</code> | | The Windows handle to the frame window |
| <code>text</code> | Visual dBASE | The title displayed in the frame window |
| <code>visible</code> | true | Whether the frame window is visible |
| <code>windowState</code> | | The state of the frame window: 0=Normal, 1=Minimized, 2=Maximized |

Description `_app.frameWin` represents the main *Visual* dBASE application window. This window is the frame window that contains all MDI windows during development and in an MDI application. If your application uses SDI windows only, the MDI frame window is usually hidden with the `SHELL()` function.

If you assign a `MenuBar` to `_app.frameWin`, that menu becomes the default menu that is visible when no MDI windows are open, or the current MDI window has no menu of its own. If you are using `SHELL()` to hide the Navigator and Command window in an MDI application, you must call `SHELL()` after assigning the `_app.frameWin` menu.

Example The following statements are used at the beginning of an application to set the title in the MDI frame window to the name of the application, attach a default menu, and hide the Navigator and Command window.

```
_app.frameWin.text := "Data Dazzler 3000"
do EMPTY.MNU with _app.frameWin
set procedure to EMPTY.MNU additive
shell( false, true )
```

See also `SHELL()`

class Menu

A menu item in a menubar or popup menu.

Syntax [`<oRef>` =] new Menu(`<parent>`)

<oRef> A variable or property—typically of `<parent>`—in which to store a reference to the newly created Menu object.

<parent> The parent object—a `MenuBar`, `Popup`, or another Menu object—that contains the Menu object.

Properties The following tables list the properties, events, and methods of the Menu class.

| Property | Default | Description |
|------------------------|---------|--|
| <i>before</i> | | The next sibling menu object |
| <i>checked</i> | false | Whether to display a checkmark next to a menu item. |
| <i>checkedBitmap</i> | | An image to represent the mark to appear next to the menu item if <i>checked</i> is <i>true</i> . |
| <i>className</i> | MENU | Identifies the object as an instance of the Menu class. |
| <i>enabled</i> | true | Determines if the menu can be selected. |
| <i>helpFile</i> | | Help file name |
| <i>helpId</i> | | Help index topic or context number for context-sensitive help |
| <i>id</i> | -1 | Supplementary ID number for menu item |
| <i>name</i> | | The name of the menu item. |
| <i>parent</i> | | The menu item's immediate container |
| <i>separator</i> | false | Whether the menu object is a separator instead of a selectable menu item. |
| <i>shortCut</i> | | The key combination that fires the <i>onClick</i> event. |
| <i>statusMessage</i> | | The message to display on the status bar when the menu item has focus. |
| <i>text</i> | | The text of the menu item prompt. |
| <i>uncheckedBitmap</i> | | An image to represent the mark to appear next to the menu item if <i>checked</i> is <i>false</i> . |

| Event | Parameters | Description |
|----------------|------------|--|
| <i>onClick</i> | | After the menu item is chosen. |
| <i>onHelp</i> | | When <i>F1</i> is pressed—overrides context-sensitive help |

| Method | Parameters | Description |
|------------------|------------|---|
| <i>release()</i> | | Explicitly removes the menu object from memory. |

Description Menu objects represent the individual menu items, or prompts, in a menu system. They can be attached to MenuBar objects, Popup objects, or other Menu objects so that:

- When attached to a menubar, they are the top-level menu items, such as the standard File and Edit menus.
- Menu items attached to a top-level menu item form the drop-down menu, such as the standard Cut and Paste menu items in the top-level Edit menu.
- Menu items attached to a popup comprise the items in the popup.

- Any other menu items that have menu items attached to them become cascading menus.

Unless a menu item has other menu items attached to it (making it a cascading menu) selecting the menu item fires its *onClick* event. Actions are assigned to each menu item by creating an *onClick* event handler for the menu object.

The actions for the standard Undo, Cut, Copy, and Paste menu items and the Window menu are handled by assigning the menu items to the menubar's *editUndoMenu*, *editCutMenu*, *editCopyMenu*, *editPasteMenu*, and *windowMenu* properties respectively.

Menu objects are also used as separators in a drop-down or popup menu by setting their *separator* property to *true*. In this case, the menu item serves no other purpose and cannot be a cascading menu or have an *onClick* event handler.

Creating accelerators and pick characters There are two ways to let the user choose a menu item by using the keyboard (which may be used at the same time):

- Assign a key combination to the menu item's *shortCut* property. This is sometimes called an accelerator. For example, *Ctrl+C* is usually used for the Cut menu item. Pressing the accelerator chooses the menu item even if the menu item is not visible.
- Specify a pick character in the *text* prompt of the menu item by preceding it with an ampersand (&). Pick characters work only when the menu item is visible. For top-level items in a menubar, you must press *Alt* and the pick character to activate the menu. Once the menu system is activated, pressing *Alt* in combination with the pick character is optional.

Note Assigning *F1* as the *shortCut* key for a menu item disables the built-in context-sensitive help based on the *helpFile* and *helpId* properties. The *onClick* for the menu item will be called instead. Therefore, if you have a menu item for Help it should *not* have *F1* assigned as its *shortCut* key unless you want to handle help yourself.

Example This excerpt from a .MNU file shows the definition of the top-level Edit menu item and its Undo menu item, where *this* is the MenuBar object:

```
this.EDIT = new MENU(this)
with (this.EDIT)
    text = "&Edit"
endwith

this.EDIT.UNDO = new MENU(this.EDIT)
with (this.EDIT.UNDO)
    text = "&Undo"
    shortCut = "Ctrl+Z"
endwith
```

The names of the menu prompts match the names of the menu objects as a convenience; it is not required. Note the use of the ampersand to designate the pick characters and that the Undo menu item's parent is the Edit menu. At the

end of the MenuBar object constructor, the Undo menu item is assigned to the menubar's *editUndoMenu* property to enable the standard Undo behavior:

```
this.editUndoMenu = this.EDIT.UNDO
```

See Also class MenuBar, class Popup, *editUndoMenu*, *editCutMenu*, *editCopyMenu*, *editPasteMenu*, *windowMenu*

class MenuBar

A form's menu.

Syntax [*<oRef>* =] new MenuBar(*<parent>* [, *<name expC>*])

<oRef> A variable or property in which to store a reference to the newly created MenuBar object.

<parent> The form (or the *_app.frameWin* object) to which you're binding the MenuBar object.

<name expC> An optional name for the MenuBar object. The Menu Designer always uses the name "root". If not specified, the MenuBar class will auto-generate a name for the object.

Properties The following table lists the properties of the Menubar class.

| Property | Default | Description |
|----------------------|---------|---|
| <i>className</i> | MENUBAR | Identifies the object as an instance of the MenuBar class. |
| <i>editCopyMenu</i> | | A menu item that copies selected text from a control to the Windows clipboard. |
| <i>editCutMenu</i> | | A menu item that deletes selected text from a control and copies it to the Windows clipboard. |
| <i>editPasteMenu</i> | | A menu item that pastes text from the Windows clipboard to the edit control with focus. |
| <i>editUndoMenu</i> | | A menu item that restores the form to the state before the last edit operation was performed. |
| <i>id</i> | -1 | A supplementary ID number for the object |
| <i>name</i> | | The menubar object's name. |
| <i>parent</i> | | An object reference that points to the parent form. |
| <i>windowMenu</i> | | A top-level menu that lists open MDI windows. |

| Event | Parameters | Description |
|-------------------|------------|--------------------------|
| <i>onInitMenu</i> | | When the menu is opened. |

| Method | Parameters | Description |
|------------------------|------------|--|
| <code>release()</code> | | Explicitly removes the menubar object from memory. |

Description A MenuBar object represents the top-level menu for a form. It contains one or more Menu objects which comprise the individual top-level menu items. The top-level menu of a form is displayed at the top of the form if the form's *mdi* property is *false*, or in the MDI frame window if the form's *mdi* property is true when the form has focus.

You may also designate a menubar that appears in the MDI frame when no MDI forms have focus by assigning a menubar to the `_app.frameWin` object.

The MenuBar object automatically binds itself to the `<parent>` form. Unlike other controls, you usually do not assign the resulting `<oRef>` as a property of the form, since this is done automatically, using the `<name expC>` that is specified. The Menu Designer always uses the name "root", so a form's menu is referred to with the object reference:

```
form.root
```

Example The following is the code generated by the Menu Designer for a basic menu that contains an empty File menu, the standard Edit menu, and a Window menu which lists all the open MDI forms. Note that the standard bootstrap code at the top of the file takes a form object as a parameter, and then binds the menubar object to that form using the name "root".

```
** END HEADER -- do not remove this line
//
// Generated on 01/09/98
//
parameter formObj
new BASICMENU(formObj, "root")

class BASICMENU(formObj, name) of MENUBAR(formObj, name)

    this.FILE = new MENU(this)
    with (this.FILE)
        text = "&File"
    endwith

    this.EDIT = new MENU(this)
    with (this.EDIT)
        text = "&Edit"
    endwith

    this.EDIT.UNDO = new MENU(this.EDIT)
    with (this.EDIT.UNDO)
        text = "&Undo"
        shortCut = "Ctrl+Z"
    endwith
```

class Popup

```
this.EDIT.CUT = new MENU(this.EDIT)
with (this.EDIT.CUT)
    text = "Cu&t"
    shortCut = "Ctrl+X"
endwith

this.EDIT.COPY = new MENU(this.EDIT)
with (this.EDIT.COPY)
    text = "&Copy"
    shortCut = "Ctrl+C"
endwith

this.EDIT.PASTE = new MENU(this.EDIT)
with (this.EDIT.PASTE)
    text = "&Paste"
    shortCut = "Ctrl+V"
endwith

this.WINDOW = new MENU(this)
with (this.WINDOW)
    text = "&Window"
endwith

this.windowMenu = this.WINDOW
this.editCutMenu = this.EDIT.cut
this.editCopyMenu = this.EDIT.copy
this.editPasteMenu = this.EDIT.paste
this.editUndoMenu = this.EDIT.undo
endclass
```

See Also *_app.frameWin*, class Menu, class Popup, *menuFile*

class Popup

A popup menu assigned to a form.

Syntax [*<oRef>* =] new Popup(*<parent>* [, *<name expC>*])

<oRef> A variable or property in which to store a reference to the newly created Popup object.

<parent> The form to which you're binding the Popup object.

<name expC> An optional name for the Popup object. If not specified, the Popup class will auto-generate a name for the object.

Properties The following tables list the properties, events and methods of the Popup class.

| Property | Default | Description |
|-------------------|--------------|--|
| <i>alignment</i> | Align Center | Aligns the popup menu horizontally relative to the right-click location (0=Align Center, 1=Align Left, 2=Align Right). |
| <i>className</i> | POPUP | Identifies the object as an instance of the Popup class. |
| <i>id</i> | -1 | A supplementary ID number for the object |
| <i>left</i> | | Sets the position of the left border. |
| <i>name</i> | | The popup object's name. |
| <i>parent</i> | | An object reference that points to the parent form. |
| <i>top</i> | | Sets the position of the top border. |
| <i>trackRight</i> | true | Determines whether the popup menu responds to a right mouse click for selection of a menu item. |

| Event | Parameters | Description |
|-------------------|------------|---------------------------------|
| <i>onInitMenu</i> | | After the popup menu is opened. |

| Method | Parameters | Description |
|------------------|------------|--|
| <i>open()</i> | | Opens the popup menu. |
| <i>release()</i> | | Explicitly removes the popup object from memory. |

Description A Popup object represents a context or popup menu that is displayed when you right-click a form. You may also open the popup explicitly by calling its *open()* method.

A form may have any number of popup menu bound to it. Only one menu at time can be assigned to the form's *popupMenu* property; that is the menu that appears when right-clicking the form.

The popup contains Menu objects that represent the menu items in the popup.

Example

See Also class Menu, *popupEnable*, *popupMenu*

class ToolBar

A toolbar assigned to a form.

Properties The following tables list the properties, events and methods of the ToolBar class.

| Property | Default | Description |
|--------------------|---------|--|
| <i>className</i> | TOOLBAR | Identifies the object as an instance of the ToolBar class. |
| <i>flat</i> | true | Logical value which toggles the appearance of buttons on the toolbar between always raised (false) to only raised when the pointer is over a button (true). |
| <i>floating</i> | false | Logical value that lets you specify your toolbar as docked (false) or floating (true). |
| <i>form</i> | null | Returns the object reference of the form to which the toolbar is attached. |
| <i>hWnd</i> | 0 | Returns the toolbar's handle. |
| <i>imageHeight</i> | 0 | Adjusts the default height for all buttons on the toolbar. Since all buttons must have the same height, if ImageHeight is set to 0, all buttons will match the height of the tallest button. If <i>ImageHeight</i> is set to a non-zero positive number, images assigned to buttons are either padded (by adding to the button frame) or truncated (by removing pixels from the center of the image or by clipping the edge of the image). |
| <i>imageWidth</i> | 0 | Specifies the width, in pixels, for all buttons on the toolbar. |
| <i>left</i> | 0 | Specifies the distance from the left side of the screen to the edge of a floating toolbar. |
| <i>text</i> | | String that appears in the title bar of a floating toolbar. |
| <i>top</i> | 0 | Specifies the distance from the top of the screen to the top of a floating toolbar. |
| <i>visible</i> | true | Logical property that lets you hide or reveal the toolbar. |

| Event | Parameters | Description |
|-----------------|------------|--|
| <i>onUpdate</i> | | Fires repeatedly while application is idle to update the status of the toolbuttons |

| Method | Parameters | Description |
|-----------------|------------|---|
| <i>attach()</i> | <form> | Establishes link between the toolbar and a form |
| <i>detach()</i> | <form> | Breaks link between the toolbar and a form |

Description Use class ToolBar to add a toolbar to a form.

Example Here's an example of an object definition program, MYTOOLBR.PRG, which defines a basic two-button toolbar for use in any form or application.

```
parameter FormObj
```

```

    if pcount() < 1
        msgbox("DO mytoolbr.prg WITH <form reference>")
        return
    endif
    t = findinstance( "myTBar" )
    if empty( t )
        ? "Creating toolbar"
        t = new myTBar()
    endif
    try
        t.attach( FormObj )
    catch ( Exception e )
        // Ignore already attached error
        ? "Already attached"
    endtry
    class myTBar of toolbar
        this.imagewidth = 16
        this.flat = true
        this.floating = false
        this.b1 = new toolbutton(this)
        this.b1.bitmap = 'filename ..\artwork\button\dooropen.bmp'
        this.b1.onClick = {;msgbox("door is open")}
        this.b1.speedtip = 'button1'
        this.b2 = new toolbutton(this)
        this.b2.bitmap = 'filename ..\artwork\button\doorshut.bmp'
        this.b2.onClick = {;msgbox("door is shut")}
        this.b2.speedtip = 'button2'
    endclass

```

See Also class ToolButton

class ToolButton

Defines the buttons on a toolbar.

Properties The following tables list the properties and events of the ToolButton class. (No methods are associated with this class.)

| Property | Default | Description |
|---------------------|---------|--|
| <i>bitmap</i> | | Graphic file (any supported format) or resource reference that contains one or more images that are to appear on the button. |
| <i>bitmapOffset</i> | 0 | Specifies the distance, in pixels, from the left of the specified Bitmap to the point at which your button graphic begins. This property is only needed when you specify a Bitmap that contain a series of images arranged from left to right. Use with <i>BitmapWidth</i> to specify how many pixels to display from the multiple-image Bitmap. Default is 0 (first item in a multiple-image Bitmap). |

| Property | Default | Description |
|--------------------|------------|---|
| <i>bitmapWidth</i> | 0 | Specifies the number of pixels from the specified Bitmap that you want to display on your button. This property is only needed when you specify a Bitmap that contain a series of images arranged from left to right. Use with <i>BitmapOffset</i> , which specifies the starting point of the image you want to display. |
| <i>checked</i> | false | Returns true if the button has its <i>TwoState</i> property set to true. Otherwise returns false. |
| <i>className</i> | TOOLBUTTON | Identifies the object as an instance of the ToolButton class. |
| <i>enabled</i> | true | Logical value that specifies whether or not the button responds when clicked. When set to false, the operating system attempts to visually change the button with hatching or a low-contrast version of the bitmap to indicate that the button is not available. |
| <i>parent</i> | N/A | An object reference that points to the parent Toolbar. |
| <i>separator</i> | false | Logical value that lets you set a vertical line on the toolbar to visually group buttons. If you specify a separator button, only its Visible property has any meaning. |
| <i>speedTip</i> | | Specifies the text that appears when the mouse rests over a button for more than one second. |
| <i>twoState</i> | true | Logical value that determines whether the button displays differently when it has been depressed and consequently sets the <i>Checked</i> property to true. |
| <i>visible</i> | false | Logical value that lets you hide (<i>false</i>) or show (<i>true</i>) the button. |

| Event | Parameters | Description |
|----------------|------------|------------------------------|
| <i>onClick</i> | | After the button is clicked. |

Description Use class ToolButton to define the buttons on an existing toolbar.

Example See class Toolbar.

See Also class Toolbar

attach()

Establishes link between a toolbar and a form.

Syntax <toRef>.attach(<oRef>)

<toRef> An object reference to the toolbar.

<oRef> An object reference to the form.

Property of ToolBar

Description Along with *detach()*, this method lets you manage toolbars in your application by connecting and disconnecting the objects as needed.

Typically, however, a toolbar is attached when a form calls a program in which the toolbar is defined, as is done in the included CLIPBAR.PRG sample:

```
parameter FormObj, bLarge
private typeCheck
local t, bNew
bNew = false
if ( PCOUNT() == 0 )
    MSGBOX("To attach this toolbar to a form use: " + ;
        CHR(13) + CHR(13) + ;
        "DO " + PROGRAM() + " WITH <form reference>","Alert")
else
    typeCheck = FindInstance("ClipToolbar")
    if ( TYPE("typeCheck") == "O" )
        t = typeCheck
    else
        SET PROCEDURE TO (PROGRAM()) ADDITIVE
        t = new ClipToolbar( bLarge )
        bNew := true
    endif
    t.attach( FormObj )
endif
return ( bNew )
```

See Also *detach()*

checked

Determines if a checkmark appears beside a menu item.

Property of Menu

Description Use *checked* to indicate that a condition or a process is turned on or off.

The checkmark appears to the left of the menu command when you set the *checked* property to *true*; the checkmark is removed when you set the *checked* property to *false*.

You may specify a bitmap to display instead of the checkmark with the *checkedBitmap* property, and a bitmap to display when *checked* is *false* with the *uncheckedBitmap* property.

Example Suppose you want to show or hide all rows where the age of the person listed is under 18. You have a menu prompt "Show minors" with the following *onClick* event handler:

```
function showMinors_onClick()
  if this.checked
    form.rowset.filter := "AGE >= 18"
  else
    form.rowset.clearFilter()
  endif
  this.checked := not this.checked
```

If the option is already checked, you hide the rows by setting a filter. If the option is unchecked, you clear the filter so that all rows are shown. In either case, you flip the value of the *checked* property.

See Also *checkedBitmap*, *uncheckedBitmap*

checkedBitmap

A bitmap to display instead of a checkmark when a menu item is *checked*.

Property of Menu

Description Use *checkedBitmap* to display a bitmap instead of a checkmark when a menu item's *checked* property is *true*.

The *checkedBitmap* setting can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

Note The display area in the menu item is very small (13 pixels square with Small fonts). If the bitmap is too large, the top left corner is displayed. Also, the color of the bitmap when the menu item is highlighted changes, depending on the system menu highlight color. Therefore, you may want to limit yourself to simple monochrome bitmaps.

See Also *checked*, *uncheckedBitmap*

CLEAR TYPEAHEAD

Clears the typeahead buffer, where keystrokes are stored while *Visual* dBASE is busy.

Syntax CLEAR TYPEAHEAD

Description If you have not issued a SET TYPEAHEAD TO 0 command, the keyboard typeahead buffer stores keystrokes the user enters while dBASE is busy processing other data. When the processing is complete and keyboard input is enabled again, dBASE processes and deletes the values in the buffer in the order they were entered until the buffer is empty. Use CLEAR TYPEAHEAD to discard any keystrokes that may have been entered during processing, to ensure that the keyboard data currently being processed comes directly from the keyboard.

For example, if you want to be able to fill in multiple screens quickly, one after the other, you would not issue CLEAR TYPEAHEAD during processing. This would let you continue typing data while data from one screen was being saved and the next (blank) one being displayed. The data you entered during processing would be entered onto the new screen when it appeared. On the other hand, if you want to make sure that no data is entered until the next screen is displayed, issue CLEAR TYPEAHEAD after displaying the blank screen and before beginning data entry.

See Also KEYBOARD, SET TYPEAHEAD

DEFINE COLOR

Creates and names a customized color.

Syntax DEFINE COLOR <color name>
<red expN>, <green expN>, <blue expN>

<red expN>, <green expN>, <blue expN> Specifies the proportions of red, green, and blue (RGB) that make up the defined color. Each number determines the intensity of the color it represents, and can range from 0 (least intensity) to 255 (greatest intensity).

Description Use DEFINE COLOR to create a custom color. Once you have defined <color name>, you can use it instead of one of the standard colors such as R, W, BG, silver, lemonchiffon, and so on.

The color you create with DEFINE COLOR is based on three numbers, <red expN>, <green expN>, and <blue expN>. Adjusting these numbers alters the color you create. For example, increasing or decreasing <green expN> increases or decreases the amount of green contained in the customized color.

`detach()`

Use the `GETCOLOR()` function to open a dialog box in which you create a custom color or choose from a palette of available colors. After exiting `GETCOLOR()`, issue `DEFINE COLOR` with the values it returns to define the desired color.

You can't override any standard color definitions. For a full list of standard colors, see the color property.

Colors defined with `DEFINE COLOR` are active only during the current *Visual* dBASE session. If you restart *Visual* dBASE, you must redefine the colors. You may redefine a custom color as often as you wish. Changing the definition of a color does not automatically change the color of objects that have been set to that color; you must reassign the color.

Example The following example defines some standard departmental colors from RGB values stored in a table:

```
function setCorporateColors( cDeptName )
    local q, r
    q = new Query( "select * from COLORSCHEME" )
    r = q.rowset
    if r.applyLocate( "DEPTNAME = '" + cDeptName + "'" )
        private rgbValue // Use private var for &macro
        rgbValue = r.fields[ "BACKGROUND" ].value
        define color DEPT_BG &rgbValue
        rgbValue = r.fields[ "LOGO" ].value
        define color DEPT_LOGO &rgbValue
        rgbValue = r.fields[ "TEXT" ].value
        define color DEPT_TEXT &rgbValue
    endif
```

The function takes the department name as a parameter. If that department has a listing in the table, RGB values like "115,180,40" are read from the table and used for different standard colors, such as the background and text color. `¯o` substitution is used to convert the value strings into the literal numeric values, separated by commas, that are expected by the `DEFINE COLOR` command.

Presumably, default departmental colors have already been defined so that if no match is found, the color names can be used.

See Also `colorNormal`, `GETCOLOR()`

detach()

Breaks links between a toolbar and a form.

Syntax `<toRef>.detach(<oRef>)`

<toRef> An object reference to the toolbar.

<oRef> An object reference to the form.

Property of ToolBar

Description Along with *attach()*, this method lets you manage toolbars in your application by connecting and disconnecting the objects as needed.

Typically, however, a toolbar is detached as part of a form's cleanup routines, as is done in the included MUGS sample:

```
function close
  private sFolder
  sFolder = this.restoreSet.folder
  CLOSE FORMS
  SET DIRECTORY TO &sFolder.
  this.toolbars.appbar.detach( _app.framewin )
  with ( _app)
    framewin.text := this.restoreSet.frameText
    speedbar      := this.restoreSet.speedBar
    app           := null
  endwith
  shell( true, true )
return
```

See Also *attach()*

editCopyMenu

Specifies a menu item that copies selected text from a control to the Windows Clipboard.

Property of MenuBar

Description *editCopyMenu* contains a reference to a menu object users select when they want to copy text.

You can use the *editCopyMenu* property of a form's menubar to copy selected text to the Windows Clipboard from any edit control in the form, instead of using the control's *Copy()* property. In effect, *editCopyMenu* calls *Copy()* for the active control. This lets you provide a way to copy text with less programming than would otherwise be needed. The Copy menu item is automatically disabled when no text is selected, and enabled when text is selected.

For example, suppose you have a Browse object (b) and an Editor object (e) on a form (f). To implement text copying, you could specify actions that would call *b.Copy()* or *e.Copy()* whenever a user wanted to copy text. However, if you use a menubar, you can set the *editCopyMenu* property to the menu item the user will select to copy text. Then, when the user selects that menu item, the text is automatically copied to the Windows Clipboard from the currently active control. You don't need to use the control's *Copy()* property at all.

If you use the Menu designer to create a menubar, *editCopyMenu* is automatically set to an item named Copy on a pulldown menu named Edit when you add the Edit menu to the menubar:

```
this.EditCopyMenu = this.Edit.Copy
```

Example See WindowMenu.

See Also CLASS MENUBAR, Copy(), *editCutMenu*, *editPasteMenu*, *editUndoMenu*, WindowMenu

editCutMenu

Specifies a menu item that cuts selected text from a control and places it on the Windows Clipboard.

Property of MenuBar

Description *editCutMenu* contains a reference to a menu object users select when they want to cut text.

You can use the *editCutMenu* property of a form's menubar to cut (delete) selected text and place it on the Windows Clipboard from any edit control in the form, instead of using the control's Cut() property. In effect, *editCutMenu* calls Cut() for the active control. This lets you provide a way to copy text with less programming than would otherwise be needed. The Cut menu item is automatically disabled when no text is selected and enabled when text is selected.

For more information, see *editCopyMenu*.

Example See WindowMenu.

See Also CLASS MENUBAR, Cut(), *editCopyMenu*, *editPasteMenu*, *editUndoMenu*, WindowMenu

editPasteMenu

Specifies a menu item that copies text from the Windows clipboard to the currently active edit control.

Property of MenuBar

Description *editPasteMenu* contains a reference to a menu object users select when they want to paste text to the cursor position in the currently active edit control.

You can use the *editPasteMenu* property of a form's menubar to paste text from the Windows Clipboard into any edit control in the form, instead of using the

control's Paste() property. In effect, *editPasteMenu* calls Paste() for the active control. This lets you provide a way to paste text with less programming than would otherwise be needed. The Paste menu item is automatically disabled when the clipboard is empty, and enabled when text is copied or cut to the Clipboard.

For more information, see *editCopyMenu*.

Example See WindowMenu.

See Also CLASS MENUBAR, Paste(), *editCopyMenu*, *editCutMenu*, *editUndoMenu*, WindowMenu

editUndoMenu

Specifies a menu item that reverses the effects of the last Cut, Copy, or Paste action.

Property of MenuBar

Description *editUndoMenu* contains a reference to a menu object users select when they want to undo their last Cut, Copy, or Paste action.

You can use the *editUndoMenu* property of a form's menubar to undo a Cut or Paste action from any edit control in the form, instead of using the control's Undo() property. In effect, *editUndoMenu* calls Undo() for the active control. This lets you provide a way to undo with less programming than would otherwise be needed.

For more information, see *editCopyMenu*.

Example See WindowMenu.

See Also CLASS MENUBAR, Undo(), *editCopyMenu*, *editCutMenu*, *editPasteMenu*, WindowMenu

GETCOLOR()

Calls a dialog box in which you can define a custom color or select a color from the color palette. Returns a character string containing the red, green, and blue values for the color selected.

Syntax GETCOLOR([<title expC>])

<title expC> A character string to appear as the title of the dialog box.

Description Use GETCOLOR() to open a dialog box in which you can choose a color from a palette of predefined colors or create a customized color. In this dialog box, you

GETFONT()

choose and create colors in the same way you do if you use the Color Palette available when you choose Color in the Windows Control Panel.

GETCOLOR() returns a string in the format "red value, green value, blue value", with each color value ranging from 0 to 255; for example "115,180,40". If you cancel the color dialog, GETCOLOR() returns an empty string.

You can use the string returned by GETCOLOR() in a related command, DEFINE COLOR, to use a specific color in a program.

Example The following event handler displays the color dialog to choose a color in a color scheme:

```
function backgroundColorButton_onClick
    private rgbValue      // Use private for &macro
    rgbValue = getcolor()
    if rgbValue # ""
        form.colorscheme1.rowset.fields[ "BACKGROUND" ].value = rgbValue
        define color SAMPLE_BG &rgbValue
        form.sampleBackground.colorNormal = "SAMPLE_BG"
    endif
```

If a color is chosen, the RGB string is stored in a field in a color scheme table, but the row is not saved, in case the user changes their mind. A sample color is defined (or redefined), and a sample rectangle's color is set to the newly defined color.

Note that you can assign RGB values directly to a color property, but those RGB values are expected to be in hexadecimal and in BGR (blue-green-red) order, for example "0x28b473" instead of "115,180,40". It's easier to define a custom color than to do the conversion.

See Also DEFINE COLOR

GETFONT()

Calls a dialog box in which you select a character font. Returns a string containing the font name, point size, font style (if you choose a style other than Regular), and family.

Syntax GETFONT([<title expC>])
<title expC> A character string to appear as the title of the dialog box.

Description Use GETFONT() to place the values associated with a specified font into a character string, as shown in the following examples. If you want to add a font to the [Fonts] section of VDB.INI but don't know its exact name or family, use GETFONT(). Then add the information GETFONT() returns into VDB.INI.

Example

| | |
|---------------------|---------------------------------|
| mNormal = GETFONT() | && choose Arial, Regular, 10-pt |
| ? mNormal | && returns "Arial,10,Swiss" |

```
mBold = GETFONT()
? mBold
```

```
&& choose Helvetica bold, 12-pt
&& returns "Helvetica,12,B,Swiss"
```

INKEY()

Gets the first keystroke waiting in the keyboard typeahead buffer. Can also be used to wait for a keystroke and return its value.

Syntax INKEY([<seconds expN>] [, <mouse expC>])

<seconds expN> The number of seconds INKEY() waits for a keystroke. Fractional times are allowed. If <expN> is zero, INKEY() waits indefinitely for a keystroke. If <expN> is less than zero, the parameter is ignored.

<mouse expC> Determines whether INKEY() returns a value when you click the mouse. If <expC> is "M" or "m", INKEY() returns -100. If <expC> is not "M" or "m", INKEY() ignores a mouse click and waits for a keystroke.

Description The keyboard typeahead buffer stores keystrokes the user enters while *Visual* dBASE is busy. A very fast typist may also fill the keyboard typeahead buffer—*Visual* dBASE is busy trying to keep up. These keystrokes are normally handled automatically; for example, characters are typed into entryfields and menu choices are made. Use INKEY() to handle the keystrokes yourself.

INKEY() returns the decimal value associated with the first key or key combination held in the keyboard typeahead buffer and removes that keystroke from the buffer. If the typeahead buffer is empty, INKEY() returns the value of zero.

Table 16.1 INKEY() return values

| Key pressed | Return value | Shift+key return value | Ctrl+Key return value | Alt+key* return value |
|-------------|--------------|---------------------------|--------------------------|--------------------------|
| 0 | 48 | Depends on keyboard | -404 | -452 |
| 1 | 49 | Depends on keyboard | -404 | -451 |
| 2 | 50 | Depends on keyboard | -404 | -450 |
| 3 | 51 | Depends on keyboard | -404 | -449 |
| 4 | 52 | Depends on keyboard | -404 | -448 |
| 5 | 53 | Depends on keyboard | 0 | -447 |
| 6 | 54 | Depends on keyboard | -30 | -446 |
| 7 | 55 | Depends on keyboard | -404 | -445 |

Table 16.1 INKEY() return values (continued)

| Key pressed | Return value | Shift+key return value | Ctrl+Key return value | Alt+key* return value |
|--------------------|--------------|---------------------------|--------------------------|--------------------------|
| <i>8</i> | 56 | Depends on keyboard | -404 | -444 |
| <i>9</i> | 57 | Depends on keyboard | -404 | -443 |
| <i>a</i> | 97 | 65 | 1 | -435 |
| <i>b</i> | 98 | 66 | 2 | -434 |
| <i>c</i> | 99 | 67 | 3 | -433 |
| <i>d</i> | 100 | 68 | 4 | -432 |
| <i>e</i> | 101 | 69 | 5 | -431 |
| <i>f</i> | 102 | 70 | 6 | -430 |
| <i>g</i> | 103 | 71 | 7 | -429 |
| <i>h</i> | 104 | 72 | 8 | -428 |
| <i>i</i> | 105 | 73 | 9 | -427 |
| <i>j</i> | 106 | 74 | 10 | -426 |
| <i>k</i> | 107 | 75 | 11 | -425 |
| <i>l</i> | 108 | 76 | 12 | -424 |
| <i>m</i> | 109 | 77 | 13 | -423 |
| <i>n</i> | 110 | 78 | 14 | -422 |
| <i>o</i> | 111 | 79 | 15 | -421 |
| <i>p</i> | 112 | 80 | 16 | -420 |
| <i>q</i> | 113 | 81 | 17 | -419 |
| <i>r</i> | 114 | 82 | 18 | -418 |
| <i>s</i> | 115 | 83 | 19 | -417 |
| <i>t</i> | 116 | 84 | 20 | -416 |
| <i>u</i> | 117 | 85 | 21 | -415 |
| <i>v</i> | 118 | 86 | 22 | -414 |
| <i>w</i> | 119 | 87 | 23 | -413 |
| <i>x</i> | 120 | 88 | 24 | -412 |
| <i>y</i> | 121 | 89 | 25 | -411 |
| <i>z</i> | 122 | 90 | 26 | -410 |
| <i>F1 (Ctrl+`)</i> | 28 | -20 | -10 | -30 |
| <i>F2</i> | -1 | -21 | -11 | -31 |
| <i>F3</i> | -2 | -22 | -12 | -32 |
| <i>F4</i> | -3 | -23 | -13 | -33 |
| <i>F5</i> | -4 | -24 | -14 | -34 |
| <i>F6</i> | -5 | -25 | -15 | -35 |
| <i>F7</i> | -6 | -26 | -16 | -36 |
| <i>F8</i> | -7 | -27 | -17 | -37 |
| <i>F9</i> | -8 | -28 | -18 | -38 |
| <i>F10</i> | -9 | -29 | -19 | -39 |

Table 16.1 INKEY() return values (continued)

| Key pressed | Return value | Shift+key return value | Ctrl+Key return value | Alt+key* return value |
|---------------|--------------|---------------------------|--------------------------|--------------------------|
| F11 | -544 | -546 | -548 | -550 |
| F12 | -545 | -547 | -549 | -551 |
| Left Arrow | 19 | -500 | 1 | 0 |
| Right Arrow | 4 | -501 | 6 | 0 |
| Up Arrow | 5 | 5 | 5 | 0 |
| Down Arrow | 24 | 24 | 24 | 0 |
| Home (Ctrl+J) | 26 | 26 | 29 | 0 |
| End | 2 | 2 | 23 | 0 |
| Tab | 9 | -400 | 0 | 0 |
| Enter | 13 | 0 | -402 | 0 |
| Esc (Ctrl+I) | 27 | 27 | - | - |
| Ins | 22 | 0 | 0 | 0 |
| Del | 7 | -502 | 7 | 7 |
| Backspace | 127 | 127 | -401 | -403 |
| PgUp | 18 | 18 | 31 | 0 |
| PgDn | 3 | 3 | 30 | 0 |

***Note** The Alt+key value returned for all character keys, except lower-case letters a through z, is the character value minus 500. For lower-case letters, the Alt+key values are the same as those for upper-case letters.

Because of the event-driven nature of *Visual* dBASE, INKEY() is rarely used. When it is used, it's in one of three ways:

- When keystrokes are expected to be buffered, INKEY() is used to get those keystrokes.
- In a loop that's busy doing something, INKEY() is used to see if a key has been pressed, and if so to take an action.
- INKEY() can be used to wait for a keystroke, and then take an action.

In any of these cases, because *Visual* dBASE is busy executing your INKEY() code, it will not respond to keystrokes and mouse clicks as it normally would.

To check if there is a key waiting in the buffer without removing it, or to determine a value in the buffer in a position other than the first position, use NEXTKEY().

Example Execute the following program and follow the on-screen directions:

```
? "In the next 5 seconds, press [C] and [Alt+F]
sleep 5
? inkey()
? inkey()
```

While *Visual* dBASE is sleeping, it stores the values 67 and -430 in the typeahead buffer. INKEY() returns 67, and a second INKEY() returns -420.

The following example continuously executes a loop that shows the value of INKEY() and the character typed. The loop ends when the Escape key (ASCII 27) is pressed:

```
CLEAR
SET ESCAPE OFF
* ESCAPE ON will interrupt the program and the
* Escape key will not be trapped by INKEY()
? "Press Esc to continue"
k=0
DO WHILE k <> 27
    k=INKEY()
    ? k
    IF k>0
        ?? CHR(k)  && show the key and the ascii char
    ENDIF
ENDDO
SET ESCAPE ON
```

The following example displays a message and waits up to 10 seconds or until any key or mouse button is clicked:

```
? "This message will display for 10 seconds max"
Pause=inkey(10,"m")
IF Pause=0
    ? "You waited the 10 seconds"
ENDIF
```

See Also CLEAR TYPEAHEAD, KEYBOARD, NEXTKEY(), ON KEY, READKEY(), SET TYPEAHEAD

KEYBOARD

Inserts keystrokes into the typeahead buffer.

Syntax KEYBOARD <expC> [CLEAR]

<expC> A character string, which may include mnemonic strings representing key labels.

CLEAR Empties the typeahead buffer before inserting <expC>.

Description Use KEYBOARD to simulate keystrokes by placing or "stuffing" them in the typeahead buffer.

KEYBOARD can place any number of characters in the typeahead buffer, up to the limit specified by SET TYPEAHEAD; subsequent characters are ignored. If SET TYPEAHEAD is 0, you may KEYBOARD one character.

Keystrokes simulated with KEYBOARD are treated like normal keystrokes, going into the control that currently has focus. Some controls support a *keyboard*() method that enables you to send keystrokes to that specific control.

In addition to the simple alphanumeric keys on the keyboard, you may also use mnemonic strings to simulate must function keys. For a list of mnemonic strings, see the *keyboard()* method.

Example The following method is a Key event handler for a custom entryfield that executes a SEEK in the current workarea as keys are pressed—an incremental search control.

```
function Key( nChar, nPosition )
  if nextkey() # 0           // If keys pending
    return ( nChar # 255 ) // do nothing, and suppress keystroke if value is 255
  endif
  if nChar == 255           // If keystroke is special value 255
    if not isblank( this.value )
      this.seek()          // call control's seek() method to do actual SEEK
    endif
  else
    if nChar >= 32 and nChar < 255
      keyboard "{255}"      // For alphanumeric keystrokes, stuff special key 255
    endif
  endif
  return nChar
endfunction
```

Like all Key event handlers, this one receives two parameters: the value of the keystroke, and the current position in the control. This method does not use the position parameter. It does use a specific keystroke, which is chosen so that it does not conflict with typical keys. That keystroke has the INKEY() value 255.

The method first checks if there are keystrokes pending in the keyboard typeahead buffer—which would happen for a fast typist—and if so, no further action is taken, since those pending keystrokes would immediately cause the Key event handler to fire again, and any action at this point would be wasted.

For an alphanumeric keystroke—that is, one that is not a control or function key—the special key is stuffed into the keyboard buffer by the KEYBOARD command. This causes the Key event to be fired again, after the key that was actually typed goes into the entryfield as usual. The Key method detects the special key and performs the SEEK, which is coded in a separate seek() method for modularity; that is, the SEEK behavior may be modified without changing the Key method.

The KEYBOARD command is used instead of the *keyboard()* method because the *keyboard()* method inserts keystrokes directly into the control, causing the Key event to fire immediately from within the Key event handler, which in turn would cause the stuffed keystroke to be handled before the key that was actually typed. The KEYBOARD command uses the general typeahead buffer instead, and the keys would be handled in sequence.

See Also INKEY(), NEXTKEY()

MSGBOX()

Opens a dialog box that displays a message and pushbuttons, and returns a numeric value that corresponds to the pushbutton the user chooses.

Syntax MSGBOX(<message expC>, [<title expC>, [<box type expN>]])





<message expC> The message to display in the dialog box.

<title expC> The title to display in the title bar of the dialog box.

<box type expN> A numeric value that determines which icon (if any) and which pushbuttons to display in the dialog box. To specify a dialog box with pushbuttons and no icon, use the following numbers:

| <box type expN> | Pushbuttons |
|-----------------|----------------------|
| 0 | OK |
| 1 | OK, Cancel |
| 2 | Abort, Retry, Ignore |
| 3 | Yes, No, Cancel |
| 4 | Yes, No |
| 5 | Retry, Cancel |

To specify a dialog box with pushbuttons and an icon, add any of the following numbers to <box type expN>:

| Number to add | Icon displayed |
|---------------|---|
| 16 |  |
| 32 |  |
| 48 |  |
| 64 |  |

When a dialog box has more than one pushbutton, the left most pushbutton is normally the default. However, if you add 256 to <box type expN>, the second pushbutton is the default, and if you add 512 to <box type expN>, the third pushbutton is the default.

If you omit *<box type expN>*, box type 0—one with just the title, message, and an OK button—is used by default.

Note If you specify *<box type expN>*, make sure it's a valid combination of the choices outlined above. An invalid number may result in a dialog box that you cannot close.

Description Use MSGBOX() to prompt the user to make a choice or acknowledge a message by clicking a pushbutton in a modal dialog box.

While the dialog box is open, program execution stops and the user cannot give focus to another window. When the user chooses a pushbutton, the dialog box disappears, program execution resumes, and MSGBOX() returns a numeric value that indicates which pushbutton was chosen.

| Pushbutton | Return value |
|------------|--------------|
| OK | 1 |
| Cancel | 2 |
| Abort | 3 |
| Retry | 4 |
| Ignore | 5 |
| Yes | 6 |
| No | 7 |

Example The simplest dialog box displays a message only, for example:

```
calculate avg( SALARY ) to nAvgSalary
msgbox( "The average salary is $" + ltrim( str( nAvgSalary, 10, 2 ) ), "Results")
```

With only one OK button, you don't care what the return value from MSGBOX() is. You can put an icon in the dialog box to dress it up:

```
msgbox( "The average salary is $" + ltrim( str( nAvgSalary, 10, 2 ) ), "Results", 64 )
```

This example shows a dialog asking for confirmation before proceeding:

```
function revertButton_onClick
  if msgbox( "Undo changes to this record?", "Revert", 4) == 6
    form.rowset.abandon()
  endif
```

In addition to adding an icon to the confirmation dialog, in this case you probably want "No", the second pushbutton, to be the default. You can also make your code easier to write and more readable by creating manifest constants for the various return values:

```
#define BUTTON_OK      1
#define BUTTON_CANCEL  2
#define BUTTON_ABORT   3
#define BUTTON_RETRY   4
#define BUTTON_IGNORE  5
#define BUTTON_YES     6
#define BUTTON_NO      7
```

These manifest constants would be defined in a standard include file, and included in the Header of the form. You could then use them in any of the methods of the form. With these three enhancements, the IF statement would look like:

```
if msgbox( "Undo changes to this record?", "Revert", 4+48+256 ) == BUTTON_YES
```

While you could do the math in advance ($4 + 48 + 256 = 308$), *Visual* dBASE is perfectly capable of doing it at runtime with no noticeable delay. In fact, you could extend this idea by using the preprocessor to create standard MSGBOX() combinations, for example:

```
#define CONFIRM(m,t) (msgbox(m,t,4+32)==BUTTON_YES)
```

This combines the dialog box options and the test to see if the Yes button was clicked. Then in your programs, you would use the CONFIRM() macro like a function that returns a logical value:

```
if CONFIRM( "This record will be lost forever! You sure?", "Delete" )
```

See Also *readModal()*, WAIT

NEXTKEY()

Checks for and returns a keystroke held in the keyboard typeahead buffer.

Syntax NEXTKEY([<expN>])

<expN> The position of the key or key combination in the typeahead buffer. If <expN> is omitted, NEXTKEY() returns the value of the first keystroke in the buffer. If <expN> is larger than the number of keystrokes in the buffer, NEXTKEY() returns 0.

Description The keyboard typeahead buffer stores keystrokes the user enters while *Visual* dBASE is busy. A very fast typist may also fill the keyboard typeahead buffer—*Visual* dBASE is busy trying to keep up. These keystrokes are normally handled automatically; for example, characters are typed into entryfields and menu choices are made. Use NEXTKEY() to check if there are any buffered keystrokes, or to look for a keystroke in a specific position in the buffer.

NEXTKEY() returns the decimal value associated with the key or key combination held in the keyboard typeahead buffer at the specified position in the buffer. Unlike INKEY(), NEXTKEY() does not remove the keystroke from the buffer. If the buffer is empty or there is no keystroke at the specified position, NEXTKEY() returns a value of zero.

For a list of keystroke values, see INKEY().

Example The following method is a Key event handler for a custom entryfield that executes a SEEK in the current workarea as keys are pressed—an incremental search control. It checks if there are keystrokes pending in the keyboard typeahead buffer—which would happen for a fast typist—and if so, no SEEK is

executed, since those pending keystrokes would immediately cause another SEEK.

```
function Key( nChar, nPosition )
  if nextkey() # 0          // If keys pending
    return ( nChar # 255 ) // do nothing, and suppress keystroke if value is 255
  endif
  if nChar == 255           // If keystroke is special value 255
    if not isblank( this.value )
      this.seek()          // call control's seek() method to do actual SEEK
    endif
  else
    if nChar >= 32 and nChar < 255
      keyboard "{255}"      // For alphanumeric keystrokes, stuff special key 255
    endif
  endif
  return nChar
endfunction
```

Like all Key event handlers, this one receives two parameters: the value of the keystroke, and the current position in the control. This method does not use the position parameter. It does use a specific keystroke, which is chosen so that it does not conflict with typical keys. That keystroke has the INKEY() value 255.

For an alphanumeric keystroke—that is, one that is not a control or function key—the special key is stuffed into the keyboard buffer by the KEYBOARD command. This causes the Key event to be fired again, after the key that was actually typed goes into the entryfield as usual. The Key method detects the special key and performs the SEEK, which is coded in a separate seek() method for modularity; that is, the SEEK behavior may be modified without changing the Key method.

See Also INKEY()

ON ESCAPE

Changes the default behavior of the *Esc* key so that it executes a specified command instead of interrupting command or program execution.

Syntax ON ESCAPE [*<command>*]

<command> The command to execute when the following conditions are in effect:

- SET ESCAPE is ON
- The user presses *Esc* during command or program execution

The *<command>* may be any valid *Visual* dBASE command, including a DO command to execute a program file, a function call that executes a program or function loaded in memory, or a codeblock.

ON ESCAPE without a *<command>* option disables any previous ON ESCAPE *<command>*.

Description The primary purpose of the *Esc* key is to interrupt command or program execution. This behavior may be changed with ON ESCAPE; either way it occurs only when SET ESCAPE is ON (its default setting).

When no ON ESCAPE *<command>* is in effect, pressing *Esc* interrupts program execution and displays the *Visual* dBASE Program Interrupted dialog box. If ON ESCAPE *<command>* is in effect, pressing *Esc* during program execution executes the specified command instead and then continues program execution.

While executing a command (like CALCULATE) from the Command window, pressing *Esc* with ON ESCAPE *<command>* in effect executes *<command>* and then terminates the command, returning control to the Command window. If no ON ESCAPE *<command>* is in effect, pressing *Esc* during a command from the Command window simply terminates that command and displays a message in the status bar.

Note that user interface elements such as menus, forms, and dialog boxes handle *Esc* differently, usually closing or dismissing that UI element. (For forms, this behavior is controlled by its *escExit* property.) In those cases, ON ESCAPE and SET ESCAPE have no effect. In fact, with the exception of dialog boxes and forms opened with ReadModal(), because of the event-driven nature of *Visual* dBASE there is no program executing when you use a menu or type into a form, so there is nothing to interrupt.

Use ON KEY to specify a new meaning or mapping for *Esc* or any other key. If both ON KEY and ON ESCAPE are in effect, ON KEY takes precedence when *Esc* is pressed. In other words, while ON ESCAPE changes the Escape behavior, ON KEY changes the meaning of the *Esc* key, so that pressing it no longer causes that Escape behavior. While the Escape behavior affects only programs or commands that are executing, ON KEY works at all times.

If you issue ON ESCAPE*<command>* in a program, you should disable the current ON ESCAPE condition by issuing ON ESCAPE without a *<command>* option before the program ends. Otherwise, the ON ESCAPE condition remains in effect for any subsequent commands and programs you issue and run until you exit *Visual* dBASE.

Example The following example uses ON ESCAPE to substitute a small program, PrgEscape, for the Escape key. In this example, the programmer has wrongly programmed an endless loop that requires the use of Escape to debug. When escape is pressed, the program and line will be shown:

```
SET PROCEDURE TO PROGRAM(1) ADDITIVE
* So that other program files can access PrgEscape
SET ESCAPE ON
ON ESCAPE DO PrgEscape WITH PROGRAM(1),LINENO()
DO WHILE .t.  && set up an endless loop
    x="always allow a way out"
    y=" of a do while loop"
    z="Press Escape to stop"
    ? x
    ? y
    ? z
```

```

ENDDO

Procedure PrgEscape
PARAMETERS prg,line
? "Programmed Escape:",prg,line
SUSPEND

```

See Also *escExit*, ON ERROR, ON KEY, SET ESCAPE

ON KEY

Changes the keyboard mapping to execute a command when a specified key or key combination is pressed.

Syntax ON KEY [LABEL <key label>] [<command>]

LABEL <key label> Identifies the key or key combination that, when pressed, causes <command> to execute. Without LABEL <key label>, Visual dBASE executes <command> when you press any key. ON KEY LABEL is not case-sensitive.

<command> The command that is executed when you press the key or key combination. If you omit <command>, the command previously assigned by ON KEY is disabled.

Description Each key on the keyboard has a default meaning or mapping. For alphanumeric keys, that mapping simply types that character. Function keys have predefined actions. The *Esc* key terminates program execution. Use ON KEY to specify a command that executes when the user presses a key or key combination, overriding the default mapping.

Actions defined by ON KEY will interrupt programs, but not commands; in a program, the ON KEY action will occur after the current command has completed and then the program will continue. ON KEY also doesn't affect the execution of some commands or functions that are specifically looking for keystrokes, such WAIT or INKEY(). On the other hand, if you KEYBOARD a key that has been remapped, the remapped behavior will occur.

When you issue both ON KEY LABEL <key label> <command> and ON KEY <command>, the key or key combination you specify with ON KEY LABEL <key label> <command> takes precedence and executes its associated <command>. This way you can define actions for specific keys, and a default global action for all other keys. There may be only one ON KEY specification for each specific key and one global action defined at a given time.

ON KEY without arguments removes the effect of all previously-entered ON KEY <command> commands, with or without a LABEL.

To change the default Escape behavior, which interrupts the currently executing program or command, use ON ESCAPE. If both ON KEY and ON ESCAPE are in effect, ON KEY takes precedence when *Esc* is pressed. In other

words, while ON ESCAPE changes the Escape behavior, ON KEY changes the meaning of the *Esc* key, so that pressing it no longer causes that Escape behavior. While the Escape behavior affects only programs or commands that are executing, ON KEY works at all times.

To assign strings to function keys, use SET FUNCTION. If both ON KEY on SET FUNCTION are in effect, ON KEY takes precedence.

ON KEY LABEL The *<key label>* for the standard alphanumeric keys is simply the character on that key, for example, *A*, *b*, *2*, or *@*. Use the following key label names to assign special keys or key combinations with ON KEY LABEL *<key label>*.

| Key identification | <key label> |
|--------------------|--|
| Backspace | <i>Backspace</i> |
| Back Tab | <i>Backtab</i> |
| Delete | <i>Del</i> |
| End | <i>End</i> |
| Home | <i>Home</i> |
| Insert | <i>Ins</i> |
| Page Up | <i>PgUp</i> |
| Page Down | <i>PgDn</i> |
| Tab | <i>Tab</i> |
| Left arrow | <i>Leftarrow</i> |
| Right arrow | <i>Rightarrow</i> |
| Up arrow | <i>Uparrow</i> |
| Down arrow | <i>Dnarrow</i> |
| F1 to F10 | <i>F1, F2, F3, ...</i> |
| Control+<key> | <i>Ctrl-<key></i> or <i>Ctrl+<key></i> |
| Shift+<key> | <i>Shift-<key></i> or <i>Shift+<key></i> |
| Alt+<key> | <i>Alt-<key></i> or <i>Alt+<key></i> |
| Enter | <i>Enter</i> |
| Escape | <i>Esc</i> |
| Space bar | <i>Spacebar</i> |

Example The following example displays selected fields from 10 records, pauses for 3 seconds and adds more records to the screen. ON KEY LABEL command is used to branch the program to procedures that either reverse the record pointer, enter a browse window, or exit scrolling:

```
CLEAR
SET TALK OFF
PUBLIC mExit
mExit=.F.
ON KEY LABEL F8 DO BackUp
ON KEY LABEL F9 DO Details
ON KEY LABEL F10 DO GetOut
USE Clients Order Company
```



```

? CENTER("INSTRUCTIONS")
? CENTER("Press F8 to go back X records")
? CENTER("Press F9 to Browse all fields")
? CENTER("Press F10 to exit")
?
WAIT "Slow Browse - Press any key when ready"
Cnt=1
DO WHILE .NOT. EOF() .AND. .NOT. mExit
    DISPLAY NEXT 10 Company, Contact
    Pause=INKEY(3)
ENDDO
SET TALK ON
ON KEY
RETURN

PROCEDURE BackUp
CLEAR
SKIP -9
RETURN

PROCEDURE Details
SKIP -9
BROWSE
RETURN

PROCEDURE GetOut
mExit=.T.
RETURN

```

See Also INKEY(), KEYBOARD, ON ESCAPE, SET FUNCTION

onInitiate

Event fired when a client application requests a DDE link with *Visual* dBASE as the server, and no DDETopic object for the specified topic exists in memory.

Parameters <topic expC> The requested DDE topic.

Property of _app object

Description The *onInitiate* event executes a DDE (Dynamic Data Exchange) initiation-handler routine. You write this routine to create DDETopic objects, which handle DDE server events.

A DDE client application initiates a DDE link by specifying the DDE service name and a topic. The *_app* object's *ddeServiceName* property contains the service name for the current instance of *Visual* dBASE; the default is "VDB". You may change the *ddeServiceName* if there is more than one instance of *Visual* dBASE running, or if you want to identify your *Visual* dBASE application with a specific DDE service name.

Once the client application locates the desired *Visual* dBASE server by service name, it attempts to create a link on a specific topic. If a DDETopic object already exists in memory for the named topic, that object is used and the link is completed. If there is no DDETopic object for that topic, the *onInitiate* event fires. The *onInitiate* event handler must then create the DDETopic object, using that topic as a parameter to the constructor, and RETURN the resulting object to complete the link.

Example See the InitStockDDE() function in the class DDETopic example.

See also class DDETopic

onInitMenu

Specifies code that executes when a menubar or popup is opened.

Property of MenuBar, Popup

Description OnInitMenu is called whenever a menubar or popup is invoked, and is processed before the menubar's child menus or the popup is displayed.

You can use onInitMenu to determine the status of menu items that will be displayed. For example, use onInitMenu to determine if the *enabled* or *checked* property of a menu item should be true or false.

Example

```
Parameter FormObj
NEW SAMPLEMENU(FormObj, "Root")
CLASS SAMPLEMENU(FormObj, Name) OF MENUBAR(FormObj, Name)
    this.OnInitMenu = {; ? "Menu opened!"}
    DEFINE MENU FILE OF THIS;
        PROPERTY;
            Text "&File"
        DEFINE MENU EXIT OF THIS.FILE;
            PROPERTY;
                Text "E&xit",;
                OnClick {; Form.Close()}
    ENDClass
```

See Also Checked, Enabled

onUpdate

Fires repeatedly while application is idle to refresh toolbuttons.

Property of ToolBar

Description This event maintains the status of toolbuttons on a toolbar by firing whenever the application hosting the toolbar is idle.

separator

Determines if a menu item is a separator line instead of a menu option.

Property of Menu

Description Set Separator to True when you want to use a menu item as a separator between groups of menu commands. When Separator is true, other properties such Text and onClick are ignored.

Example For example, a menu titled Accounting might use a separator to emphasize the distinction between Accounts Receivable items and Accounts Payable items.

```
DEFINE FORM f1
DEFINE MENU Main OF f1
DEFINE MENU mOpt1 OF f1.Main;
  PROPERTY;
  Text "Option 1"
DEFINE MENU mSlct1 OF f1.Main.mOpt1;
  PROPERTY;
  Text "Select 1"
DEFINE MENU mLine1 OF f1.Main.mOpt1;
  PROPERTY;
  Separator .T.
DEFINE MENU mSlct2 OF f1.Main.mOpt1;
  PROPERTY;
  Text "Select 2"
DEFINE MENU mLine2 OF f1.Main.mOpt1;
  PROPERTY;
  Separator .T.
DEFINE MENU mSlct3 OF f1.Main.mOpt1;
  PROPERTY;
  Text "Select 3"
OPEN FORM f1
```

See Also none

SET CONFIRM

Controls the cursor's movement from one entry field to the next during data entry.

Syntax SET CONFIRM ON | off

Description When SET CONFIRM is OFF, *Visual* dBASE moves the cursor immediately to the next input area when the current one is full. When SET CONFIRM is ON,

the cursor moves to the next input area only when you press *Enter* or a cursor-control key, or when you click another input area with the mouse.

Use SET CONFIRM ON to prevent moving the cursor from one input area to the next automatically, thus avoiding data-entry errors such as the overflow of contents from one input area into the next. Use SET CONFIRM OFF when input speed is more important.

SET CUAENTER

Determines whether *Enter* simulates *Tab* in a form.

Syntax SET CUAENTER ON | off

Default The default for SET CUAENTER is ON. To change the default, update the CUAENTER setting in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the CUAENTER parameter directly in VDB.INI.

Description The CUA (Common User Access) interface standard dictates that the *Tab* key moves focus from one control to another, while the *Enter* key submits the entire form (which fires the form's *onSelection* event). Use SET CUAENTER to control whether *Enter* follows the CUA standard. If SET CUAENTER is OFF, the *Enter* key emulates the *Tab* key, moving the focus to the next control.

There are two good reasons to ignore the CUA behavior:

- In many forms, especially ones that take advantage of the numeric keypad, using the *Enter* key to move to the next control speeds data entry.
- For non-trivial forms, there are usually a number of pushbuttons which take completely different actions; for example, adding a new record, deleting the current record, navigating forward, navigating backward, etc. If you press *Enter* while the focus is in an entryfield for example, the act of submitting the form tells you nothing about what the user wants to do next.

In fact, few applications consider the action of submitting the form; the *onSelection* event is rarely used. When the *onSelection* event is left undefined, nothing happens when you press *Enter* (except in a control like the Editor). So you might as well make the *Enter* key do something useful and have it move the focus to the next control.

Regardless of the setting of SET CUAENTER, you can move focus from object to object with the mouse or by pressing *Tab* and *Shift-Tab*.

Example The following example creates a basic form with three entry fields from the Contact table and two pushbuttons to advance or retard the record pointer. SET CUAENTER OFF is used to make the cursor behave as it would in a DOS application, that is, advance between fields or to the pushbuttons when the user presses *Enter*. Changing the command to SET CUAENTER ON causes the cursor to respond only to the TAB key and mouse clicks:

```

SET PROCEDURE TO PROGRAM(1) ADDITIVE
DEFINE FORM Main FROM 0,0 to 13,40;
    PROPERTY ColorNormal "BG+/BG",;
    Text "System Update"
DEFINE TEXT T1 OF Main AT 3,5 ;
    PROPERTY TEXT "Enter Current Time (24 hour):",;
    Width 30
DEFINE ENTRYFIELD F1 OF Main AT 3,28 ;
    PROPERTY Value SPACE(8), Picture "99:99:99",;
    Width 8
DEFINE TEXT T2 OF Main AT 5,5 ;
    PROPERTY TEXT "Enter Current Date",;
    Width 20
DEFINE ENTRYFIELD F2 OF Main AT 5,28 ;
    PROPERTY Value {}, Picture "99/99/99",;
    Width 8
DEFINE PUSHBUTTON Update OF Main AT 9,7;
    PROPERTY TEXT "Update System Time and Date",;
    Height 2, Width 26, OnClick Update
OPEN FORM Main

PROCEDURE Update
SET DATE TO DTOC(Form.F2.Value)
SET TIME TO Form.F1.Value
? "Update complete"
CLOSE FORM Main
RETURN

```

See Also onSelection

SET ESCAPE

Specifies whether pressing *Esc* interrupts program execution.

Syntax SET ESCAPE ON | off

Default The default for SET ESCAPE is ON. To change the default, set the ESCAPE parameter in VDB.INI.

Description The primary purpose of the Esc key is to interrupt command or program execution. While this behavior may be changed with ON ESCAPE, this behavior occurs only when SET ESCAPE is ON.

Typically, SET ESCAPE is ON during application development. This allows you to stop processes which are taking too long or have run amok. When an application is deployed, you should either:

- SET ESCAPE OFF so that the user cannot cause your application to terminate abnormally, or

- Define a specific ON ESCAPE behavior so that your application or process can shutdown or be cancelled gracefully, usually after confirming that the user really wants to do so.

Note If SET ESCAPE is OFF and you have not used ON KEY or some other method to interrupt your program, you can interrupt program execution only by forcing the termination of *Visual* dBASE or your *Visual* dBASE application. Forced termination can cause data loss.

Note that user interface elements such as menus, forms, and dialog boxes handle *Esc* differently, usually closing or dismissing that UI element. (For forms, this behavior is controlled by its *escExit* property.) In those cases, ON ESCAPE and SET ESCAPE have no effect. In fact, with the exception of dialog boxes and forms opened with ReadModal(), because of the event-driven nature of *Visual* dBASE there is no program executing when you use a menu or type into a form, so there is nothing to interrupt.

Example In the following example there is a bug in the subroutine WontWork. The instructions and the loop test do not correspond. With SET ESCAPE ON, you can press the Escape key to interrupt the program. You can then choose SUSPEND to examine the variable MORE or to follow the program through the loop:

```
SET ESCAPE ON
DO WontWork

PROCEDURE WontWork
More = ""
DO WHILE More <> "X"                                && should be Upper(More)<>"X"
? "Beginning the loop"
* ...
WAIT "Enter E to exit the loop" TO More
ENDDO
```

See Also ON ESCAPE, ON KEY

SET FUNCTION

Assigns a string to a function key or to a combination of the *Ctrl* (control) key or the *Shift* key and a function key.

Syntax SET FUNCTION <key> TO <expC>

<key> A function key number, function key name, or character expression of a function key name—for example, 3, F3, or "F3". Specify a character expression for <key> to assign a key combination using the Ctrl or Shift key with a function key. Type "CTRL+" or "SHIFT+" and then a function key name—for example, "shift+F5" or "Ctrl+f3". The function key names are not case-sensitive and you may use a hyphen in place of the plus sign. You can't combine Ctrl and Shift, such as "Ctrl+Shift+F3".

<expC> Any character string, often the text of a command. Use a semicolon (;) to represent the *Enter* key. Placing a semicolon at the end of a command has the effect of executing that command when you press the function key in the Command window. You can execute more than one command by separating each command in the list with a semicolon.

Default The following function key settings are in effect when *Visual* dBASE starts:

| Key | Command | Key | Command |
|-----|--------------------|-----|--------------------|
| F1 | HELP; | F7 | DISPLAY MEMORY; |
| F3 | LIST; | F8 | DISPLAY; |
| F4 | DIR; | F9 | APPEND; |
| F5 | DISPLAY STRUCTURE; | F10 | Activates the menu |
| F6 | DISPLAY STATUS; | | |

Description Use SET FUNCTION to simulate typing a string with a single keystroke. These strings are usually commands to be executed in the Command window, or common strings used in data entry.

Note F2 is reserved for toggling between views while in the Browse window. You can program it, but it will not be recognized when in the Browse window. You cannot program F10, or any combination using F11 or F12. You cannot program Shift-F4 or Shift-F5, which are shortcuts for the Window/Tile Vertically and Window/Cascade commands, respectively. You also cannot program keys that are used as standard Windows functions, such as Ctrl-F4.

When you press the programmed function key or key combination, the assigned string appears at the cursor. Strings for the Command window usually end in a semicolon, which represents the *Enter* key. The simulated *Enter* key causes the command to be executed immediately.

While SET FUNCTION is specifically intended to simulate typing a string, you can use the ON KEY command to program a function key or any other key to execute any command. For example, these two commands (executed separately, not consecutively):

```
set function f7 to "display memory;"
on key label f7 display memory
```

would both cause the F7 key to execute the DISPLAY MEMORY command if the key was pressed on a blank line in the Command window. But suppose the line in the Command window contained the word "field" and the cursor was at the beginning of that line. Then with SET FUNCTION F7, pressing the function key would cause the string "display memory" to be typed into the line, resulting in "display memoryfield", and then the *Enter* key would be simulated, causing *Visual* dBASE to attempt to display a field named "memoryfield" in the current workarea. With ON KEY LABEL F7, the DISPLAY MEMORY command would be executed with nothing being typed into the Command window.

If the cursor was in an entryfield for a city in a form, then with SET FUNCTION F7, you would get the city of "display memory" and the cursor would move to

the next control if SET CUAENTER was OFF. Again, with ON KEY LABEL F7, the DISPLAY MEMORY command would be executed without affecting the data entry.

To see the list of strings currently assigned to function keys, use DISPLAY STATUS.

Example The following example changes the *F8* key to the string "modify command ". Without the semicolon at the end, you type the name of the file you want to edit and press *Enter*:

```
set function f8 to "modify command "
```

See Also DISPLAY STATUS, ON KEY

SET MESSAGE

Specifies the default message to display in the status bar.

Syntax SET MESSAGE TO [<message expC>]
<message expC> The message to display

Description Use SET MESSAGE to set the default message that appears the status bar. Menu items and controls on forms have a *statusMessage* property. When that object has focus, and that property is not empty, that message is displayed instead.

SET MESSAGE TO without the option <message expC> sets the default message to an empty string, and removes any message from the status bar.

The status bar may be suppressed by setting the *_app.statusBar* property to *false*.

SET TYPEAHEAD

Sets the size of the typeahead buffer, where keystrokes are stored while *Visual* dBASE is busy.

Syntax SET TYPEAHEAD TO <expN>
<expN> the size of the keyboard typeahead buffer, any number from 0 to 1600.

Default The default size of the typeahead buffer is 50 characters. To change the default, set the TYPEAHEAD parameter in VDB.INI.

Description The keyboard typeahead buffer stores keystrokes the user enters while *Visual* dBASE is busy, for example while reindexing a table. When the processing is complete and the application is ready to accept keystrokes, *Visual* dBASE fetches and deletes the values in the buffer in the order they were entered. Any

keys typed while there are still keystrokes in the buffer are added to the end of the buffer.

If the size of the typeahead buffer is set to 50, dBASE can store values for 50 keypresses; further keystrokes are ignored without any warnings. A large typeahead buffer is useful if the user does not want to stop typing when dBASE is unavailable for processing direct keyboard input.

For some programs, you may want to disable the typeahead buffer with SET TYPEAHEAD TO 0. This ensures that user input comes directly from the keyboard, rather than from the typeahead buffer.

For example, if you want to be able to fill in multiple forms quickly, one after the other, you might SET TYPEAHEAD to a relatively high number during form processing. This would let you continue typing data while one form was being saved and the next (blank) one being displayed. The data you entered during processing would be entered onto the new form when it appeared. On the other hand, if you want to make sure that no data is entered until the form is displayed on the screen, you can issue SET TYPEAHEAD TO 0.

You can also clear the typeahead buffer manually with CLEAR TYPEAHEAD.

SET TYPEAHEAD limits the number of characters you can put into the typeahead buffer using KEYBOARD.

See Also CLEAR TYPEAHEAD, KEYBOARD

SHELL()

Hides or restores the components of the application shell: the Command window (and Navigator) and the MDI frame window. Returns a logical value corresponding to the previous SHELL() state.

Syntax SHELL([<expL1>, [<expL2>]])

<expL1> The value that determines whether to display the shell.

<expL2> The value that determines whether to force the display of the MDI frame window. If <expL1> is true, the full shell is on and <expL2> is ignored. If <expL1> is false, <expL2> defaults to false.

Description SHELL() controls the display of the components of the application shell:

- The Command window
- The Navigator
- The MDI frame window, which contains the Command window, Navigator, and all MDI forms and their toolbars and menus. This window is represented by the `_app.frameWin` object.

In *Visual* dBASE, SHELL() defaults to true; all three components are displayed. In a compiled application, SHELL() defaults to false; none of the elements are displayed, unless either:

- An MDI form is open, in which case the MDI frame window must be displayed to contain the MDI form(s), or
- A menu has been assigned to _app.frameWin.

In either case, the MDI frame window stays visible regardless of the <expL2> value.

Use SHELL(.F.) in programs to temporarily hide the standard dBASE environment, allowing your application to take control of the user's working environment. To restore the dBASE interactive environment, issue SHELL(.T.). The environment is also restored when the user closes the form that SHELL() is activated for.

SHELL(.F., .F.) operates differently when you are working in a form that is defined as a top-level MDI form (formname.MDI=.F.) or in a form that is not a top-level MDI form (formname.MDI=.T.).

- When formname.MDI=.F. for the active form, SHELL(.F., .F.) appears to remove dBASE from the user's system. The form name becomes the application name that appears in the Windows Task List in place of "*Visual* dBASE." This makes your application look like a standalone application, and is the typical use for SHELL().
- When formname.MDI=.T. for the active form, the menu system associated with the form appears as the menu at the top of the screen instead of at the top of the form. The user remains in dBASE, but the dBASE menu is replaced by the menu defined by the active form. However, the user can still access the SpeedBar if it is active. The user can click in the Command window to close the form.

Using SHELL() in a program has the same effect as changing the Visible property of the FrameWin object of _app, as shown in the following example. For more information, see _app.

```
SHELL(.F.)  && same effect as next line
_app.FrameWin.Visible = .F.
```

If you issue SHELL(.F.) in the Command window, you exit to Windows momentarily and then return to dBASE.

Example This example shows the code generated by the form designer for a Shell Test form that simply sets the left double click button to SHELL(.t.) and the right double click button to SHELL(.f.).

```
LOCAL f
f = NEW SHELL ( )
f.Open()

CLASS SHELL OF FORM
    this.OnRightDbClick = {shell(.t.)}
    this.OnLeftDbClick = {shell(.f.)}
```

```

this.EscExit = .T.
this.mdi = .f.
* set mdi=.t. to see effect with mdi
this.Text = "Shell Test .t."
this.Width =      48.00
this.Top =        2.00
this.Left =       2.00
this.Height =     15.00
this.Minimize = .F.
this.Maximize = .F.
ENDCLASS

```

When the form is activated e.g. with DO Shell.wfm, the Shell Test form appears on the screen (with mdi=.f.). Double clicking with the left mouse button makes other windows in the dBASE screen disappear. Double clicking with the right button makes them reappear. * When you set mdi=.t., the Shell Test form can be accessed by pressing Ctrl Tab to show the Windows windows.

See Also _app, DEFINE, QUIT, SET DESIGN

shortCut

Specifies a key combination that fires the OnClick event of a menu object.

Property of Menu

Description Use ShortCut to provide a quick way to execute a menu command with the keyboard. For example, if you assign the character string "CTRL+S" to a menu option's ShortCut property, the user can execute that menu option by pressing *Ctrl+S*.

The value you specify with ShortCut is displayed next to the prompt you specify with the Text property.

Example NEW operator syntax:

```

FileMnt= NEW MENU ITEM(this)
FileMnt.ShortCut = "Alt+F"

```

DEFINE object syntax:

```

DEFINE MENU ITEM FileMnt OF THIS;
PROPERTY ShortCut "Alt+F"

```

See Also onClick

SLEEP

Pauses a program for a specified interval or until a specified time.

Syntax SLEEP

<seconds expN> |
UNTIL <time expC> [, <date expC>]

<seconds expN> The number of seconds to pause the program. The number must be greater than zero and no more than 65,000 (a little over 18 hours). Fractional times are allowed. Counting starts from the time you issue the SLEEP command.

UNTIL <time expC> Causes program execution to pause until a specified time (<time expC>) on the current day. If you also specify <date expC>, the program pauses until the time on that day. The time and date *Visual* dBASE uses are the system time and date. You can set the system time with SET TIME and the system date with SET DATE TO. If the time has already passed, SLEEP UNTIL <time expC> has no effect.

The <time expC> argument is a 24-hour time that matches the format returned by the TIME() function. A typical format for <time expC> is "HH:MM:SS". The delimiter is conventionally a colon but can be changed through the Regional Settings in the Windows Control Panel. The time string must include the seconds.

<date expC> An optional date until which the program is to pause. The <date expC> argument is a character expression (*not* a date expression) that represents a date in the current date format; it would match the string returned by the DTOC() function. For example, if SET DATE is AMERICAN, the format would be "MM/DD/YY".

If the date has already passed, SLEEP UNTIL <time expC> [, <date expC>] has no effect. If you want to specify a value for <date expC>, you must also specify a value for <time expC>.

Description Use SLEEP to pause a program either for <seconds expN> seconds or until a specified time (<time expC>). The specified time is the same day the program is running unless you specify a date with <date expC>. If SET ESCAPE is ON, you can interrupt SLEEP by pressing *Esc*.

Note If SET ESCAPE is OFF, there is no way to interrupt SLEEP. However, you can use *Ctrl+Esc* and *Alt+Tab* to switch to another Windows application, or *Alt+F4* to exit *Visual* dBASE.

Although SLEEP can generate a pause from the Command window, programmers use it primarily within programs. For example, you can use SLEEP to generate a pause between multiple displaying windows or to allow a user to read a message on the screen or complete an action. Pauses are also useful when you need to delay program execution until a specific time.

While SLEEP is active, *Visual* dBASE is considered busy; that is, busy sleeping. Program execution is suspended, keystrokes go into the typeahead buffer, and *Visual* dBASE does not respond to events like mouse clicks or timers. If you want an event to occur at a specified time without putting *Visual* dBASE to sleep, use a Timer object.

SLEEP is an alternative to using a DO WHILE loop, a FOR loop, or WAIT to generate pauses in a program. SLEEP is more accurate than using loops because it's independent of the execution speed of the system. You can also use INKEY(<expN>) if you want the user to be able to interrupt the pause and continue with program processing.

Example The following example uses SLEEP to delay execution for five seconds:

```
sleep 5
```

The next example uses SLEEP to delay execution until 7:30 p.m. on the same day the program is running. If it's already past 7:30 p.m., execution is delayed until that time the next day:

```
#define SLEEP_TIME "19:30:00"
if time() > SLEEP_TIME
    sleep until SLEEP_TIME, dtoc( date() + 1 )
else
    sleep until SLEEP_TIME
endif
```

The last example uses SLEEP to delay execution until 11:59:59 p.m. on December 31, 1999 (SET DATE is AMERICAN):

```
sleep until "23:59:59", "12/31/99"
```

See Also class Timer, INKEY(), SET DATE TO, SET TIME, WAIT

trackRight

Determines if the user can select a popup menu item with a right mouse click.

Property of Popup

Description When TrackRight is true (the default), users can select popup menu items with either the right mouse button or the left mouse button.

Set TrackRight to false if you don't want users to be able to select items from a popup menu with a right mouse click.

Example

```
f = NEW Form()
DEFINE POPUP p OF f;
PROPERTY;
TrackRight .F.
```

See Also onClick

uncheckedBitmap

A bitmap to display when a menu item is not *checked*.

Property of Menu

Description Use *uncheckedBitmap* to display a bitmap when a menu item's *checked* property is *false*. If no bitmap is specified, nothing is displayed when a menu item is not *checked*.

The *uncheckedBitmap* setting can take one of two forms:

- RESOURCE <resource id> <dll name>
specifies a bitmap resource and the DLL file that holds it.
- FILENAME <filename>
specifies a bitmap file. See class Image for a list of bitmap formats supported by Visual dBASE.

Note The display area in the menu item is very small (13 pixels square with Small fonts). If the bitmap is too large, the top left corner is displayed. Also, the color of the bitmap when the menu item is highlighted changes, depending on the system menu highlight color. Therefore, you may want to limit yourself to simple monochrome bitmaps.

See Also *checked*, *checkedBitmap*

WAIT

Pauses the current program, displays a message in the results pane of the Command window, and resumes execution when any key is pressed. The keystroke may be stored in a variable.

Syntax WAIT [<prompt expC>] [TO <memvar>]

<prompt expC> A character expression that prompts the user for input. If you don't specify <prompt expC>, Visual dBASE displays "Press any key to continue..." when you issue WAIT.

TO <memvar> Assigns a single character to the memory variable you specify for <memvar> as a character-type variable. If <memvar> doesn't exist, Visual dBASE creates it. If <memvar> does exist, WAIT overwrites it.

Description Use WAIT to halt program execution temporarily. Pressing any key exits WAIT and resumes program execution. WAIT is usually used during application development to display information and create simple breakpoints. It is usually not used in deployed applications.

In simple test applications, you can also WAIT to get a single key or character. If the user presses *Enter* without typing any characters, WAIT assigns an empty string ("") to <memvar>.

Note If SET ESCAPE is ON, pressing *Esc* at the WAIT prompt causes Visual dBASE to interrupt program execution. If SET ESCAPE is OFF, pressing *Esc* in response to WAIT causes program execution to resume the same as any other key.

Example WAIT can be used without a prompt or an input variable:

```
WAIT
```

The default prompt then appears on the next row, in column 0:

```
* Press any key to continue ...
```

The following example shows WAIT with a prompt:

```
WAIT "Press any key to move to the next screen"
```

This examples shows WAIT with a prompt and input variable:

```
WAIT "Do you wish to print the report (Y/N)? " to Answer
```

See Also SET ESCAPE

windowMenu

Specifies a menu object that displays a list of all open MDI windows.

Property of MenuBar

Description WindowMenu contains a reference to a menu object that has a menubar as its parent. When users open this menu object, dBASE displays a pulldown list of all open MDI windows.

WindowMenu automatically places a separator line on the pulldown list between any menu prompts and the list of open windows. The currently active window shows a check next to the window name.

If you use the Menu Designer to create a menubar, WindowMenu is automatically set to an item named Window on the menubar:

```
this.WindowMenu = this.Window
```

Example

```
NEW SAMPLEMENU(FormObj, "Root")
CLASS SAMPLEMENU(FormObj, Name) OF MENUBAR(FormObj, Name)
    DEFINE MENU FILE OF THIS;
        PROPERTY;
        Text "&File"
        DEFINE MENU EXIT OF THIS.FILE;
            PROPERTY;
            Text "E&xit"
    DEFINE MENU EDIT OF THIS;
        PROPERTY;
        Text "&Edit"
        DEFINE MENU UNDO OF THIS.EDIT;
            PROPERTY;
            Text "&Undo"
        DEFINE MENU CUT OF THIS.EDIT;
            PROPERTY;
            Text "Cu&t"
        DEFINE MENU COPY OF THIS.EDIT;
```

```
        PROPERTY;  
        Text "&Copy"  
    DEFINE MENU PASTE OF THIS.EDIT;  
        PROPERTY;  
        Text "&Paste"  
    DEFINE MENU WINDOW OF THIS;  
        PROPERTY;  
        Text "&Window"  
    DEFINE MENU ARRANGE OF THIS.WINDOW;  
        PROPERTY;  
        Text "&Arrange"  
    DEFINE MENU HELP OF THIS;  
        PROPERTY;  
        Text "&Help"  
    DEFINE MENU ABOUT OF THIS.HELP;  
        PROPERTY;  
        Text "&About"  
    This.EditUndoMenu = This.Edit.Undo  
    This.EditCutMenu  = This.Edit.Cut  
    This.EditCopyMenu = This.Edit.Copy  
    This.EditPasteMenu = This.Edit.Paste  
    This.WindowMenu   = This.Window  
ENDCLASS
```

See Also CLASS MENUBAR, *editCopyMenu*, MDI

Report objects

Report objects generate formatted output from data in tables. The Report wizard and Report designer allow you to create and modify reports visually. Reports are saved as code in a .REP file that you can modify.

Measurements in reports default to *twips* (20th of a point). There are exactly 1440 twips per inch.

At the top of the report object class hierarchy is the Report class. A Report object acts as a container for four main groups of objects:

- Data access objects, which give access to data in tables
 - Query objects
 - Database objects
 - Session objects

These objects are created and used the same way they are in forms, except that a report does not have a primary rowset like a form does.

- Report layout objects, which determine the appearance of the page and where data is output, or streamed
 - PageTemplate objects
 - StreamFrame objects

A Report object contains one or more PageTemplates, and each PageTemplate usually contains one or more StreamFrames.

- Data stream objects, which read and organize the data from a query's rowset and stream it out to a report's StreamFrame objects
 - StreamSource objects
 - Band objects
 - Group objects

Each StreamSource object contains a Band object that is assigned to its *detailBand* property. The contents of the *detailBand* are rendered for each row

in the rowset. A `StreamSource` may also have one or more `Group` objects, which group data and have their own header and footer `Band` objects.

- Visual components—objects that display the report’s data
 - Text objects
 - Image objects
 - Line objects
 - Rectangle objects
 - Shape objects
 - `CheckBox` objects
 - `RadioButton` objects

These objects are created as properties of a `PageTemplate` object if they are fixed elements on the page, such as a report’s date and page number; otherwise they are properties of a `Band` object and are used to display data.

The primary method of displaying information in a report is through `Text` objects. For text that varies, such as the data from the rowset, the *text* property of the `Text` object is set to an expression codeblock, which is evaluated every time the object is rendered. By using an expression in the codeblock that accesses the fields in the rowset, the `Text` object displays data from tables.

You may use the other visual components in a report to display static images or images from a table, draw lines, or display table data with check boxes or radio buttons.

Note Visual component objects are used in forms as well as reports, and most of the properties, methods, and events associated with the objects are described in Chapter 15, “Form objects.” Some `Text` object properties used only in reports are described in this chapter.

A simple report example

To get a sense of how everything fits together, imagine a report of students grouped by grade, with the total number of students in each grade.

The report has a query that accesses the table of students, named *students1*; a `StreamSource` object, by default named *streamSource1*, to stream the data from the query; and a `PageTemplate` object, by default named *pageTemplate1*, that describes the physical attributes of the page, such as its dimensions, background color, and margins.

pageTemplate1 contains one `StreamFrame` object, by default named *streamFrame1*, where the data stream will be rendered. It occupies most of the space inside *pageTemplate1*’s margins. The rest of the space is used by `Text` components that display the report title, date, and page.

streamFrame1 has a *streamSource* property that identifies its `StreamSource` object. It is assigned *streamSource1*.

streamSource1 has a *rowset* property that identifies the `StreamSource` object’s rowset. It is assigned *students1.rowset*.

students1.rowset and *streamFrame1* are now linked. To fill *streamFrame1* with data, the report engine will traverse *students1.rowset*, from the first row to the last row. But at this point, no data will be displayed, because there are no visual components in any Band objects.

Text components are assigned to *streamSource1.detailBand*. The *text* properties of these components are expression codeblocks that refer to the *value* properties of the fields of the *rowset* of the StreamSource object. For example, the *text* of the Text component that displays the student's last name is

```
{||this.form.students1.rowset.field[ "Last name" ].value}
```

When a visual component is placed in a report, its *form* property refers to the report.

To group the data, a Group object, named *group1* by default, is assigned to *streamSource1*. Its *groupBy* property contains the name of the group field, "Grade". The report engine will watch the value of this field in the rowset, that is:

```
students1.rowset.field[ "Grade" ].value
```

and whenever the value of the field changes, a new group begins. Therefore, it's important that the data is sorted by grade. If the report's *autoSort* property is *true*, all of the report's queries will automatically be sorted to match the groups in the StreamSource objects.

group1 has two Band objects of its own: a header band and a footer band, assigned to the *headerBand* and *footerBand* properties respectively. The *headerBand* is currently empty, and the *footerBand* displays the count of the students in that grade.

The Group object's *agCount()* method counts the number of rows in the group. To display that number, the *text* of the Text component in the *footerBand* is set to the following expression codeblock:

```
{||"Count: " + this.parent.parent.agCount({||this.parent.rowset.fields["ID"].value))}
```

The expression codeblock concatenates the text label with the return value of the Group object's *agCount()* method. To get to that method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The *agCount()* method expects a code reference as a parameter that it can evaluate. If the return value is not *null*, the count is incremented. The code reference here is another expression codeblock that uses dot operators:

- *this* is the Group object *group1*.
- *group1*'s *parent* is *streamSource1*.
- *streamSource1*'s *rowset* is *students1.rowset*, the rowset that the report engine is traversing to fill *streamFrame1*.

That's all the objects that go into a report of students, grouped by grade, with the number of students in each grade. There are two final details that are needed to make the report work.

Because a report can have multiple PageTemplate objects, a Report object has a *firstPageTemplate* property that refers to the PageTemplate object to use for the first page. It is assigned *pageTemplate1*.

Each PageTemplate object has a *nextPageTemplate* property that refers to the PageTemplate object to use when the current page is done. For *pageTemplate1*, it is assigned a reference to itself. This means that the same page layout is used for every page in the report.

Everything described in this sample report can be handled automatically by the Report Wizard. To run the report, call the Report object's *render()* method.

How a report is rendered

When a Report object's *render()* method is called, the first thing the report does is check its *firstPageTemplate* property to find the first page to render. It renders the page by rendering all the components and StreamFrame objects assigned to it, in the order they were originally created (the same order as they appear in the class definition in the .REP file).

To render a StreamFrame object, Visual dBASE looks to its *streamSource* property. The Band objects in that StreamSource object—the *detailBand* and the *headerBand* and *footerBand* of any groups—are rendered in the StreamFrame object to fill it with data.

Before each component in the band is rendered, its *canRender* event fires. The *canRender* event can be used to supplement the *suppressIfBlank* and *suppressIfDuplicate* properties of the Text component by returning *false*, but it is more often used to alter the properties of a component just before it is rendered. For example, you can set a component's *colorNormal* to red if it's going to display a negative number. When used this way, the *canRender* event handler does what it wants and returns *true*, so that component is rendered. After the component is rendered, its *onRender* event fires. You can use the *onRender* event to reset the component to its original state.

Until the data from the StreamSource object is exhausted—that is unless the StreamSource object's *rowset* reaches the end-of-set—Visual dBASE knows that it needs to fill another StreamFrame. If there is another StreamFrame object in the same PageTemplate that used the same *streamSource*, the report engine will continue to stream bands from that StreamSource into that StreamFrame.

For example, if a PageTemplate has three tall StreamFrame objects side-by-side that have the same *streamSource* property, data would be printed in three columns on each page. To create a page of labels, create one StreamFrame for each label, all with the same *streamSource* property. Then set the *beginNewFrame* property of the *streamSource*'s *detailBand* to *true*, so that each row of data is rendered in a new StreamFrame.

If there are no more StreamFrame objects that can be filled on the current page, another page is scheduled. The current PageTemplate object's *nextPageTemplate* property refers to the PageTemplate to use.

Once the current page has finished rendering, the Report object's *onPage* event fires. If there is another page scheduled, it is rendered. Its StreamFrame objects are filled with data and the process repeats itself until all the StreamSource objects are exhausted. The *onPage* event fires one last time and the report is done.

class Band

Contains the objects to output for a single row in a stream, or the header or footer of a group.

Syntax These objects are automatically created by the StreamSource and Group objects.

Properties The following table lists the properties and events of the Band class. (No methods are associated with this class.)

| Property | Default | Description |
|----------------------|---------|---|
| <i>beginNewFrame</i> | false | Whether rendering always starts in a new StreamFrame |
| <i>className</i> | BAND | Identifies the object as an instance of the Band class (Property discussed in Chapter 5, "Core language.") |
| <i>context</i> | Normal | The context in which the band is being rendered: (0=Normal) or for (1=For Drilldown summary) |
| <i>expandable</i> | true | Whether the band will increase in size automatically to accommodate the objects within it |
| <i>firstOnFrame</i> | | Whether the band is being rendered for the first time in a StreamFrame. |
| <i>height</i> | 0 | The height of the band in the Report object's current metric units. (See page 15-107.) |
| <i>name</i> | | The name of the Band object. (See page 15-123.) |
| <i>parent</i> | | The StreamSource or Group object that contains the Band (Property discussed in Chapter 5, "Core language.") |
| <i>renderOffset</i> | | The offset of the bottom of the band from the top of the current stream frame. |
| <i>streamFrame</i> | | The StreamFrame object in which the band is currently rendering. |
| <i>visible</i> | | Whether the band is visible. (See page 15-181.) |

| Event | Parameters | Description |
|------------------|------------|--|
| <i>onRender</i> | | After the contents of the band have rendered |
| <i>preRender</i> | | Before the contents of the band are rendered |

Description A Band object acts as a container for visual components. They are created automatically for StreamSource and Group objects and cannot be created manually. There are three kinds of Band objects: detail bands, header bands, and footer bands.

A detail band is assigned to a StreamSource's *detailBand* property. The contents of the band are output once for each row in the StreamSource's rowset. Header and footer bands are assigned to a Group object's *headerBand* and *footerBand* properties respectively. They are rendered at the beginning and end of each group.

For a detail band, setting its *beginNewFrame* property to *true* causes each row from the StreamSource's rowset to be rendered in a new StreamFrame, which is the desired behavior when creating labels.

For a summary-only report, leave the detail band empty and set its *height* to zero.

When a band's *expandable* property is *true* and it contains components, the band will expand to show those components, even if its *height* is set to zero.

See also class StreamSource, class Group

class Group

Describes a group in a report.

Syntax [*<oRef>* =] new Group(<streamSource>)

<oRef> A variable or property—typically of <streamSource>—in which you want to store a reference to the newly created Group object.

<streamSource> The StreamSource object to which the Group object binds itself.

Properties The following tables list the properties and methods of the Group class. (No events are associated with this class.)

| Property | Default | Description |
|-------------------------|---------|--|
| <i>className</i> | GROUP | Identifies the object as an instance of the Group class. (Property discussed in Chapter 5, "Core language.") |
| <i>drillDown</i> | None | How the group's bands are displayed in drilldown format. |
| <i>footerBand</i> | | Specifies a Band that renders after a group of detail bands. |
| <i>groupBy</i> | | A character string containing the field name by which groups are formed. If blank, the group is for the entire report. |
| <i>headerBand</i> | | Specifies a Band that renders before a group of detail bands. |
| <i>headerEveryFrame</i> | false | Specifies whether to repeat the <i>headerBand</i> when a Group spans more than one StreamFrame. |

| Property | Default | Description |
|---------------|---------|--|
| <i>name</i> | | The name of the Group object. (See page 15-123.) |
| <i>parent</i> | | The Report or StreamSource object that contains the Group. (Property discussed in Chapter 5, “Core language.”) |

| Method | Parameters | Description |
|-------------------------------|-------------|---|
| <i>agAverage()</i> | <codeblock> | Aggregate method that returns the mean average for a group |
| <i>agCount()</i> | <codeblock> | Aggregate method that returns the number of items in a group |
| <i>agMax()</i> | <codeblock> | Aggregate method that returns the highest value within a group |
| <i>agMin()</i> | <codeblock> | Aggregate method that returns the lowest value in a group |
| <i>agStandardDeviation()</i> | <codeblock> | Aggregate method that returns the standard deviation of the values in a group |
| <i>agSum()</i> | <codeblock> | Aggregate method that returns the total of a group |
| <i>agVariance()</i> | <codeblock> | Aggregate method that returns the variance of the values in a group |
| <i>release()</i> | | Explicitly releases the Group object from memory. (See page 5-34.) |

Description Use Group objects to group data and calculate aggregate values for the group. Groups may be nested, and are handled in the order that they are created (the same order that they appear in the class definition in a .REP file).

The *groupBy* property contains the name of the field that defines the group, and may include an optional ascending or descending modifier. Whenever the value of that field changes, a new group starts. Therefore, the data must be sorted on the grouping field(s).

A Group object's *headerBand* is rendered before each group and its *footerBand* is rendered afterward. If the *headerEveryFrame* property is *true*, the group's *headerBand* is rendered at the beginning of every StreamFrame.

If the Report object's *autoSort* property is *true*, data in a report is automatically sorted to match groups.

The Report object has its own Group object that is referred to by its *reportGroup* property. Its *groupBy* property is an empty string, and the group is used for report-wide aggregates.

You may organize the report in *drilldown* format: the header and footer bands showing summary information are displayed first, followed by the detail rows. This allows you to see summary information at the top, and then “drill down” to the supporting data.

See also class Report, class Band

class PageTemplate

Describes the layout of a page of a report.

Syntax [*<oRef>* =] new PageTemplate(*<report>*)

<oRef> A variable or property—typically of *<report>*—in which you want to store a reference to the newly created PageTemplate object.

<report> The Report object to which the PageTemplate object binds itself.

Properties The following tables list the properties and methods of the PageTemplate class. (No events are associated with this class.)

| Property | Default | Description |
|-------------------------|-----------------------|--|
| <i>background</i> | | Background image on the page. (See page 15-63.) |
| <i>className</i> | PAGETEMPLATE | Identifies the object as an instance of the PageTemplate class (Property discussed in Chapter 5, “Core language.”) |
| <i>colorNormal</i> | white | Background color for the page. (See page 15-75.) |
| <i>gridLineWidth</i> | 0 | Width of lines around elements in the report (0=no grid lines). (See page 15-104.) |
| <i>height</i> | | Height of the page in current metric units. (See page 15-107.) |
| <i>marginBottom</i> | .75 inch = 1080 twips | The space between the bottom of the page and the usable area of the PageTemplate |
| <i>marginLeft</i> | .75 inch = 1080 twips | The space between the left side of the page and the usable area of the PageTemplate |
| <i>marginRight</i> | .75 inch = 1080 twips | The space between the right side of the page and the usable area of the PageTemplate |
| <i>marginTop</i> | .75 inch = 1080 twips | The space between the top of the page and the usable area of the PageTemplate |
| <i>name</i> | | The name of the PageTemplate object. (See page 15-123.) |
| <i>nextPageTemplate</i> | | The PageTemplate object that is used for the following page |
| <i>parent</i> | | The Report object that contains the PageTemplate (Property discussed in Chapter 5, “Core language.”) |
| <i>width</i> | | Width of the page in current metric units. (See page 15-183.) |

| Method | Parameters | Description |
|------------------|------------|---|
| <i>release()</i> | | Explicitly releases the PageTemplate object from memory. (See page 5-34.) |

Description A PageTemplate object describes the layout of a page, including its background color or image. It acts as a container for StreamFrame objects and visual

components, which represent fixed output, such as a report date and page number.

The location of these objects is relative to (and restricted by) the four *margin*-properties that dictate the usable area of the page. Changing the *marginLeft* or *marginTop* will move everything that's inside the PageTemplate.

Although you may create multiple PageTemplate objects in a report, for example a different first page or alternating odd and even pages, the Report Designer currently does not support multiple PageTemplate objects visually.

See also class Report, class StreamFrame

class Report

A container and controller of report elements.

Syntax [*<Ref>* =] new Report()

<Ref> A variable or property in which you want to store a reference to the newly created Report object.

Properties The following tables list the properties, events, and methods of the Report class.

| Property | Default | Description |
|--------------------------|---------|--|
| <i>autoSort</i> | true | Whether to automatically sort data to match specified groups |
| <i>className</i> | REPORT | Identifies the object as an instance of the Report class (Property discussed in Chapter 5, "Core language.") |
| <i>endPage</i> | -1 | Last page number to render (-1 for no limit) |
| <i>elements</i> | | An array containing object references to the visual components on the reports |
| <i>firstPageTemplate</i> | | Reference to the first PageTemplate object, which describes the first page |
| <i>inDesign</i> | | Whether the report was instantiated by the Report designer |
| <i>mdi</i> | | Whether the report window is an MDI window |
| <i>metric</i> | Twips | Units of measurement (0=Chars, 1=Twips, 2=Points, 3=Inches, 4=Centimeters, 5=Millimeters, 6=Pixels) |
| <i>output</i> | Default | Target media (0=Window, 1=Printer, 2=Printer file, 3=Default, 4=HTML, 5=HTML file) |
| <i>outputFilename</i> | | Name of file if output goes to printer or HTML file |
| <i>printer</i> | | An object describing various printer output options |
| <i>reportGroup</i> | | Reference to a Group object for the report as a whole, for master counts and totals |
| <i>reportPage</i> | | Current page number being rendered |
| <i>reportViewer</i> | null | Reference to the ReportViewer object that instantiated the report, if any. |

| Property | Default | Description |
|----------------------|---------|--|
| <i>scaleFontBold</i> | false | When the <i>metric</i> is Chars, determines whether the Char units of the ScaleFont assume that the font is bold |
| <i>scaleFontName</i> | Arial | When the <i>metric</i> is Chars, the typeface of the font used as the basis of measurement |
| <i>scaleFontSize</i> | 10 | When the <i>metric</i> is Chars, the point size of font used as the basis of measurement |
| <i>startPage</i> | 1 | First page number to output |
| <i>title</i> | | Title of the report; appears in the title bar of the preview window. |

| Event | Parameters | Description |
|---------------------|------------|---|
| <i>onDesignOpen</i> | | After the report is first loaded into the Report Designer. (See page 15-130.) |
| <i>onPage</i> | | After a page is rendered |

| Method | Parameters | Description |
|---------------------|------------|---|
| <i>close()</i> | | Closes the report window |
| <i>isLastPage()</i> | | Determines whether there are any more pages to render |
| <i>release()</i> | | Explicitly releases the Report object from memory. (See page 5-34.) |
| <i>render()</i> | | Generates the report |

Description A Report object acts as the controlling container for all the objects that make up the report, including data access, page layout, and data stream objects.

The *reportGroup* property refers to a report-level Group object that can be used for report-wide summaries. This Group object is created automatically.

To generate the report, call its *render()* method. The report's *output* property determines where the report is rendered: to the screen, a printer, or a file. The report's *printer* object contains properties that control output to a printer (or printer file). Call the *printer* object's *choosePrinter()* method before calling *render()* to allow the user to choose a printer.

You can control the pages that are output by setting the *startPage* and *endPage* properties.

See also class PageTemplate, class StreamSource, class Group

class StreamFrame

Describes an area on a page into which output is streamed.

Syntax [*<Ref>* =] new StreamFrame(*<pageTemplate>*)

<oRef> A variable or property—typically of *<pageTemplate>*—in which you want to store a reference to the newly created StreamFrame object.

<pageTemplate> The PageTemplate object to which the StreamFrame object binds itself.

Properties The following table lists the properties and events of the StreamFrame class. (No methods are associated with this class.)

| Property | Default | Description |
|-------------------------|-------------|--|
| <i>borderStyle</i> | Default | The border around the StreamFrame object (0=Default, 1=Raised, 2=Lowered, 3=None, 4=Single, 5=Double, 6=Drop Shadow, 7=Client, 8=Modal, 9=Etched In, 10=Etched Out) |
| <i>className</i> | STREAMFRAME | Identifies the object as an instance of the StreamFrame class (Property discussed in Chapter 5, “Core language.”) |
| <i>form</i> | | Reference to the report that contains the StreamFrame object. (See page 15-101.) |
| <i>height</i> | 0 | Height of the StreamFrame object in its PageTemplate’s Report’s current metric units. (See page 15-107.) |
| <i>left</i> | 0 | The location of the left edge of the StreamFrame object in its PageTemplate’s Report’s current metric units, relative to the PageTemplate’s <i>marginLeft</i> . (See page 15-115.) |
| <i>marginHorizontal</i> | 0 | Horizontal margin inside the StreamFrame |
| <i>marginVertical</i> | 0 | Vertical margin inside the StreamFrame |
| <i>name</i> | | The name of the StreamFrame object. (See page 15-123.) |
| <i>parent</i> | | The PageTemplate object that contains the StreamFrame (Property discussed in Chapter 5, “Core language.”) |
| <i>streamSource</i> | | Reference to a StreamSource object that contains objects to be rendered in the StreamFrame |
| <i>top</i> | 0 | The location of the top edge of the StreamFrame object in its PageTemplate’s Report’s current metric units, relative to the PageTemplate’s <i>marginTop</i> . (See page 15-174.) |
| <i>width</i> | | Width of the StreamFrame object in its PageTemplate’s Report’s current metric units. (See page 15-183.) |

| Event | Parameters | Description |
|------------------|------------|---|
| <i>canRender</i> | | Before the StreamFrame is rendered; return value determines whether StreamFrame is rendered |
| <i>onRender</i> | | After the contents of the StreamFrame have rendered |

Description A StreamFrame object describes a rectangular region inside the margins of a PageTemplate into which data from a StreamSource object is rendered.

Although you may create multiple StreamFrame objects in a PageTemplate, the Report Designer currently does not support multiple StreamFrame objects visually.

See also class PageTemplate, class StreamSource

class StreamSource

Describes a data source for streaming.

Syntax [*<oRef>* =] new StreamSource(*<report>*)

<oRef> A variable or property—typically of *<report>*—in which you want to store a reference to the newly created StreamSource object.

<report> The Report object to which the StreamSource object binds itself.

Properties The following tables list the properties and methods of the StreamSource class. (No events are associated with this class.)

| Property | Default | Description |
|-------------------|--------------|--|
| <i>className</i> | STREAMSOURCE | Identifies the object as an instance of the StreamSource class (Property discussed in Chapter 5, “Core language.”) |
| <i>detailBand</i> | | A Band object that corresponds to the <i>rowset</i> |
| <i>maxRows</i> | | The maximum number of rows the StreamSource will provide per StreamFrame per page |
| <i>name</i> | | The name of the StreamSource object. (See page 15-123.) |
| <i>parent</i> | | The Report object that contains the StreamSource (Property discussed in Chapter 5, “Core language.”) |
| <i>rowset</i> | | The Rowset object that drives the StreamSource |

| Method | Parameters | Description |
|-------------------------|------------|---|
| <i>beginNewFrame()</i> | | Forces the next band to display in a new StreamFrame. |
| <i>release()</i> | | Explicitly releases the StreamSource object from memory. (See page 5-34.) |

Description A StreamSource object acts as the common ground between a rowset that contains data you want to display and a band that contains components to display that data.

Every StreamFrame is assigned a StreamSource. The same StreamSource object may be assigned to multiple StreamFrame objects. The data from a StreamSource is rendered in all the StreamFrame objects that are linked to it. You may limit the number of rows that are rendered per StreamFrame, and therefore per page, by setting the StreamSource object's *maxRows* property.

A StreamSource object may contain Group objects that group data to perform aggregate functions.

See also class Report, class StreamFrame, class Band

agAverage()

Aggregate method that returns the mean average for a group.

Syntax `<oRef>.agAverage(<codeblock>)`

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value to average.

Property of Group

Description Use *agAverage()* to calculate the mean average of the value returned by *<codeblock>* in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the Group object's *parent* StreamSource object's *rowset*.

If *<codeblock>* returns a *null* value, it is not considered in the average.

You may call *agAverage()* at any time. If necessary, the report engine will look ahead to calculate the result.

Example Suppose you're reporting test scores, grouped by age. You display the average in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agAverage()* method:

```
{||this.parent.parent.agAverage({||this.parent.rowset.fields[ "Score" ].value})}
```

`agCount()`

To get to the Group object's *agAverage()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agAverage()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the *StreamSource*.
- The *StreamSource* object contains the *rowset*.

See also *agCount()*, *agMax()*, *agMin()*, *agStandardDeviation()*, *agSum()*, *agVariance()*

agCount()

Aggregate method that returns the number of items in a group.

Syntax `<oRef>.agCount(<codeblock>)`

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to count.

Property of Group

Description Use *agCount()* to count the number of items in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* *StreamSource* object.

If *<codeblock>* returns a *null* value, that item is not counted, so that empty rows will be skipped. To count a row even if it is empty, have the *<codeblock>* return a constant non-*null* value, for example,

```
{||1}
```

You may call *agCount()* at any time. If necessary, the report engine will look ahead to calculate the result.

Example Suppose you're reporting test scores, grouped by age. You display the number of tests scored in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agCount()* method:

```
{||this.parent.parent.agCount({||this.parent.rowset.fields[ "Score" ].value)}}}
```

To get to the Group object's *agCount()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agCount()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agMax()*, *agMin()*, *agStandardDeviation()*, *agSum()*, *agVariance()*

agMax()

Aggregate method that returns the highest value within a group.

Syntax `<oRef>.agMax(<codeblock>)`

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to track.

Property of Group

Description Use *agMax()* to return the highest value returned by *<codeblock>* in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If *<codeblock>* returns a *null* value, it is ignored.

You may call *agMax()* at any time. If necessary, the report engine will look ahead to determine the result.

Example Suppose you're reporting test scores, grouped by age. You display the highest score in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agMax()* method:

```
{||this.parent.parent.agMax({||this.parent.rowset.fields[ "Score" ].value})}}
```

To get to the Group object's *agMax()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agMax()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agCount()*, *agMin()*, *agStandardDeviation()*, *agSum()*, *agVariance()*

agMin()

Aggregate method that returns the lowest value within a group.

Syntax `<oRef>.agMin(<codeblock>)`

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to track.

Property of Group

Description Use *agMin()* to return the lowest value returned by *<codeblock>* in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If *<codeblock>* returns a *null* value, it is ignored.

You may call *agMin()* at any time. If necessary, the report engine will look ahead to determine the result.

Example Suppose you're reporting test scores, grouped by age. You display the lowest score in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agMin()* method:

```
{||this.parent.parent.agMin({||this.parent.rowset.fields[ "Score" ].value})}
```

To get to the Group object's *agMin()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agMin()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agCount()*, *agMax()*, *agStandardDeviation()*, *agSum()*, *agVariance()*

agStandardDeviation()

Aggregate method that returns the standard deviation of the values in a group.

Syntax `<oRef>.agAverage(<codeblock>)`

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to sample.

Property of Group

Description Use *agStandardDeviation()* to calculate the standard deviation of the value returned by **<codeblock>** in the group. **<codeblock>** is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If **<codeblock>** returns a *null* value, it is not considered in the sample.

You may call *agStandardDeviation()* at any time. If necessary, the report engine will look ahead to calculate the result.

Example Suppose you're reporting test scores, grouped by age. You display the standard deviation in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agStandardDeviation()* method:

```
{||this.parent.parent.agStandardDeviation({||this.parent.rowset.fields[ "Score"
].value)}}}
```

To get to the Group object's *agStandardDeviation()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agStandardDeviation()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agCount()*, *agMax()*, *agMin()*, *agSum()*, *agVariance()*

agSum()

Aggregate method that returns the total of a group.

Syntax *<oRef>.agSum(<codeblock>)*

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to total.

Property of Group

Description Use *agSum()* to calculate the total of the value returned by *<codeblock>* in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If *<codeblock>* returns a *null* value, it is ignored.

You may call *agSum()* at any time. If necessary, the report engine will look ahead to calculate the result.

Example Suppose you're tracking overtime hours, grouped by employee. You display the average in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agSum()* method:

```
{||this.parent.parent.agSum({||this.parent.rowset.fields[ "Overtime" ].value))}
```

To get to the Group object's *agSum()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agSum()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agCount()*, *agMax()*, *agMin()*, *agStandardDeviation()*, *agVariance()*

agVariance()

Aggregate method that returns the variance of the values in a group.

Syntax *<oRef>.agVariance(<codeblock>)*

<oRef> The Group object that defines the group you want to summarize.

<codeblock> A codeblock or pointer to a function that returns the value you want to sample.

Property of Group

Description Use *agVariance()* to calculate the variance of the value returned by *<codeblock>* in the group. *<codeblock>* is usually an expression codeblock that returns the *value* property of a field in the *rowset* of the Group object's *parent* StreamSource object.

If *<codeblock>* returns a *null* value, it is not considered in the sample.

You may call *agVariance()* at any time. If necessary, the report engine will look ahead to calculate the result.

Example Suppose you're reporting test scores, grouped by age. You display the variance in an Text component in the group's *footerBand*. The *text* of the Text component is an expression codeblock that calls the *agVariance()* method:

```
{||this.parent.parent.agVariance({||this.parent.rowset.fields[ "Score" ].value)}}}
```

To get to the Group object's *agVariance()* method from a component in the *footerBand*,

- *this* is the component.
- The component's *parent* is the *footerBand*.
- The *footerBand*'s *parent* is the Group.

The expression codeblock that is passed to *agVariance()* also uses dot operators:

- *this* is the Group object that performs the calculation.
- The Group object's *parent* is the StreamSource.
- The StreamSource object contains the *rowset*.

See also *agAverage()*, *agCount()*, *agMax()*, *agMin()*, *agSum()*, *agVariance()*

autoSort

Whether to automatically sort data to match specified groups.

Property of Report

Description For groups to work properly, data must be sorted to match the groups.

If a Report object's *autoSort* property is *true* (the default), then the *sql* property of any query that is accessed by a StreamSource object that has groups will be modified automatically to include an ORDER BY clause that sorts the rowset in the correct order.

For example, if you have two Group objects, the first grouping by the field State and the second by Zip, then even if the query's *sql* property is set as:

```
select * from SALES
```

the rowset will actually be generated internally with the SQL statement:

```
select * from SALES order by STATE, ZIP
```

If *autoSort* is *false*, the rowset is not altered by the report engine. It assumes that the query is correct and contains the necessary fields in the right order.

Therefore, if you use the *indexName* property to set the rowset order, you should set *autoSort* to *false*; otherwise it defeats the purpose of using *indexName*.

See also *groupBy*

beginNewFrame

Specifies whether rendering always starts in a new StreamFrame.

Property of Band

Description Set the *beginNewFrame* property of the StreamSource object's *detailBand* to *true* if you want each row to be rendered in its own StreamFrame. If you have one StreamFrame in each PageTemplate, you will get one row per page. If you have multiple StreamFrames in each PageTemplate, each one will have at most one row of data.

You would create a page of labels by creating a StreamFrame for each label, set all the StreamFrame objects' *streamSource* property to the same StreamSource, and set its *detailBand*'s *beginNewFrame* property to *true*.

Set the *beginNewFrame* property of a group's *headerBand* to *true* if you want each group to start in a new StreamFrame. If you have one StreamFrame per page, that makes each group start on a new page.

If the *beginNewFrame* property of a group's *footerBand* is *true*, then whenever it is rendered, it will start in a new StreamFrame. For example, you could print a summary page for a report by creating a large *footerBand* for the Report object's *reportGroup* and set its *beginNewFrame* property to *true*.

See also *beginNewFrame()*, *headerEveryFrame*

beginNewFrame()

Forces the next band to display in a new StreamFrame.

Property of StreamSource

Description Use *beginNewFrame()* to conditionally force the next band—whether it is a *detailBand*, *headerBand*, or *footerBand*—to display in a new StreamFrame. In effect, it's as if that band's *beginNewFrame* property is temporarily set to *true*.

Example See the example for *renderOffset* for an example using *beginNewFrame()*.

See also *beginNewFrame*, *maxRows*, *renderOffset*

context

Reports the context in which the band is being rendered.

Property of Band

Description Check the band's *context* property to determine the context in which it's being rendered. *context* is a read-only enumerated property that can have the following values:

| Value | Context |
|-------|-----------------------|
| 0 | Normal |
| 1 | For drilldown summary |

For detail bands, the value will always be Normal, since they are never rendered in the drilldown summary. For footer and header bands, the value will change accordingly.

The *context* property is checked primarily during the *canRender* event of components in a header or footer band when the the header or footer is shown in both the summary and details in a drilldown report; you can change the content of the component accordingly.

See also *drillDown*

canRender

Just before the component is rendered; return value determines whether the component is displayed.

Parameters none

Property of All visual components on a report, StreamFrame

Description *canRender* fires for visual components only when they are in a report. It is fired every time the object is rendered. For a component in a detail band, that means for every row in the rowset.

While you can use *canRender* to evaluate some condition and return *false* to prevent the component from being displayed, the more common use of *canRender* is to alter a component's properties conditionally and always return *true*. You can create a calculated field in a report by altering an HTML component's *text* property in its *canRender* event handler.

You can use the *onRender* event to reset the component to its default state afterward, or always choose the desired state in the *canRender* event.

For a StreamFrame object, the *canRender* event fires before it attempts to retrieve data from its *streamSource*. Note that it is the rendering of StreamFrame objects that cause additional pages to be scheduled. If a report has only one stream frame, and its *canRender* returns *false* so that it is not rendered, no more pages will be printed; the report will terminate. You can call *streamSource.beginNewFrame()* to skip the current stream frame, but in that case, its *canRender* event handler must return *true*.

Example Suppose you're printing a balance sheet and you want to highlight all the negative numbers by making them red. The default `colorNormal` property of the Text component is black. Use the following `canRender` event handler to set the `colorNormal` property appropriately just before the component is rendered:

```
function someFigure_canRender()
  this.colorNormal := iif( this.text() < 0, "red", "black" )
  return true
```

Because the `text` property of the Text component is an expression codeblock, to get the value that is going to be displayed, call the component's `text` property. Don't forget to RETURN `true`; otherwise the component is never displayed.

See also `onRender`

detailBand

The Band object in a StreamSource, which displays data from the rowset.

Property of StreamSource

Description A StreamSource object automatically has a Band object assigned to its `detailBand` property. This is the band that is rendered to display data in the rowset.

Visual components for displaying detail rows in the report should be created as a property of the StreamSource object's `detailBand`.

See also `beginNewFrame`, `footerBand`, `headerBand`

drillDown

How the group's bands are displayed in drilldown format.

Property of Group

Description By choosing a drilldown format, the header and footer bands in the specified group are displayed first, followed by the detail rows. The `drillDown` property controls whether the header and footer bands are repeated with the detail rows. `drillDown` is an enumerated property that can have one of the following values:

| Value | Description |
|-------|---|
| 0 | None—do not format as drilldown (default) |
| 1 | Drilldown. Do not repeat headers or footers |
| 2 | Drilldown, repeat headers |
| 3 | Drilldown, repeat footers |
| 4 | Drilldown, repeat headers and footers |

See also *context*

endPage

The last page number to render.

Property of Report

Description When rendering to a window (the default), only one page, the *startPage*, is rendered at a time. When rendering to a printer or file, pages are rendered from the *startPage* to the *endPage*.

By default, *endPage* is -1, which means that all the pages in the report are rendered. Set *endPage* to a number greater than zero to set the last page to render. When and if the report engine gets to that page, it stops after it has finished rendering it.

Example

See also *output, render(), startPage, isLastPage()*

expandable

Specifies whether an object will increase in size automatically to accommodate the objects within it.

Property of Band

Description If a Band object's *expandable* property is *true* (the default), it will increase in size to display all the components inside it, even if its *height* is set to zero.

Set *expandable* to *false* if you want to make the size and number of rows displayed on each page constant, no matter what is displayed.

See also *fixed, height* (page 15-107)

firstOnFrame

Whether the band is being rendered for the first time in the StreamFrame.

Property of Band

Description Check the *firstOnFrame* property in a component's *canRender* event if you want to render it (or don't want to render it) the first time the component's band is being rendered in a StreamFrame only.

Example If you place column headings inside a `StreamFrame`, you can create Text components in the detail band that have the following *canRender* event:

```
{||this.parent.firstOnFrame}
```

See also *beginNewFrame*

firstPageTemplate

The `PageTemplate` object that is used for the report's first page.

Property of Report

Description Because a report may have multiple `PageTemplate` objects, the *firstPageTemplate* property is used to identify the `PageTemplate` that the report should render as its first page.

Once the first `PageTemplate` has been chosen, each `PageTemplate` object has a *nextPageTemplate* property that identifies the page to render next.

See also *nextPageTemplate*

fixed

Specifies whether an object's position in a band is fixed or if it can be "pushed down" or "pulled up" by the rendering or suppression of other objects.

Property of Visual components in a band.

Description Consider two components in a band named *object1* and *object2*. Suppose that

- The bottom of *object1* is at or above the *top* of *object2*.
- *object1*'s *variableHeight* property is *true*.
- *object1* grows in height to accommodate the data in it.
- The bottom of *object1* is now below the *top* of *object2*.

Then if *object2*'s *fixed* property is *false* (the default), *object2* will be pushed down by *object1* so that *object2*'s *top* will be at the bottom of *object1*. This in turn might push down other objects in the band.

object2 can also be pulled up if the bottom of *object1* is at or above the top of *object2* and *object1* is suppressed by its *suppressIfBlank* or *suppressIfDuplicate* properties.

The horizontal position of the objects in question doesn't matter, only the vertical position of their *top* and bottom ends.

If an object's *fixed* property is *true*, it is not moved by the rendering or suppression of other objects.

After the band has been rendered, all the objects return to their original positions and sizes.

See also *expandable, variableHeight, suppressIfBlank, suppressIfDuplicate*

footerBand

The Band object that renders after each group.

Property of Group

Description A Group object automatically has a Band object assigned to its *footerBand* property. This band is rendered after each group is completed; that is, just before the next group starts or at the end of the rowset. It usually contains components that display summary information.

For components in the *footerBand*, the Group object's aggregate functions will return summary values for the group that has just completed.

Example Suppose you're printing a list of students grouped by grade. At the end of each grade, you want to display the number of students in that grade. In the Group object's *footerBand*, you create an Text component with the following expression codeblock assigned to its *text* property:

```
{||"Count: " + this.parent.parent.agCount({||this.parent.rowset.fields["ID"].value))}
```

See also *beginNewFrame, detailBand, headerBand*

groupBy

The name of the field upon which groups are formed.

Property of Group

Description *Visual* dBASE groups rows by monitoring the value of a field in the rowset. The *groupBy* property contains the name of a field as a character string with an optional ascending or descending sort designation (not case-sensitive, ascending is the default). You may abbreviate ascending as "ASC" and descending as "DESC". For example, if you're grouping by the Age field, you could have one of the following strings as the *groupBy* property:

```
Age
Age asc
Age desc
Age ascending
Age descending
```

The ascending and descending options have an effect only if the report's *autoSort* property is set to *true*. In that case, *Visual* dBASE will make sure that the data in the rowset is sorted by the correct field in the correct direction.

No matter what the value of *autoSort* is, the named field must exist in the rowset. *Visual* dBASE looks for that field in the rowset's *fields* array, just as you would. For example,

```
rowset.fields[ "Age" ].value
```

Because *Visual* dBASE uses the rowset's *fields* array, you can group on calculated fields. There are two ways to do this. You can create a calculated field in a SQL SELECT statement, in which case set *autoSort* to *true*; or you can create the calculated field by adding a Field object to the rowset's *fields* array, in which case you must set *autoSort* to *false*, because the named field doesn't exist in the table, so you can't sort on it. You would still have to make sure that the rows are sorted in the correct order so that all the rows in the same group are contiguous in the rowset.

Example Suppose you're tracking sales and want to generate a summary report, grouped by quarter. The data stores the actual date, so you'll need to calculate the quarter.

To calculate the quarter, divide the month of the date by 3 and round up:

| Month | Month number | Divided by 3 | Rounded up |
|----------|--------------|--------------|------------|
| January | 1 | 1/3 | 1 |
| March | 3 | 1 | 1 |
| April | 4 | 1 1/3 | 2 |
| December | 12 | 4 | 4 |

You can add the calculated field in the query's *onOpen* event:

```
function sales1_onOpen()
  local c
  c = new Field()           // Create a new Field object
  c.fieldName := "Quarter"  // Give it a name
  this.rowset.fields.add( c ) // Add it to the fields array
  c.beforeGetValue := {|ceiling( month( this.parent.fields[ "Date" ].value ) / 3 )}
```

The group's *groupBy* property is set to "Quarter". You still need to sort the report by date so that the groups will be in the right order. You can't use *autoSort*, since it will try to sort by a field named "Quarter", and there isn't one. So you use the following SQL SELECT statement in the query's *sql* property:

```
select * from OVERTIME order by OVERTIME."DATE"
```

DATE is an SQL reserved word, so you need to place the field name in quotes and use the table name.

See also *autoSort*

headerBand

The Band object that renders before each group.

Property of Group

Description A Group object automatically has a Band object assigned to its *headerBand* property. This band is rendered before the start of a new group, including the first group in the report; and if the Group object's *headerEveryFrame* property is true, at the beginning of every StreamFrame. It usually contains components that identify the current group or display summary information.

Example Suppose you're tracking sales and want to generate a summary report, grouped by quarter. You've already created a calculated field "Quarter" that contains a number from 1 to 4. To print "1st quarter", "2nd quarter" and so forth, set the *text* property of an Text component to the following expression codeblock (all in one line):

```
{|{"1st","2nd","3rd","4th"}
[this.parent.parent.parent.rowset.fields["Quarter"].value] +
" quarter"}
```

This codeblock uses a literal array that contains the corresponding text for the quarter number. To get the quarter number in the calculated field "Quarter" from the Text component:

- *this* is the component
- the component's *parent* is the *headerBand*
- the *headerBand*'s *parent* is the Group object
- the group's *parent* is the StreamSource object
- from the StreamSource object, you can access its *rowset*

See also *beginNewFrame*, *detailBand*, *footerBand*, *headerEveryFrame*

headerEveryFrame

Specifies whether to repeat a group's *headerBand* when a group spans more than one StreamFrame.

Property of Group

Description A group's *headerBand* is rendered at the beginning of the group. By default, *headerEveryFrame* is *false*; that means that if the contents of the group go into another frame, the *headerBand* is not repeated.

Set *headerEveryFrame* to *true* if you want a group's *headerBand* to be rendered at the top of every StreamFrame. For example, if you have one StreamFrame per page, setting *headerEveryFrame* to *true* will print the *headerBand* at the top of each

`isLastPage()`

page, underneath the fixed components (for example column headings) on the page.

If you have nested groups with *headerEveryFrame* set to *true* for each *headerBand*, the header bands will appear in group order at the top of every *StreamFrame*.

The *headerBand*'s *beginNewFrame* property determines whether the header band for a new group should start in a new *StreamFrame*. In contrast, *headerEveryFrame* determines whether the header band should be repeated in subsequent *StreamFrame* objects.

See also *beginNewFrame*, *headerBand*

isLastPage()

Returns *true* or *false* to let you know if additional pages are due to be rendered.

Syntax `<oRef>.isLastPage()`

<oRef> An object reference to the report you want to check.

Property of Report

Description *isLastPage()* returns *true* if the current page is the last page of the report, and *false* if additional pages are to be rendered.

Its main purpose is to allow you to make informed decisions about whether or not to display a button to display additional pages. You may also use *isLastPage()* to display something on the last page of a report.

Visual dBASE does not determine in advance how many pages a report will take. It renders the report one page at a time by filling all the *StreamFrame* objects on that page with data drawn from the *StreamFrame* objects' *streamSource*. If there is more data to render and all the *StreamFrame* objects in the page are full, another page is scheduled.

If the page being rendered is before the report's *startPage*, the rendering is not output. When rendering to a window (the default), rendering stops once a page is output; the window only displays one page at a time. Rendering to a printer or file renders multiple pages. After the page has finished rendering, if another page is scheduled, it is rendered. The process repeats until all the pages are rendered, or the report's *endPage* page is rendered. In that case, the rendering process stops, even though another page may be scheduled.

isLastPage() ignores the *endPage* setting and determines if another page is scheduled to be rendered. It can be reliably called only after the last *StreamFrame* on a *PageTemplate* has been rendered, since it is the rendering of *StreamFrame* objects that determines the scheduling of new pages.

isLastPage() is usually called from the *canRender* event handler for a component attached to the *PageTemplate*—not in a band—that is defined after all the *StreamFrame* objects. The order in which objects are created and assigned in the

report class constructor directly determines their order of definition and rendering.

Example The following is the *canRender* event handler for a next page button on the PageTemplate. It hides the button on the last page of the report.

```
{|not this.form.isLastPage()}
```

See also *canRender*, *endPage*, *startPage*

leading

The distance between consecutive lines inside a component.

Property of Text

Description *leading* controls the line spacing within an Text component. By default, it's zero, which uses the default line space for the font.

You can set *leading* to a non-zero value to set the baseline-to-baseline distance of the text in the component.

See also *alignVertical*, *tracking*, *verticalJustifyLimit*

marginBottom

The space between the bottom of the page and the usable area of the PageTemplate.

Property of PageTemplate

Description Use *marginBottom* in conjunction with the PageTemplate's other *margin*-properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginBottom indicates the distance, in the report's current *metric* units, between the bottom of the page and the bottom of the usable area.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

See also *marginLeft*, *marginRight*, *marginTop*

marginHorizontal

The horizontal margin between the edge of the object and its contents.

Property of Editor, Text, StreamFrame

Description An object's horizontal margin is the same on both the left and right sides. In a Text component, the text is indented inside its rectangular frame. The *left* position of all bands inside a StreamFrame object are relative to the horizontal margin on the left and restricted by the horizontal margin on the right.

marginHorizontal is measured in the form or report's current *metric* units.

See also *marginVertical*

marginLeft

The space between the left edge of the page and the usable area of the PageTemplate.

Property of PageTemplate

Description Use *marginLeft* in conjunction with the PageTemplate's other *margin-* properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginLeft indicates the distance, in the report's current *metric* units, between the left edge of the page and the left edge of the usable area.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

See also *marginBottom*, *marginRight*, *marginTop*

marginRight

The space between the right edge of the page and the usable area of the PageTemplate.

Property of PageTemplate

Description Use *marginRight* in conjunction with the PageTemplate's other *margin-* properties to define the usable area of the page. The position of StreamFrame

objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginRight indicates the distance, in the report's current *metric* units, between the right edge of the page and the right edge of the usable area.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

See also *marginBottom*, *marginLeft*, *marginTop*

marginTop

The space between the top of the page and the usable area of the PageTemplate.

Property of PageTemplate

Description Use *marginTop* in conjunction with the PageTemplate's other *margin*- properties to define the usable area of the page. The position of StreamFrame objects and components bound to the PageTemplate is relative to the top left corner of the usable area and cannot extend beyond the bottom right corner.

marginTop indicates the distance, in the report's current *metric* units, between the top of the page and the top of the usable area.

When using multiple PageTemplate objects, you can position items differently just by changing the margins. For example, you may use different left and right pages that have increased margins on the inside edge (the gutter space) for binding.

See also *marginBottom*, *marginLeft*, *marginRight*

marginVertical

The vertical margin between the edge of the object and its contents.

Property of Editor, Text, StreamFrame

Description An object's vertical margin is the same on both the top and bottom. All rendering in the object, whether it's text in a Text component or bands inside a StreamFrame object, is relative to the vertical margin on the top and restricted by the vertical margin on the bottom.

marginVertical is measured in the form or report's current *metric* units.

See also *marginHorizontal*

maxRows

The maximum number of rows a `StreamSource` will provide to a `StreamFrame` on each page.

Property of `StreamSource`

Description Use *maxRows* to limit the number of rows displayed in each `StreamFrame`. For example, on a page with a single `StreamFrame`, if *maxRows* is 10, only 10 rows maximum will be displayed in the `StreamFrame` on each page.

See also *beginNewFrame*, *beginNewFrame()*

nextPageTemplate

The `PageTemplate` object that is used for the following page.

Property of `PageTemplate`

Description Because a report may have multiple `PageTemplate` objects, the *firstPageTemplate* property is used to identify the `PageTemplate` that the report should render as its first page.

Once the first `PageTemplate` has been chosen, each `PageTemplate` object has a *nextPageTemplate* property that identifies the page to render next.

For a report that uses the same page over and over, the same `PageTemplate` object is used for both the report's *firstPageTemplate* property and that `PageTemplate`'s own *nextPageTemplate* property.

You can create a different introduction or cover page for a report by specifying the cover page as the report's *firstPageTemplate* property, and then set the cover page's *nextPageTemplate* property to the `PageTemplate` for the body pages.

To alternate left and right pages, set the *nextPageTemplate* of the left page to the right page, and vice versa. Then specify the page on the right as the report's *firstPageTemplate*.

See also *firstPageTemplate*

onPage

After the page has finished rendering.

Parameters none

Property of `Report`

Description *onPage* fires after each page has finished rendering, including the last page. By that time, it's too late to do anything to the page, but you can take actions for the next page.

In an *onPage* event handler, the report's *reportPage* property indicates the page that has just finished rendering.

Example Suppose you're going to print a report, make two-sided copies, and bind them. You want to shift the margins slightly on both pages to accommodate the binding. The right pages need to move to the right and the left pages need to move to the left. Other than that, the pages are identical.

You can use the *onPage* event handler to shift the margins of the PageTemplate after each page has printed, in preparation for the next page:

```
function Report_onPage()
  #define TWIPS(x) ((x)*1440)
  if this.reportPage % 2 == 0    // Finished left page, start right
    this.pageTemplate1.marginLeft  = TWIPS( 1.0 )
    this.pageTemplate1.marginRight = TWIPS( 0.5 )
  else                          // Finished right page, start left
    this.pageTemplate1.marginLeft  = TWIPS( 0.5 )
    this.pageTemplate1.marginRight = TWIPS( 1.0 )
  endif
```

Pages on the left are even-numbered. An even number modulo 2 yields zero, so if a left page has just been printed, the margins are set to those for a right page, and vice versa.

onRender

After the component is rendered.

Parameters none

Property of All visual components in a report, Band, StreamFrame

Description *onRender* fires for visual components only when they are in a report. It is fired every time the object is rendered, after it has finished rendering. For a component in a detail band, that means for every row in the rowset.

You can use the *onRender* event to reset the component to its default state after changing it in its *canRender* event.

onRender also fires for the Band objects, after all the objects in that band have been rendered (for a detail band, after every row); and StreamFrame objects, after that StreamFrame has been rendered, on each page.

Example Suppose you're printing a list of test scores, grouped by age. You have a *headerBand* that prints in every StreamFrame. After printing in the first StreamFrame, you want it to add the word "continued". Every time a new

group starts, you want to remove the word. There is also a *footerBand* to print totals for the group.

You create an Text object with the following expression codeblock as its *text* property:

```
{|"Age: " + this.parent.parent.parent.rowset.fields["Age"].value}
```

In the component's *onRender* event, you change the codeblock to include the word "continued":

```
function header1_html1_onRender()
  this.text = {|"Age: " + ;
               this.parent.parent.parent.rowset.fields["Age"].value + " continued"}
```

So now, once it is rendered at the beginning of the group, it is changed so that it will contain the word "continued" for the rest of the group.

To change it back for the start of a new group, use the following *onRender* event handler for the Text component in the *footerBand*:

```
function footer1_html1_onRender()
  this.text = {|"Age: " + this.parent.parent.parent.rowset.fields["Age"].value}
```

This restores the original codeblock at the end of the group, preparing the Text component for the beginning of the next group.

See also *canRender*

output

Designates the target medium for the report.

Property of Report

Description Set the report's *output* property to designate how you want the report to be rendered. *output* may contain one of the following values:

| Value | Target |
|-------|--------------|
| 0 | Window |
| 1 | Printer |
| 2 | Printer file |
| 3 | Default |
| 4 | HTML file |

The Default output is to a window. When rendering to a window, only one page at a time is rendered.

If you designate either Printer file or HTML file, the report's *outputFilename* property must be set to the name of the target file.

See also *outputFilename, printer*

outputFilename

The name of the target output file.

Property of Report

Description If a report's *output* property is set to either Printer file or HTML file, the *outputFilename* property must be set to the target file.

The file will contain the output from the report. If the file already exists, it will be overwritten.

See also *output*

preRender

Before the band begins rendering.

Parameters none

Property of Band

Description A band's *preRender* event fires before the band starts rendering. Note that unlike *canRender*, you cannot prevent the band from rendering in the *preRender* event handler.

See also *canRender, onRender*

printer

An object that describes various printer output options.

Property of Report

Description A *printer* object contains properties for the following printer options:

| Property | Default | Description |
|----------------------|--|--|
| <i>color</i> | Monochrome | Whether the output should be in color/grayscale or plain monochrome (0=Default, 1=Monochrome, 2=Color) |
| <i>copies</i> | 1 | Number of copies |
| <i>duplex</i> | None | Whether to print in duplex, and in which orientation (0=Default, 1=None, 2=Vertical, 3=Horizontal) |
| <i>orientation</i> | Portrait | The orientation of the output (0=Default, 1=Portrait, 2=Landscape) |
| <i>paperSize</i> | printer-dependent | The size of the paper to use |
| <i>paperSource</i> | printer-dependent | The paper tray or bin to use |
| <i>printerName</i> | | The name of the target printer (blank for default printer) |
| <i>printerSource</i> | Visual dBASE default | Which <i>printerName</i> to use (0=Visual dBASE default, 1=Windows default, 2=Specific) |
| <i>resolution</i> | High | Graphics resolution (0=Default, 1=Draft, 2=Low, 3=Medium, 4=High) |
| <i>trueTypeFonts</i> | Depends on the currently specified default printer | How to handle TrueType fonts (0=Default, 1=Bitmap, 2=Download, 3=Substitute, 4=Outline) |

and one method:

| Method | Parameters | Description |
|------------------------|------------|---|
| <i>choosePrinter()</i> | | Displays Print Setup dialog to allow user to choose a printer and set printing options. |

Each Report object has its own *printer* object which controls the report output. The global *_app* object also has a *printer* object that represents the default printer.

If a report's *printer* object's *printerName* property is blank, the printer specified in *_app.printer.printerName* is used (all other properties in the report's *printer* object are applied). If *_app.printer.printerName* is blank, the default Windows printer is used.

You may call the *choosePrinter()* method to allow the user to choose a printer and set printing options. Each *printer* object's *choosePrinter()* method changes

the printer settings for that object. Calling `_app.printer.choosePrinter()` sets the default printer; the same as the `CHOOSEPRINTER()` function.

See also *output*

render()

Renders the reports to the designated target.

Syntax `<oRef>.render()`

<oRef> An object reference to the report you want to render.

Property of Report

Description Call a Report object's `render()` method to generate the report. The output of the report goes to the target designated by the report's *output* property.

The report engine renders the report internally and outputs the results starting with the page designated by the *startPage* property. When rendering to a window (the default), rendering stops once a page is output; the window only displays one page at a time. Rendering to a printer or file renders multiple pages. It continues to render all the objects on each page until it exhausts all *StreamSource* objects, or it has finished rendering the page designated by the *endPage* property.

Example The following is the *onClick* event handler for button on the report that displays the next page. It increments the *startPage* and re-renders the report.

```
{; this.form.startPage++; this.form.render();}
```

See also *endPage, output, startPage*

renderOffset

The offset of the bottom of the band from the top of the current stream frame.

Property of Band

Description *renderOffset* is a read-only property that contains the position of the bottom of the band that was just rendered, measured from the top of the stream frame, in the report's current *metric* units. It is updated just before the band's *onRender* event fires.

A common use of *renderOffset* is to prevent a band from being split between two pages when using *expandable* bands—that is, bands that can vary in height. The *renderOffset* is checked the *onRender* event. If the offset is too close to the bottom, less than a predetermined minimum size, the stream source's

beginNewFrame() method is called to force the output to the next stream frame. For fixed height bands, the stream source's *maxRows* property would be used instead.

Example The following *onRender* event handler for the *detailBand* of the stream source prevents a band from being split between two pages by checking the *renderOffset*.

```
function DETAILBAND_onRender
  if this.renderOffset > this.streamFrame.height - 1000
    this.parent.beginNewFrame()
  endif
```

A minimum height of 1000 twips is used in this example. The band references the current *StreamFrame* through the *streamFrame* property. The band's *parent* is the *StreamSource*, which has the *beginNewFrame()* method.

See also *beginNewFrame*, *beginNewFrame()*, *expandable*, *maxRows*, *streamFrame*

reportGroup

A Group object for the report as a whole.

Property of Report

Description A Report object automatically has a Group object assigned to its *reportGroup* property. The *groupBy* property of this Group object is an empty string. Use the *reportGroup* to calculate aggregates for the entire report, for example, a grand total; and for items in a report introduction or summary.

A *reportGroup* has a *headerBand*, a *footerBand*, and aggregate methods, just like any other Group object. Because the *parent* of the *reportGroup* is the Report object instead of a *StreamSource* or Group object, the object reference path to data is slightly different for components in the *reportGroup*.

reportPage

The current page number being rendered.

Property of Report

Description The *reportPage* property contains the number of the page that is being rendered. During an *onPage* event, the *reportPage* is the page that has just finished rendering.

Example To display the current page number on a report, create an Text component on the PageTemplate that contains the following expression codeblock as its *text* property:

```
{|"Page " + this.parent.parent.reportPage}
```

See also *isLastPage()*, *onPage*

reportViewer

A reference to the ReportViewer object that instantiated the report.

Property of Report

Description Use the *reportViewer* property to reference the ReportViewer object that instantiated the report. You may access the form that contains the ReportViewer through the ReportViewer object's *form* property.

If the report was not instantiated through a ReportViewer object, *reportViewer* is *null*.

rotate

The orientation of an object, in 90-degree increments.

Property of Text

Description To rotate the text inside an Text component, set its *rotate* property to one of the following values:

| Value | Clockwise rotation |
|-------|--------------------|
| 0 | none |
| 1 | 90 degrees |
| 2 | 180 degrees |
| 3 | 270 degrees |

startPage

The first page number to output.

Property of Report

Description When rendering to a window (the default), only one page, the *startPage*, is rendered at a time. When rendering to a printer or file, pages are rendered from the *startPage* to the *endPage*.

By default, *startPage* is 1, which means that all the pages that are rendered are output. Set *startPage* to a number greater than 1 to delay the beginning of the output.

The report engine must still render each page until it gets to the *startPage* because it dynamically paginates the report. The position of a row in a report may change whenever someone changes the table, so use caution when using *startPage* to display segments of a report.

Example

See also *endPage*, *output*, *render()*

streamFrame

The StreamFrame object in which the band is currently rendering.

Property of Band

Description The read-only *streamFrame* property contains a reference to the StreamFrame object in which the band is currently rendering. In addition to being a direct reference to that object, it is particularly useful when you have more than one stream frame, and want to take different actions dependent on which stream frame is being rendered.

streamSource

The StreamSource object that contains objects to render in the StreamFrame.

Property of StreamFrame

Description A StreamFrame object's *streamSource* property identifies the StreamSource object that supplies the StreamFrame object's data stream.

If multiple StreamFrame objects have the same *streamSource* property, that StreamSource will stream data to those StreamFrame objects in series, in the order in which the StreamFrame objects were created (the same order as they are listed in the class definition in the .REP file).

If multiple StreamFrame objects have different *streamSource* properties, each StreamFrame will be filled from its own StreamSource object (in the same order as above) until all the StreamFrame objects in the page are filled. If a particular StreamFrame object's StreamSource is exhausted, it is no longer filled. When all StreamSource objects are exhausted, all the objects on that last PageTemplate are rendered, and the report is done.

suppressIfBlank

Specifies whether an object is suppressed, or not rendered, if it is blank.

Property of Text

Description *suppressIfBlank* has an effect only for visual components that are in a report. If *true*, it suppresses the rendering of the component if its display value is blank.

For example, suppose you're printing two-line addresses with the city underneath. If the second address line is blank, you don't want it to occupy any space. By setting that component's *suppressIfBlank* property to *true*, if it's blank, nothing gets rendered and all the components below it that have their *fixed* properties set to *false* are moved up to fill the space.

Components are rendered in their z-order, the order in which they are defined in the report. *suppressIfBlank* does not affect the position of components that have already been rendered. The z-order of the components in the report should match their top-to-bottom order; otherwise *suppressIfBlank* will have no effect.

By default *suppressIfBlank* is *false*. You can also use the component's *canRender* event to suppress rendering.

See also *canRender*, *suppressIfDuplicate*

suppressIfDuplicate

Specifies whether an object is suppressed, or not rendered, if its value is the same as the previous time it was rendered.

Property of Text

Description *suppressIfDuplicate* has an effect only for visual components that are in a report. If *true*, it suppresses the rendering of the component if its display value is the same as the previous time it was rendered in the same report, even if the previous time was in another group in the report.

Use *suppressIfDuplicate* to eliminate duplicating the same data value over and over again for multiple rows in a report. For example, you may have information sorted by date. With *suppressIfDuplicate* set to *true*, each date will be rendered only once.

By default *suppressIfDuplicate* is *false*. You can also use the component's *canRender* event to suppress rendering.

See also *canRender*, *suppressIfDuplicate*

title

The title of a report.

Property of Report

Description The *title* property contains the title of the report. It is displayed in the title bar of the report preview window.

tracking

The amount of extra space between characters.

Property of Text

Description *tracking* adds extra space between characters within an Text component. By default, it's zero, which means no extra spacing.

You can set *tracking* to a non-zero value to add extra space between characters.

See also *alignHorizontal*, *leading*, *trackJustifyThreshold*

trackJustifyThreshold

The maximum amount of added space allowed between words in a fully justified line. Exceeding that amount switches to character tracking.

Property of Text

Description *trackJustifyThreshold* sets a threshold for the amount of extra space between words that can be added to try to justify the line. If a line requires more than the threshold amount, the line is justified by adding space between each character in the line, in addition to the maximum space between each word.

If a line contains only one word and *trackJustifyThreshold* is non-zero, the word will be fully justified with character tracking, unless it is on the last line of text. The last line of text is never justified.

An Text component's *alignHorizontal* property must be set to Justify in order for *trackJustifyThreshold* to have any effect.

See also *alignHorizontal*, *tracking*

variableHeight

Whether an object's *height* can increase automatically to accommodate its contents.

Property of Text

Description Set *variableHeight* to *true* so that an object can grow to accommodate its contents. If an object's *height* is not large enough to display everything, it is increased. *variableHeight* does not shrink objects to fit their contents.

If the object is in a Band object in a report and it grows, it might push down other objects in the band if those objects have their *fixed* property set to *false*.

By default *variableHeight* is *false*.

See also *fixed*

verticalJustifyLimit

The maximum amount of added space allowed between lines in a vertically justified object. Exceeding that amount makes the text top-aligned instead.

Property of Text

Description *verticalJustifyLimit* sets the maximum amount of extra space between lines that can be added to try to vertically justify the lines in an object. If the maximum amount does not justify the lines, *Visual* dBASE gives up and makes the text top-aligned instead.

An Text component's *alignVertical* property must be set to Justify in order for *verticalJustifyLimit* to have any effect.

See also *alignVertical*, *leading*

Text streaming

This chapter describes *Visual* dBASE commands that control text streaming to the Command window, a file, or a printer.

?

Outputs the results or return values of one or more expressions to a new line in the Command window results pane. You can also simultaneously stream output to a print buffer and/or text file.

Syntax

?
[<exp 1>
 [PICTURE <format expC>
 [FUNCTION <function expC>
 [AT <column expN>
 [STYLE [<fontstyle expN>] or [<fontstyle expC>]]
[,<exp 2>...]

<exp 1>[,<exp 2> ...] Expression(s) of any data type. Output consists of expression results or return values.

PICTURE <format expC> Formats <exp 1>, or a specified portion of it, with the picture template <format expC>, a character expression consisting of one of the following:

- Template characters.
- Function symbols preceded by @. (You can also use the FUNCTION option, discussed below.)
- Literal characters.
- A combination of template characters, function symbols, and literal characters.

?

- A variable containing the character expression.

You can use all the template character and function symbols except A, M, R, and S.

FUNCTION <function expC> Formats all characters in <exp 1> with the function template <function expC>, which must contain one or more function symbols. When you specify function symbols with the FUNCTION option, you don't have to precede them with @.

AT <column expN> Specifies a character position at which to start the line. The <column expN> argument, effectively a temporary indent, must be between 0 and 255. Note: The AT parameter is ignored if *_wrap* is *true*.

STYLE [] or [<style expC>] Specifies a font number or style for printed output only. Does not apply to file or Command window output (files always use the default printer font, typically Courier, and Command window font style is controlled through Command window properties). See description below for print STYLE specification details.

Description Use ? to output the results or return values of one or more expressions to a new line in the Command window results pane. You can also simultaneously stream output to a print buffer and/or text file.

Output can be streamed to any or all of these targets at the same time, and streaming to a print buffer or file can be switched on or off any time without affecting the stream to other targets.

You can also stream output using the ?? command. The difference is that ?? appends output to the current line and ? always outputs to a new line.

To use either command, type ? or ?? in the input pane of the Command window (or issue it from a program), follow it with your expression(s) and any optional parameters, then press *Enter*. The results or return values are immediately streamed to the results pane of the Command window as well as to a print buffer and/or file, if either of those streams is turned on.

To erase the contents of the results pane any time, use the CLEAR command in the input pane. Note that this command only clears the results pane of the Command window and does not affect output to a print buffer or file, if either stream is turned on.

To clear all or a portion of the input pane, select the portion you want to clear and press *Del*.

Streaming output to a text file

To stream output to a file, first specify a target file with the command SET ALTERNATE TO, e.g.,

```
set alternate to "c:\output.log"
```

A filename is required (no default is supplied).

Then turn on the file output stream with the command SET ALTERNATE ON.

If you specify a file name without an extension, .TXT is appended to the name. If you don't specify a path, the current path (as shown in the Navigator "Look In" path selector) is used.

You can pause streaming to a file with the command SET ALTERNATE OFF (and resume it again with SET ALTERNATE ON), but to stop streaming and close the file, you must either use CLOSE ALTERNATE or switch output files by issuing another SET ALTERNATE TO command with a different filename or with no filename parameter.

Only one file at a time can be open for text output streaming, and the contents of that open file cannot be viewed until the file is closed using one of the methods above.

If the file you specify in a SET ALTERNATE TO command already exists, you're given the option of overwriting the existing file. If you choose No, text the named file is not opened for text streaming. In addition, if another file was already open for output, streaming to that file stops and the file is closed. To reopen the closed file, you must reissue SET ALTERNATE TO with the filename; to resume output (append to the previously closed file at the point output was cut off), use ADDITIVE as described below, then reissue the SET ALTERNATE ON command.

If you do choose Yes and overwrite the existing file, the file is immediately emptied. Text streaming will not start again, however, until you issue the SET ALTERNATE ON command.

To append to an existing file, use ADDITIVE with SET ALTERNATE TO:

```
set alternate to "c:\output.log" additive
```

The overwrite warning is not issued when ADDITIVE is used.

Tip Since ? begins with a line feed to create each new output line, the first line in a new output file or print buffer will be blank. If you want the first line to contain text output instead, use the ?? command for the first item (rather than the ? command).

```
?? "first item"
? "second item"
```

Streaming output to a printer

To stream output to a print buffer, use SET PRINTER ON. To print the contents of the print buffer to your current printer, use SET PRINTER TO. To print the buffer to a different printer, specify a port or network printer, e.g.,

```
set printer to lpt1
or
set printer to \\server9\printer4
```

The print buffer continues to receive output until you print the contents of the buffer or issue SET PRINTER OFF.

To format printed output in a particular font, size and style, you can either specify a font in "Font,Size,Style" format or use GETFONT() to choose a style.

?

Either way, the font definition is applied to the ? command's STYLE parameter to format the associated line, word or block. This example shows three ways to set the font:

```
// turn on the print buffer
set printer on
// assign a font definition to the variable headingStyle by using getfont()
headingStyle = getfont()
? "Program Log" style headingStyle
// you can also specify a font and apply it without using getfont()
bodyStyle = "Arial,9,Swiss"
// font formatting for the date() item in this block
// is specified directly with the STYLE parameter
? "Date: " style "Arial,9,BI,Swiss", date() style bodyStyle
// print the two lines to the default printer
set printer to
```

You can get definitions for any font on your system by using GETFONT() to open a Font dialog, selecting your font options, and examining the result:

```
s = getfont()
? s
// selecting 16-pt Arial bold in the Font dialog returns "Arial,16,B,Swiss"
```

To apply a print style to the default font, you can use codes as shown below in place of a font specification:

```
//make default text:
? "bold" style "B"
? "italic" style "I"
? "strikeout" style "S"
? "underlined" style "U"
? "superscript" style "R"
? "subscript" style "L"
? "or use any combination, such as bold italic" style "BI"
```

To overstrike a line of text from a program file in printed output, use the AT option with `_wrap` set to *false*. To overwrite rather than overstrike text in printed output, use the AT option with `_wrap` set to *true*, which causes only the second line to print.

To override both an overall `_alignment` setting and individual paragraph alignments in a memo field, use the B, I, or J functions.

Tip If SET SPACE is ON (startup default; to test, use ? SET("SPACE")), a space is inserted between multiple expressions streamed out to the same line and between expressions appended to the current line with the ?? command. To remove the spaces for literal formatting, use SET SPACE OFF.

Example This example uses ? and ?? to display a first and last name in various formats:

```
Firstname="Sally  "
Lastname ="Stephens "
? Firstname,Lastname
// simple display, no formatting or positioning
// Sally      Stephens
```



```

? Firstname picture "@T"
?? " "
?? Lastname picture "@!"
// trim Firstname, make lastname uppercase
// Sally STEPHENS

? Lastname STYLE "B"
?? Firstname AT 20
// display in fixed columns.
// Lastname will print in boldface
// Stephens Sally

```

This example formats -3273.68 four different ways:

```

n=-3273.68
? n picture "9,999,999.99" // insert commas
//      -3,273.68
? n picture "9,999,999"    // no decimals
//      -3,274
? n picture "@L 9,999,999" // zero fill
// -0003,274
? n picture "@(BT"         // use () for negative number, left-align, and trim
// (3273.68)

```

See Also ??, `_alignment`, `_wrap`, `DISPLAY`, `GETFONT()`, `LIST`, `PRINTJOB`, `SET MEMOWIDTH`, `SET ALTERNATE`, `SET PRINTER`, `SET SPACE`, `TEXT`

??

Appends the value of one or more expressions to the current line in the Command window, print buffer or a file.

Syntax ??
 [<exp 1>
 [`PICTURE` <format expC>]
 [`FUNCTION` <function expC>]
 [`AT` <column expN>]
 [`STYLE` [<fontstyle expN>] [<fontstyle expC>]]
 [,<exp 2>...]

Description The ?? command is identical to the ? command, except it appends output to the end of the current line rather than to a new line.

Example See ? command

See Also ?

CHOOSEPRINTER()

Opens a printer setup dialog box. Returns *false* if you cancel out of the dialog, *true* otherwise.

Syntax CHOOSEPRINTER([<title expC>])

<title expC> Optional custom title for the printer setup dialog box.

Description Use CHOOSEPRINTER() to open a printer setup dialog box, which lets you change the current printer or printer options.

```
p=chooseprinter()
? p
// opens the printer setup dialog; p = false only if the dialog is cancelled
chooseprinter ("Tip: For 2-sided printing, see Options, Paper/Output options")
// opens the printer setup dialog with a printing tip in the title
```

If you use CHOOSEPRINTER() to switch printers, SET PRINTER TO, _pdriver, _plength, and _porientation will automatically point to the new printer.

To activate a specific printer driver, you can also use _pdriver.

Menu equivalent: File | Print opens the Print dialog, which offers a printer selection list and properties dialog for the selected printer.

See Also _pdriver, CLOSE PRINTER, SET DEVICE, SET PRINTER

CLEAR

Clears the Command window results pane.

Syntax CLEAR

Description Use CLEAR to remove the contents of the results pane of the Command window.

CLOSE ALTERNATE

Close the text stream file opened with SET ALTERNATE

Syntax CLOSE ALTERNATE

Description CLOSE ALTERNATE is equivalent to issuing SET ALTERNATE TO with no filename. See SET ALTERNATE for details.

See also CLOSE ALL, SET ALTERNATE

CLOSE PRINTER

Close the print buffer, sending buffered output to the printer.

Syntax CLOSE PRINTER

Description CLOSE PRINTER is equivalent to issuing SET PRINTER TO with no options. See SET PRINTER for details.

See also CLOSE ALL, SET PRINTER

EJECT

Advances printer paper to the top of the next page.

Syntax EJECT

Description Use EJECT to position printed output on the page. If you are using a tractor-feed printer (such as a dot matrix printer) and the paper is correctly positioned, EJECT advances the paper to the top of the next sheet. If you are using a single-sheet printer (such as a laser printer), EJECT prints any data in the print queue and ejects the page. Before printing or executing EJECT, connect and turn on the printer.

EJECT works in conjunction with `_padvance`, `_plength`, and `_plineno`. If `_padvance` is set to "FORMFEED" (the default), issuing the EJECT command from dBASE is equivalent to using your printer's formfeed button or sending the formfeed character (ASCII 12) to the printer. If `_padvance` is set to "LINEFEEDS", issuing EJECT sends individual linefeeds to the printer until `_plineno` equals `_plength`, then resets `_plineno` to 0. Then, `_pageno` is incremented by 1. For more information, see `_padvance`.

EJECT is often used in when printing reports. For example, if `PROW()` returns a value that is close to the bottom of the page, issue EJECT to continue the report at the top of the next page. EJECT automatically resets the printhead to the top left corner of the new page, which is where `PROW() = 0` and `PCOL() = 0`.

EJECT is the same as EJECT PAGE, except EJECT PAGE also executes any page-handling routine you've defined with ON PAGE.

Example In this example, one line is written to the printer and EJECT is then used to ensure that further commands are written on the next page:

```
set printer on
? "This page intentionally left blank"
eject
```

See Also `_padvance`, `_plength`, `_plineno`, EJECT PAGE, ON PAGE, `PCOL()`, `PROW()`, SET PRINTER, SET PROW

EJECT PAGE

Advances printer paper to the top of the next page and executes any ON PAGE command.

Syntax EJECT PAGE

Description Use EJECT PAGE with ON PAGE to control the ejection of pages by a printer. If you define a page-handling routine with ON PAGE AT LINE *<expN>* and then issue EJECT PAGE, dBASE checks to see if the current line number (*_plineno*) is greater than the line number specified by *<expN>*. If *_plineno* is less than the ON PAGE line, EJECT PAGE sends sufficient linefeeds to trigger the ON PAGE page-handling routine.

If *_plineno* is greater than the ON PAGE line, or if you don't have an ON PAGE page-handling routine, EJECT PAGE advances the output as follows:

- If *_padvance* is set to "FORMFEED" and SET PRINTER is ON, dBASE issues a formfeed (ASCII code 12).
- If *_padvance* is set to "LINEFEEDS" and SET PRINTER is ON, dBASE issues sufficient linefeeds (ASCII code 10) to advance to the next page. It uses the formula *_plength - _plineno* to calculate the number of linefeeds.
- If you direct output to a destination other than the printer (for example, if you use SET ALTERNATE or SET DEVICE), dBASE uses the formula *_plength - _plineno* to calculate the number of linefeeds.

After ejecting a page, EJECT PAGE increments *_pageno* by 1 and resets *_plineno* to 0.

Example See ON PAGE

See Also ?, ??, *_padvance*, *_pageno*, *_plength*, *_plineno*, EJECT, ON PAGE, PRINTJOB...ENDPRINTJOB, SET ALTERNATE, SET DEVICE, SET PRINTER, SET PROW

ON PAGE

Executes a specified command when printed output reaches a specified line on the current page.

Syntax ON PAGE
[AT LINE *<expN>* *<command>*]

AT LINE *<expN>* Identifies the line number at which to execute the specified page-formatting command.

<command> The command to execute when printed output reaches the specified line number, *<expN>*. To execute more than one command, issue ON

PAGE DO <filename>, where <filename> is a program or procedure file containing the sequence of commands to execute.

Description Use ON PAGE to specify a command to execute when printed output reaches a specific line number. ON PAGE with no options disables any previous ON PAGE statement.

The value of the _plineno system variable indicates the number of lines that have been printed on the current page. As soon as the _plineno value is equal to the value you specify for <expN>, dBASE executes the ON PAGE command.

Use the ON PAGE command to print *headers* and *footers*. For example, the on page command can call a procedure when the _plineno system memory variable reaches the line number that signifies the end of a page. In turn, that procedure can call two procedures, one to print the footer on the current page and one to print the header on the next page.

You can begin header routines with EJECT PAGE to ensure that the header text prints at the top of the following page. EJECT PAGE also sets the _plineno system memory variable to 0. Use the ? command at the beginning of a header procedure to skip several lines before printing the header information. You can also use the ? command at the end of the procedure to skip several lines before printing the text for the page.

Begin footer routines with the ? command to move several lines below the last line of text. You can use the ?? command with the _pageno system memory variable to print a page number for each page on the same line as the footer.

To calculate the appropriate footer position, add the number of lines for the bottom margin and the number of lines for the footer text to get the total lines for the bottom of the page. Subtract this total from the total number of lines per page. Use this result to specify a number for the AT LINE argument. If the footer text exceeds the number of lines per page, the remainder prints on the next page.

Example This example uses EJECT PAGE in conjunction with ON PAGE to print a footer on each page. In this example, a page length of 5 is set up (lines 0 through 4). Text is printed on four lines: 0,1,2,3, and the footer prints on line 4. Eight lines of text are printed on two pages:

```
set talk off
clear
set printer on
_padvance="LINEFEEDS"
_plength=5
_pageno=1
eject
on page at line 3 do Page_Brk
printjob
for i = 1 to 8
  ?? "Line of text   ",_pageno,_plineno,i
  ?
endfor
```

PCOL()

```
on page // turn off on page
endprintjob
close printer

procedure Page_Brk
?? "    Page Footer",_pageno,_plineno
eject page
return
////
// This prints out as:
//
// Line of text          1          0          1
// Line of text          1          1          2
// Line of text          1          2          3
// Line of text          1          3          4
//   Page Footing        1          4
// Line of text          2          0          5
// Line of text          2          1          6
// Line of text          2          2          7
// Line of text          2          3          8
//   Page Footer         2          4
```

See Also `?, ??, _pageno, _plineno, EJECT PAGE, PRINTJOB...ENDPRINTJOB, SET PCOL, SET PRINTER, SET PROW`

PCOL()

Returns the printing column position of a printer. Column numbers begin at 0.

Syntax `PCOL()`

Description Use `PCOL()` to determine the horizontal printing position of a printer—that is, the column at which the printer is set to begin printing. Use `PCOL()` in mathematical statements to direct the printer to begin printing at a position relative to its current column position. For example, `PCOL() + 5` represents a position five columns to the right of the current position, and `PCOL() - 5` represents a position five columns to the left of the current position.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of `_ppitch`. See the table in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font of the parent form window.

`PCOL()` returns a column number that reflects the current value of `_ppitch`, regardless of whether you're printing with proportional or monospaced fonts. If you're printing with a proportional font, you can add and subtract fractional numbers to and from the `PCOL()` value to move the printing position accurately.

SET PRINTER must be ON or SET DEVICE TO PRINTER must be in effect for PCOL() to return a column position; otherwise, it returns 0.

Example The following example writes "Jack & Jill" to the printer. It uses PCOL() to note the column position three times, at the beginning, after "Jack", and after "Jill":

```
set talk off
set printer on
// now ?s are directed to printer
?          // sets printer at col 0 of next line
beginpos=pcol() // note the current column
?? "Jack"
lastjackpos=pcol()
?? " & Jill"
lastjillpos=pcol()
set printer off
close printer
? beginpos      // 0.00
? lastjackpos   // 4.00
? lastjillpos   // 11.00
set talk on
```

See Also COL(), PROW(), SET DEVICE, SET PCOL, SET PRINTER

PRINTJOB...ENDPRINTJOB

Uses the values stored in system memory variables to control a printing operation.

Syntax printjob
<statements>
ENDPRINTJOB

<statements> Any valid dBASE statements.

Description Use PRINTJOB...ENDPRINTJOB to control a printing operation with the values of the system memory variables _pbpage, _pepage, _pcopies, _peject, and _plineno. When dBASE begins executing PRINTJOB, it does the following:

- 1 Closes the current print document (if any) and begins a new one, as if you had issued CLOSE PRINTER before issuing PRINTJOB
- 2 Ejects a page if _peject is set to "BEFORE" or "BOTH"
- 3 Sets _pcolno to 0

When dBASE reaches ENDPRINTJOB, it does the following:

- 1 Ejects a page if _peject is set to "AFTER" or "BOTH"
- 2 Resets _pcolno to 0

Before using PRINTJOB...ENDPRINTJOB, set the relevant system memory variables and issue SET PRINTER ON. After ENDPRINTJOB, use CLOSE PRINTER to close and print the document.

PRINTSTATUS()

Example The following example uses PRINTJOB to print one line of text making three copies:

```
_pcopies=3           // 3 copies
_peject="none"       // no page eject before or after
_plineno=0           // initialized to 0
set printer on
printjob
? "A one line print job"
?
endprintjob
close printer        // initiate printing
// prints:
// A one line print job
//
// A one line print job
//
// A one line print job
//
```

See Also _pbpage, _pcopies, _peject, _pepage, _plineno, EJECT, EJECT PAGE, ON PAGE, SET PRINTER

PRINTSTATUS()

Returns *true* if the print device is ready to accept output.

Syntax PRINTSTATUS([<port name expC>])

<port name expC> A character expression such as "lpt1" that identifies the printer port to check.

Description Use PRINTSTATUS() to determine whether you've designated a printer port as an output device with SET PRINTER TO <port name expC>. In *Visual dBASE*, the Windows Print Manager spools print output to and manages the printer port. Therefore, the Print Manager informs you when a printer isn't ready to receive output.

If you don't pass <port name expC> to PRINTSTATUS(), it checks the default port you specified with SET PRINTER TO. PRINTSTATUS() returns only *false* if you haven't specified a printer port with SET PRINTER TO or if the port you specify hasn't been set with SET PRINTER TO.

Note *Visual dBASE* automatically executes SET PRINTER TO on startup if the WIN.INI file contains a valid printer definition. See your Windows documentation for information on WIN.INI settings.

Example This example reads the default PRINTSTATUS and then queries LPT1, LPT2 and LPT3:

```
? printstatus()
? printstatus("lpt1")
```



```
? printstatus("lpt2")
? printstatus("lpt3")
```

See Also CLOSE..., SET DEVICE, SET PRINTER

PROW()

Returns the printing row position of a printer. Row numbers begin at 0.

Syntax PROW()

Description Use PROW() to determine the vertical printing position of a printer—that is, the row at which the printer is set to begin printing. Use PROW() in mathematical statements to direct the printer to begin printing at a position relative to its current row position. For example, PROW() + 5 represents a position five rows below the current position and PROW() – 5 represents a position five rows above the current position.

When you direct output to the printer, *Visual* dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of `_ppitch`. See the table in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font of the parent form window.

If you're printing with a proportional font, you can add and subtract fractional numbers to and from the PROW() value to move the printing position accurately. If you issue ? without the STYLE option and use only integer coordinates, *Visual* dBASE uses the default printer font (typically Courier or another monospaced font).

SET PRINTER must be ON or SET DEVICE TO PRINTER must be in effect for PROW() to return a row position; otherwise, it returns 0.

See Also PCOL(), ROW(), SET PROW

SET ALTERNATE

Controls the recording of input and output in an alternate text file.

Syntax SET ALTERNATE on | OFF

SET ALTERNATE TO [`<filename>` | ? | `<filename skeleton>`] [ADDITIVE]]

<filename> | ? | <filename skeleton> The alternate text file, or target file, to create or open. The ? and `<filename skeleton>` options display a dialog box in which you can specify a new file or select an existing file. If you specify a file without including its path, dBASE looks for the file in the current directory,

then in the path you specify with SET PATH. If you specify a file without including its extension, dBASE assumes .TXT.

ADDITIVE Appends dBASE output that appears in the results pane of the Command window to the specified existing alternate file. If the file doesn't exist, dBASE returns an error message.

Default The default for SET ALTERNATE is OFF. To change the default, set the ALTERNATE parameter in the [OnOffSetting Settings] section of VDB.INI. To set a default file name for use with SET ALTERNATE, specify an ALTERNATE parameter in the [CommandSettings] section of VDB.INI.

Description Use SET ALTERNATE TO to create a record of dBASE output and commands. You can edit the contents of this file with the Text Editor for use in documents, or store it on disk for future reference. You can record, edit, and incorporate command sequences into new programs.

SET ALTERNATE TO <filename> only opens an alternate file, while SET ALTERNATE ON | OFF controls the storage of input and output to that file. Only one alternate file can be open at a time. When you issue SET ALTERNATE TO <filename> to open a new file, dBASE closes the previously open alternate file.

When SET ALTERNATE is ON, dBASE stores output to the results pane of the Command window in the text file you've opened by previously issuing SET ALTERNATE TO <filename>. An alternate file must be open for SET ALTERNATE ON to have an effect. SET ALTERNATE doesn't affect a program's output; it only determines when that output is saved in the alternate file. (Keyboard entries in the Command window aren't stored to the alternate file.)

To prevent your text file from beginning with a blank line, use two question marks (??) before the first word that you send to the alternate file.

Issuing SET ALTERNATE OFF does not close the alternate file. Before accessing the contents of an alternate file, formally close it with CLOSE ALTERNATE or SET ALTERNATE TO (with no file name). This ensures that all data recorded by *Visual* dBASE for storage in the alternate file is transferred to disk, and automatically turns SET ALTERNATE to OFF.

If SET SAFETY is ON and you don't use the ADDITIVE option, and a file exists with the same name as the target file, *Visual* dBASE displays a dialog box asking if you want to overwrite the file. If SET SAFETY is OFF and you don't use the ADDITIVE option, any existing file with the same name as the target file is overwritten without warning.

Example This example uses the SET ALTERNATE commands to write text to the screen and an ASCII file:

```
set alternate to Rose
// Open Rose.txt for text output
? "Opening alternate file" // to screen only
set alternate on
```

```
// ?,?? commands now go to Rose.txt
? "A rose "
set alternate off // Stop storing to Rose.txt.
?? "tended carefully in your garden "
set alternate on // Add to Rose.txt.
?? "is a rose "
?? "is a rose "
? "You will be proud of"
close alternate // Close Rose.txt
// Rose.txt contains:
// A rose is a rose is a rose
// You will be proud of
```

See Also CLOSE..., SET DEVICE

SET CONSOLE

Controls the display of output in the results pane of the Command window during program execution.

Syntax SET CONSOLE ON | off

Description When SET CONSOLE is ON, *Visual* dBASE displays all text stream output in the results pane of the Command window. Use SET CONSOLE OFF to disable this output. For example, if you are creating a text file with SET ALTERNATE, you usually do not need to see the output in the Command window at the same time. By using SET CONSOLE OFF, your program will execute faster.

Whenever the input pane of the Command window gets focus, SET CONSOLE is always turned ON. The SET CONSOLE command has no effect when issued in the Command window.

You may use the WAIT command while SET CONSOLE is OFF; however, *Visual* dBASE displays neither the prompt for the input nor the input itself.

Example In the following example, SET CONSOLE is used with SET ALTERNATE when creating a text file:

```
set console off
?
set alternate to RESULTS.TXT
// Generate text file
close alternate
set console on
```

After disabling output to the Command window, the ? command is used to help ensure that the output in the text file starts at the beginning of the line.

See also SET ALTERNATE, SET PRINTER, WAIT

SET MARGIN

Sets the width of the left border of a printed page.

Syntax SET MARGIN TO <expN>

<expN> The column number at which to set the left margin. The valid range is 0 to 254, inclusive. You can specify a fractional number for <expN> to position output accurately with a proportional font.

Default The default for SET MARGIN is 0. To change the default, set the MARGIN parameter in VDB.INI.

Description Use SET MARGIN to adjust the printer offset for the left margin for all printed output. The margin established by SET MARGIN becomes the printer's column 0 position. set margin resets the value of the _ploffset system memory variable but doesn't affect the value of the _lmargin system memory variable.

Use SET MARGIN to adjust the position of text on the printed page according to the type of paper. For example, if you're printing to three-hole paper, you might need to increase the left border. You can also use SET MARGIN to compensate for the placement of paper in the printer. For example, if the paper is off-center in the printer, you can adjust the width of the left border to properly place the text.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of _ppitch. See the table in the description of _ppitch, which lists _ppitch values. The height of each character cell is determined by the size of the font.

If you change the value of SET MARGIN, dBASE takes the current value of _ppitch into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

Example This example displays the 10 digits with the default margin and then sets the margin to column 10 and displays the 10 digits this time indented:

```
set printer on
set margin to 0                // The default
? "1234567890"
set margin to 10
? "1234567890"
? _ploffset
set margin to 0                // reset margin
set printer off
close printer
```

Produces:

```
// this displays as:
// 1234567890
```

```
//          1234567890
//          10
//
// _ploffset is set by set margin
```

See Also `_indent`, `_lmargin`, `_ploffset`, `_rmargin`

SET PCOL

Sets the printing column position of a printer, which is the value of `PCOL()`.

Syntax `SET PCOL TO <expN>`

<expN> The column number to which to set `PCOL()`. The valid range is 0 to 32,767, inclusive.

Description Use `SET PCOL` to set the horizontal printing position of a printer, which is the value the `PCOL()` function returns. Generally, you use the command `SET PCOL TO 0` to reset the printer column to the left edge of the page.

When you move the printing position to a new line, `dBASE` reinitializes `PCOL()` to 0, so `SET PCOL` affects the value of `PCOL()` for the current line only. When you send output to your printer, `dBASE` updates `PCOL()` by adding 1 to the current `PCOL()` value for each character it sends to the printer. The printing position moves one column for each character the printer prints.

When you send a printer control code or escape sequence to your printer, the printing position doesn't move. (Printer control codes and escape sequences are strings that give the printer instructions, such as to print underlining, boldface type, or different fonts.) Although control codes and escape sequences don't move the printing position, `dBASE` nonetheless increments the `PCOL()` value by the number of characters that you send to the printer. Each control code character increments the value of `PCOL()` by 1 just like any other character. As a result, the value of `PCOL()` might not reflect the actual printing position. Use `SET PCOL` to reset the value of `PCOL()` to the same value as the printing position.

To send a control code to the printer without changing the value of `PCOL()`, save the current value of `PCOL()` to a memory variable, send the control code to the printer, then `SET PCOL` to the contents of the memory variable.

Example The following example writes "Jack & Jill" to the printer. It uses `PCOL()` to note the column position three times, at the beginning, after "Jack", and after "Jill":

```
set talk off
set printer on
// now ?s are directed to printer
?                               // printer at col 0 of next line
beginpos=pcol()                // note the current column
?? "Jack"
lastjackpos=pcol()
```

```

?? " & Jill"
lastjillpos=pcol()
// prints:
// Jack & Jill
set printer off
close printer
? beginpos                // 0.00
? lastjackpos              // 4.00
? lastjillpos              // 11.00
// Displays column positions for reference

```

See Also PCOL(), PROW(), SET PROW

SET PRINTER

The SET PRINTER TO setting specifies a file to receive streaming output, or uses a device code recognized by the Windows Print Manager to designate a printer. The On/Off setting controls whether dBASE also directs streaming output that appears in the Command window to the device or file specified by SET PRINTER TO.

Syntax SET PRINTER on | OFF

SET PRINTER TO [<filename> | ? | <filename skeleton>] | [<device>]

<filename> | ? | <filename skeleton> The text file to send output to instead of the printer. By default, dBASE assigns a .PRT extension to <filename> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box, in which you specify the name of the target file and the directory to save it in.

<device> The printer port of the printer to send output to. Specify printers and their ports with the Windows Control Panel.

Default The default for SET PRINTER is OFF. To change the default, set the PRINT parameter in the [OnOffCommandSettings] section in VDB.INI. The default for SET PRINTER TO is the default printer you specify with the Windows Control Panel.

Description Use SET PRINTER TO to direct *streaming output* from commands such as ?, ??, and LIST to a printer or a text file. SET PRINTER TO with no option sends this output to the default printer.

Use SET PRINTER ON/OFF to enable or disable the printer you specify with SET PRINTER TO.

To send streaming output to a file rather than the printer, issue SET PRINTER TO FILE <filename>. When you issue SET PRINTER TO FILE <filename>, issuing SET PRINTER ON directs streaming output to the text file <filename> rather than to the printer. The file has the default extension of .PRT.

When SET PRINTER is OFF, dBASE directs streaming output only to the result pane of the Command window. SET PRINTER must be ON to output data to a text file unless you issue a command with its TO PRINTER option. The following example illustrates this behavior:

```
set printer off
set printer to file test.prt
type file.txt                      // displays on screen only
type file.txt to print              // output sent to screen and test.prt
```

To send non-streaming output to the printer, use SET DEVICE TO PRINTER. To send non-streaming output to a text file, use SET DEVICE TO FILE.

Example The following example uses SET PRINTER ON and OFF:

```
set printer to                      // Set printer to default
set printer on
? "Hello"                          // to printer
set printer off
? prow(),pcol()                   // only displayed to screen
close printer                     // initiate printing
set printer to                    // resets to default
set printer to prn                // sets to DOS output device
set printer to lpt1
set printer to null
set printer to file Test
// Test.prt receives streaming output, including any control codes,
// that would have gone to the printer.
```

See Also SET ALTERNATE, SET CONSOLE, SET DEVICE

SET PROW

Sets the current row position of a printer's print head, which is the value of PROW().

Syntax SET PROW TO <expN>

<expN> The row number to which to set PROW(). The valid range is 0 to 32,767, inclusive.

Description Use SET PROW to set the vertical printing position of a printer, which is the value the PROW() function returns. Generally, you use the command SET PROW TO 0 to reset the printer row to top-of-page.

See Also PCOL(), PROW(), SET PCOL

SET SPACE

Determines whether dBASE inserts a space between expressions displayed or printed with a single ? or ?? command.

Syntax SET SPACE ON | off

Default The default for SET SPACE is ON. To change the default, set the SPACE parameter in VDB.INI.

Description Use SET SPACE OFF when you use a single ? or ?? command to print a list of expressions and you don't want spaces between the expressions. If you want the expressions printed with spaces between them, issue SET SPACE ON.

SET SPACE has no effect on multiple ? or ?? commands. For example, if you issue the command ?? <exp> twice, the second instance of <exp> will be printed adjacent to the first, even if SET SPACE is ON. However, if SET SPACE is ON and you issue ?? <exp>, <exp> as a single command, there will be a space between the two instances of <exp>.

Example This example displays a first and a last name using SET SPACE ON and then SET SPACE OFF:

```
Firstname="Rachel"
Lastname ="Jays"
set space on                // the default
? Firstname,Lastname
// Rachel Jays
set space off
? Firstname,Lastname
// RachelJays
// The two variables are not separated
```

See Also ?, ??, LTRIM(), RTRIM(), TRIM()

_alignment

Left-aligns, right-aligns, or centers ? and ?? command output within margins specified by _lmargin and _rmargin when _wrap is true.

Syntax _alignment = <expC>

<expC> The character expression "LEFT", "CENTER", or "RIGHT". You can enter <expC> in any combination of uppercase and lowercase letters.

Default The default for _alignment is "LEFT".

Description Use `_alignment` to left-align, right-align, or center output from the `?` and `??` commands between the margins you set with `_lmargin` and `_rmargin`. The `_alignment` setting is effective only when `_wrap` is true (*true*).

To control the alignment of text within a field, use the "B," "I," and "J" format options with the PICTURE or FUNCTION options.

Example The following example sets wrap on and then prints in the three different alignments: left, center, and right:

```
savewrap=_wrap      // save last wrap setting
_wrap=true          // must be true for alignment
savealign=_alignment // save last alignment setting
_alignment="left"
? "Hello left"
_alignment="right"
? "Hello right"
_alignment="center"
? "Hello center"
_alignment=savealign // reset
_wrap=savewrap       // reset
```

See Also `?, ??, _lmargin, _rmargin, _wrap, SET MARGIN`

_indent

Specifies the number of columns to indent the first line of a paragraph of `?` command output when `_wrap` is true.

Syntax `_indent = <expN>`

<expN> The column number, relative to the left margin, where the first line of a new paragraph begins. You can specify a fractional number for `<expN>` to position output accurately with a proportional font.

Default The default for `_indent` is 0.

Description Use `_indent` to specify where the first line of a new paragraph begins relative to the left margin. (Specify the left margin with `_lmargin`.) The `_indent` setting is effective only when `_wrap` is *true*.

When you direct `?` output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of `_ppitch`. See the table in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font.

If you change the value of `_indent`, dBASE takes the current value of `_ppitch` into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

To indent the first line of a paragraph, use a value greater than 0. For example, to begin the line five columns to the right of the left margin, set `_indent` to 5. To create a hanging indent (sometimes called an *outdent*), use a negative value. For example, to begin the first line five columns to the *left* of the left margin, set `_indent` to -5. Using the default value of 0 (no indent or outdent) aligns all lines in a paragraph to the left margin. The sum of `_lmargin` and `_indent` must be greater than 0 and less than `_rmargin`.

Example The following example sets wrap on and indents the first line of a text that wraps around:

```
_indent=3           // set the indentation
savewrap=_wrap      // save last wrap setting
_wrap=true          // must be true for alignment
savelmarg=_lmargin  // save last alignment setting
_lmargin=5
savermarg=_rmargin  // save last alignment setting
_rmargin=20
? "New York, Chicago and Boston are "+
  "cold in wintertime."
// Now the text wraps around between columns 5 and 20
//
//      New York,
//      Chicago and
//      Boston are cold
//      in wintertime.
//
_rmargin=savermarg  // restore the previous margin
_lmargin=savelmarg  // restore the previous margin
_wrap=savewrap      // reset wrap
```

See Also `?, ??, _alignment, _lmargin, _ploffset, _rmargin, _wrap, SET MARGIN`

`_lmargin`

Defines the left margin for `?` and `??` command output when `_wrap` is true.

Syntax `_lmargin = <expN>`

<expN> The column number of the left margin. The valid range is 0 to 254, inclusive. You can specify a fractional number for `<expN>` to position output accurately with a proportional font.

Default The default for `_lmargin` is 0.

Description Use `_lmargin` to set the left margin for `?` and `??` command output. If you're sending output to a printer, `_lmargin` sets the left margin from the `_ploffset` (page left offset) column. For example, if `_ploffset` is 10 and `_lmargin` is 5,

output prints from the 15th column. The `_lmargin` setting is effective only when `_wrap` is *true*.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of `_ppitch`. See the table in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font.

If you change the value of `_lmargin`, dBASE takes the current value of `_ppitch` into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

If you use `_indent` to specify the indentation of the first line of each paragraph, the combined values of `_lmargin` and `_indent` must be less than the value of `_rmargin`.

Example The following example uses `_lmargin`. It sets wrap on and then changes the left margin and displays text:

```
_wrap=true           // must be true for _lmargin
_lmargin=0
? "01234567890"
_lmargin=5
? "Changing the margin"
// produces:
// 01234567890
//      Changing the margin
```

See Also `?, ??, _alignment, _indent, _ploffset, _rmargin, _wrap, SET MARGIN, Style`

_padvance

Determines whether the printer advances the paper of a print job with a formfeed or with linefeeds.

Syntax `_padvance = <expC>`

<expC> The character expression "FORMFEED" or "LINEFEEDS".

Default The default for `_padvance` is "FORMFEED".

Description Use `_padvance` to specify whether dBASE advances the paper to the top of the next sheet one sheet at a time using a formfeed character, or one line at a time using linefeed characters. If you use the default "FORMFEED" setting, the paper advances according to the printer's default form length setting.

Tractor-feed printers (such as dot matrix printers) generally use a "LINEFEEDS" setting, while form feed printers (such as laser printers) generally use a "FORMFEED" setting.

Note Sending CHR(12) to the printer always issues a formfeed, even if you set _padvance to "LINEFEEDS".

Use the "LINEFEEDS" setting if you change the length of the paper or want to print a different number of lines than the default form length of the printer without adjusting its setting. For example, to print short pages, such as checks that are 20 lines long, set _plength to the length of the output (20 in this example) and _padvance to "LINEFEEDS."

The number of linefeeds dBASE uses to reach the top of the next page depends on whether you issue an eject during streaming or non-streaming output mode.

An eject occurs during streaming output mode when you issue:

- EJECT PAGE without an ON PAGE handler
- EJECT PAGE with an ON PAGE handler when the current line position is past the ON PAGE line
- PRINTJOB or ENDPRINTJOB and _peject causes an eject

In these cases, dBASE calculates the number of linefeeds to send to the print device using the formula $_plength - _plineno$.

An eject occurs in nonstreaming output mode when you issue:

- EJECT
- SET DEVICE TO PRINTER and force a page eject with the @ command

In these cases, dBASE calculates the number of linefeeds to send to the print device using the formula $_plength - \text{MOD}(\text{PROW}(), _plength)$.

Example There are two _padvance settings:

```
_padvance="formfeed"
_padvance="linefeeds"
```

See Also _peject, _plength, EJECT, EJECT PAGE, PRINTJOB...ENDPRINTJOB, ON PAGE, SET DEVICE

_pageno

Determines or sets the current page number.

Syntax _pageno = <expN>

<expN> An integer from 1 to 32,767, inclusive.

Description Use _pageno to number pages of *streaming output* from commands such as ?, ??, and LIST.

With _pageno, you can determine the current page number or set the page number to a specific value. Use it to print page numbers in a report or, when

combining documents, to assign an incremented number to the first page of the second document.

A page break occurs when the value of `_plineno` (the line number count) becomes greater than the value of `_plength` (the currently defined printed page length in lines). At each page break of streaming output, dBASE automatically increments the value of `_pageno`.

Example This example prints 100 lines of output and prints a heading on line 1 of each page:

```
set talk off
set printer on
_pageno=1
for i=1 TO 100
  if _plineno=1 // At first line of page
    ? "Top of Page ",_pageno
  endif
  ? "Line",i
endifor
set printer to
close printer
set talk on
```

See Also `?, ??, _pbpage, _pepage, _plength, _plineno, LIST, ON PAGE`

_pbpage

Specifies the page number of the first page PRINTJOB prints.

Syntax `_pbpage = <expN>`

<expN> The page number at which to begin printing. The valid range is 1 to 32,767, inclusive. Specify a positive integer for `<expN>`.

Default The default for `_pbpage` is 1.

Description Use `_pbpage` to begin printing a print job at a specific page number. Pages with numbers less than `_pbpage` don't print. To stop printing at a specific page number, use `_pepage`.

If you set `_pbpage` to a value greater than `_pepage`, dBASE returns an error.

Example This example uses `_pbpage` to omit a page of a report. It outputs 100 lines and prints the page and line number on each line as in the example for `_plineno`. Here, the beginning page number is set to 2 so that page 1 does not print:

```
_pageno=1
_pbpage=2 // begin on page 2
set printer on
printjob
for i=1 TO 100
```

`_pcolno`

```
?? "Page",_pageno," Line",_plineno
?                               // now force a linefeed
endfor
endprintjob
set printer off
close printer                 // start printing
```

See Also `_pageno`, `_pepage`, `PRINTJOB...ENDPRINTJOB`

`_pcolno`

Identifies or sets the current column number of streaming output.

Syntax `_pcolno = <expN>`

<expN> The column number at which to begin printing. The valid range is 0 to 255, inclusive. You can specify a fractional number for *<expN>* to position output accurately with a proportional font.

Default The default for `_pcolno` is 0.

Description Use `_pcolno` to position printing streaming output from commands such as `?`, `??`, and `LIST`.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of `_ppitch`. See the table in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font.

If you change the value of `_pcolno`, dBASE takes the current value of `_ppitch` into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

The `PCOL()` function also returns the current printhead position of the printer, but if `SET PRINTER` is `OFF`, the `PCOL()` value doesn't change. `_pcolno`, on the other hand, returns or assigns the current position in the streaming output regardless of the `SET PRINTER` setting.

Example This example displays the numbers 1 through 5. It uses `_pcolno` to position the numbers so that each number begins at its own position:

```
set talk off
for i=1 to 5
  _pcolno=i           // set the column position
  string=ltrim(str(i))
  // convert i to a single character
  ?? string
  ?
endfor
```

```
set talk on
// the output looks like this
// 1
// 2
// 3
// 4
// 5
```

See Also `?, _plineno, _rmargin, PCOL(), SET DEVICE, SET PRINTER`

_pcopies

Specifies the number of copies to print for a PRINTJOB.

Syntax `_pcopies = <expN>`

<expN> The number of copies to print. The valid range is 1 to 32,767, inclusive. Specify a positive integer for <expN>.

Default The default for _pcopies is 1.

Description Use _pcopies to print a specific number of copies of a print job. You can assign a value to _pcopies in the Command window or in a program. The value of _pcopies has an effect only when you send a print job to the printer by issuing PRINTJOB. In a program, assign a value to _pcopies before issuing PRINTJOB.

Example This example uses _pcopies to print the print job three times:

```
_pcopies=3           // Three copies
set printer on
printjob
? "Very Small Report"
endprintjob
set printer off
close printer
```

See Also `PRINTJOB...ENDPRINTJOB`

_pdriver

Identifies the current printer driver or activates a new driver.

Syntax `_pdriver = <expC>`

<expC> The name of the printer driver to activate.

Default The default for _pdriver is the printer driver you specify with the Windows Control Panel. If you haven't specified a printer driver with the Control Panel, the value of _pdriver is an empty string ("").

Description Use `_pdriver` to identify the current printer driver or to activate an installed driver. (To install a new printer driver, use the Windows Control Panel.)

The `_pdriver` value contains two elements separated by a comma: the base file name of the Windows driver file and the name of the printer as it appears in WIN.INI. The current driver might not identify a printer name, in which case, `_pdriver` contains only the driver file name. For example, if the current printer driver is for the HP Laserjet IIISi PostScript printer, `_pdriver` may contain the value "pscript,HP LaserJet IIISi PostScript". To activate this driver, issue the command `_pdriver = "pscript,HP LaserJet IIISi PostScript"`.

To activate a driver from within dBASE, it may be easier to use `CHOOSEPRINTER()` than to assign a value to `_pdriver`. To activate a driver in Windows, use the Printers program of the Windows Control Panel. `CHOOSEPRINTER()` opens the Print Setup dialog box, in which you can also select options such as paper size, source, and orientation (portrait or landscape). In the Windows Control Panel, you can choose Setup to select these options.

Example Use `_pdriver` to determine the current print driver:

```
? _pdriver
// With an Epson FX 80, the response is:
//      EPSON9,Epson FX-80
// With an HP Laserjet running postscript, the
// response is:
//      pscript,HP LaserJet IIISi PostScript
```

You can set the print driver with `_pdriver`:

```
_pdriver="pscript"
```

See Also `_pform`, `_ppitch`, `_pquality`, `CHOOSEPRINTER()`, `SET PRINTER`

`_pject`

Determines whether dBASE ejects a sheet of paper before and after a `PRINTJOB`.

Syntax `_pject = <expC>`
<expC> The character expression "before", "after", "both", or "none".

Default The default for `_pject` is "before", which tells the printer to eject a sheet of paper before starting the print job.

Description Use `_pject` to specify if and when the printer should eject a sheet of paper. Assign a new value to `_pject` (and to any other system memory variable) before issuing `PRINTJOB` in a program to make the new value affect the print job.

The following table describes _peject options.

| <expC> | Result |
|----------|---|
| "before" | Eject sheet before printing the first page |
| "after" | Eject sheet after printing the last page |
| "both" | Eject sheet before and after the print job |
| "none" | Don't eject sheet before or after the print job |

Note The _peject system memory variable is distinct from the EJECT command, which tells the printer to advance the paper to the top of the next page.

Example This example shows the four possible _peject setting. The last setting is operational in the PRINTJOB:

```
_peject="before"
_peject="after"
_peject="both"
_peject="none"
// _peject must be set before printjob
printjob
? "Hello World"
endprintjob
```

See Also _padvance, EJECT, EJECT PAGE, PRINTJOB...ENDPRINTJOB

_pepage

Specifies the page number of the last page of a print job.

Syntax _pepage = <expN>

<expN> The page number of the last page to print. The valid range is 1 to 32,767, inclusive. You must specify a positive integer for <expN>.

Default The default for _pepage is 32,767.

Description Use _pepage to stop printing a print job at a specific page number. Pages with numbers greater than _pepage don't print. To begin printing at a specific page number, use _pbpage.

If you set _pepage to a value less than _pbpage, dBASE returns an error.

Example This example selects two pages from a PRINTJOB. The program prints 500 lines of output and prints the page and line number on each line as in the example for _plineno. Here, the ending page number is set to 2 so that only pages 1 and 2 print:

```
_pageno=1
_pbpage=1 // reset the default pbpage
_pepage=2 // end on page 2
```

```
set printer on
printjob
for i=1 TO 500
    ?? "Page",_pageno," Line",_plineno
    ?                               // now force a linefeed
endfor
endprintjob
set printer off
close printer                     // begin printing
```

The ? command issues a linefeed before processing consequently in this case, the correct line number is obtained by using ??.

See Also `_pageno`, `_pbpage`, `PRINTJOB...ENDPRINTJOB`

`_pform`

Identifies the current print form file or activates another one.

Syntax `_pform = <filename>`

<filename> The name of a print form file (.PRF).

Default The default for `_pform` is an empty string ("").

Description Use `_pform` to determine the name of the current print form file or to activate another one. A print form file (.PRF) is a binary file that contains print settings for printing a print job. The print form file contains the following system memory variables:

| Variable | Action |
|------------------------|---|
| <code>_padvance</code> | Determines whether the printer advances the paper with a formfeed or linefeeds. |
| <code>_pageno</code> | Determines or sets the current page number. |
| <code>_pbpage</code> | Specifies the page number at which PRINTJOB begins printing. |
| <code>_pcopies</code> | Specifies the number of copies to print in a printjob. |
| <code>_pdriver</code> | Activates a specified printer driver. (If the print form file is from dBASE IV, Visual dBASE ignores this value.) |
| <code>_peject</code> | Controls page ejects before and after PRINTJOB. |
| <code>_pepage</code> | Specifies the number of the last page that PRINTJOB prints. |
| <code>_plength</code> | Specifies the number of lines per page for streaming output. |
| <code>_ploffset</code> | Determines the width of the left border on a printed page. |
| <code>_ppitch</code> | Sets the printer pitch, the number of characters per inch that the printer prints. |
| <code>_pquality</code> | Specifies if the printer prints in letter-quality or draft mode. |
| <code>_pspacing</code> | Sets the line spacing for streaming output. |

When you specify a print form file by assigning its name to `_pform`, the values stored in the file are assigned to their respective variables. Jobs you send to the printer then behave in accordance with these variables.

Example This example assumes there are two reports, Report1 and Report2. It uses Report1's print form file, Report1.PRF to print Report2:

```
_pform= "Report1"
report form Report2
```

See Also PRINTJOB...ENDPRINTJOB

_plength

Specifies the number of lines per page for streaming output.

Syntax `_plength = <expN>`

<expN> The number of lines per page. The valid range is 1 to 32,767, inclusive. You can specify a fractional number for `<expN>` to position output accurately with a proportional font.

Default The default page length is determined by the default page size of the current printer driver and the current page orientation (portrait or landscape).

Description Use `_plength` to specify a page length that is different from the default of the current printer driver. For example, to print short pages, such as checks that are 20 lines long, set `_plength` to the length of the output (20 in this example) and `_padvance` to "LINEFEEDS" to advance to the top of the next page.

When you change printer drivers or page orientation, dBASE changes the value of `_plength` automatically. You can change printer drivers in dBASE by issuing CHOOSEPRINTER() or by assigning a value to `_pdriver`. In Windows, you can change printer drivers with the Printers program of the Control Panel (however, it won't take effect until you quit dBASE and start a new dBASE session). You can change page orientation in any of these ways or, in dBASE, by changing the value of `_porientation`.

Example This example sets the form length to 10 lines and prints 25 lines of output. Each line simply prints line number and count (1 through 25). A three-line heading prints "Top of Page" on each line 1:

```
_plength=10
_pageno=1
_plineno=0
set printer on
for i=1 TO 25
  if _plineno=0 // At first line of page
    ?
    ? "Top of Page ",_pageno
    ?
  ?
  _plineno=_plineno+1
endfor
```

`_plineno`

```
endif
? "Line",_plineno,"i=",i
endfor
set printer off
close printer
// The first two pages are:
//
// Top of Page          1
//
// Line          3.00 i=          1
// Line          4.00 i=          2
// Line          5.00 i=          3
// Line          6.00 i=          4
// Line          7.00 i=          5
// Line          8.00 i=          6
// Line          9.00 i=          7
//
// Top of Page          2
//
// Line          3.00 i=          8
// Line          4.00 i=          9
// Line          5.00 i=         10
// Line          6.00 i=         11
// Line          7.00 i=         12
// Line          8.00 i=         13
// Line          9.00 i=         14
```

See Also `_padvance`, `_pdriver`, `_porientation`, `CHOOSEPRINTER()`, `EJECT`, `EJECT PAGE`

`_plineno`

Identifies or sets the current line number of streaming output.

Syntax `_plineno = <expN>`

<expN> The line number at which to begin printing. The valid range is 0 to `_plength - 1`. You can specify a fractional number for `<expN>` to position output accurately with a proportional font.

Default The default for `_plineno` is 0.

Description Use `_plineno` to position printing streaming output from commands such as `?`, `??`, and `LIST`.

The `PROW()` function also returns the current printhead position of the printer, but if `SET PRINTER` is `OFF`, the `PROW()` value doesn't change. `_plineno`, on the other hand, returns or assigns the current position in the streaming output regardless of the `SET PRINTER` setting.

Example This example prints 32 lines of output on four pages. The page length is set at 10 lines per page, and the report prints the page and line number on each line using `_pageno` and `_plineno`:

```
_pbpage=1      // reset the default pbpage
_pegpage=32767 // reset the default pegpage
_plength=10    // set the page length to 10 lines
_pageno=1
set printer on
printjob       // printjob resets _plineno to 0
for i=1 TO 32
  if _plineno=0 // At first line of page
    //?
    ? "Top of Page ",_pageno
    ?
  endif
  ?? "Page",_pageno," Line",_plineno,i
  ? // now force a linefeed
endfor
endprintjob
set printer off
close printer
// The first two pages of output appear as follows:
//
// Top of Page          1
// Page      1 Line      2.00      1
// Page      1 Line      3.00      2
// Page      1 Line      4.00      3
// Page      1 Line      5.00      4
// Page      1 Line      6.00      5
// Page      1 Line      7.00      6
// Page      1 Line      8.00      7
// Page      1 Line      9.00      8
//
// Top of Page          2
// Page      2 Line      2.00      9
// Page      2 Line      3.00     10
// Page      2 Line      4.00     11
// Page      2 Line      5.00     12
// Page      2 Line      6.00     13
// Page      2 Line      7.00     14
// Page      2 Line      8.00     15
// Page      2 Line      9.00     16
```

See Also `_pcolno`, `_plength`, `_ppitch`, `EJECT PAGE`, `ON PAGE`, `PCOL()`, `PROW()`

_ploffset

Displays or sets the width of the left border of a printed page.

Syntax `_ploffset = <expN>`

<expN> The column number at which to set the left margin. The valid range is 0 to 254, inclusive. You can specify a fractional number for *<expN>* to position output accurately with a proportional font.

Default The default for _ploffset is 0. To change the default, set the MARGIN parameter in VDB.INI.

Description Use _ploffset (page left offset) to specify the distance from the left edge of the paper to the left margin of the print area. Use _lmargin to set the left margin from the _ploffset column. For example, if _ploffset is 10 and _lmargin is 5, output prints from the 15th column.

The _ploffset system memory variable is equivalent to the SET MARGIN value. Changing the value of one changes the other. For more information, see SET MARGIN.

Example See SET MARGIN

See Also _indent, _lmargin, SET MARGIN

_porientation

Determines whether the printer prints in portrait or landscape mode.

Syntax _porientation = <expC>

<expC> The character expression "PORTRAIT" or "LANDSCAPE".

Default The default for _porientation is the orientation you specify with the Printers program of the Windows Control Panel or, in dBASE, with the CHOOSEPRINTER() function. By default, this orientation is portrait.

Description Use _porientation to specify whether you want to print in portrait or landscape mode. When you print in portrait mode, each page is read vertically; a standard American letter-size piece of paper is 8.5 inches wide by 11 inches long. When you print in landscape mode, each page is read horizontally; a standard American letter-size piece of paper is 11 inches wide by 8.5 inches long.

Most printer drivers support landscape printing; however, if you specify landscape while using a printer driver that doesn't support it, the printer continues to print in portrait mode, possibly truncating text.

Changing page orientation automatically resets _plength.

Example _porientation has two settings:

```
_porientation="portrait"  
_porientation="landscape"
```

It takes effect only on a page boundary.

See Also _pdriver, _plength, _ppitch, SET PRINTER

_ppitch

Sets the printer pitch, the number of characters per inch that the printer prints.

Syntax _ppitch = <expC>

<expC> The character expression "pica", "elite", "condensed", or "default".

Default The default for _ppitch is "default", the pitch defined by your printer's settings or by setup codes or commands you sent to the printer before you started dBASE. "Default" means that dBASE hasn't sent any pitch control codes to the printer.

Description Use _ppitch to set the pitch (characters per inch) on the printer. The _ppitch setting sends a control code appropriate to the current printer driver. Use the Windows Control Panel or CHOOSEPRINTER() to select the printer driver.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose sizes depend on the value of _ppitch. The height of each character cell is determined by the size of the font.

The following table lists _ppitch values.

| _ppitch value | Character cell width |
|---------------|----------------------------|
| "pica" | 1/10" (10 characters/inch) |
| "elite" | 1/12" (12 characters/inch) |
| "condensed" | 1/17" (17 characters/inch) |

If you change the value in other system memory variables such as _lmargin, _rmargin, and _ploffset, dBASE takes the current value of _ppitch into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

Example This example shows the three settings of _ppitch:

```
s_ppitch=_ppitch  // save current pitch
set printer on
_ppitch="pica"
? "John Brown's body: 10 characters per inch"
_ppitch="elite"
? "John Brown's body: 12 characters per inch"
_ppitch="condensed"
? "John Brown's body: 17 characters per inch"
_ppitch=s_ppitch  // restore original setting
close printer
```

`_pquality`

`_ppitch` is not valid for all printers.

See Also `_pdriver`, `_pquality`, `CHOOSEPRINTER()`

`_pquality`

Specifies whether the printer prints in letter-quality or draft mode. Used primarily with dot-matrix printers; the `_pquality` value usually has no effect on printers that don't support draft mode, such as laser and Postscript printers.

Syntax `_pquality = <expl>`

<expl> The logical expression *true* for letter quality and *false* for draft quality.

Default The default for `_pquality` is *false* for draft mode.

Description Use `_pquality` to determine whether the printer prints in letter-quality or draft mode. Letter-quality mode produces printed copy of higher quality (finer resolution) than draft; however, draft mode usually prints more quickly than letter-quality, depending on the printer.

Example This example shows the two settings for print quality. Print quality cannot be changed in mid page and might or might not be available on your printer:

```
close printer
set printer on
_pquality= false    // draft quality
? "John Brown's body"
close printer
_pquality= true     // letter quality
? "John Brown's body"
close printer
```

See Also `_pdriver`, `_ppitch`

`_pspacing`

Sets the line spacing for streaming output.

Syntax `_pspacing = <expN>`

<expN> The amount of line spacing. The valid range is 1 to 3, inclusive:

- A value of 1 represents single spacing.
- A value of 2 represents double spacing. There is one blank line between printed lines.
- A value of 3 represents triple spacing. There are two blank lines between printed lines.

Paragraph spacing is in multiples of the height of the line just printed, which depends on the tallest font used in printing the line. You can specify a fractional number for *<expN>* to space text by partial line heights.

Default The default for _pspacing is 1, which sets line spacing to single-line.

Description Use _pspacing to set the line spacing of streaming output from commands such as ?, ??, and LIST. To insert a single blank line into output, use the ? command.

Example This example uses _pspacing to set the spacing to 1 then 2 lines between lines of text and finally back to 1 line:

```
_pspacing=1
? "Jack 1"
? "Jill 1"
_pspacing=2
? "Jack 2"
? "Jill 2"
_pspacing=1
? "Jack 1"
? "Jill 1"
// produces:
//      Jack 1
//      Jill 1
//
//      Jack 2
//
//      Jill 2
//      Jack 1
//      Jill 1
```

Notice that _pspacing takes place immediately so that the double spacing occurs before Jack 2 and before Jill 2.

See Also ?, ??, _ppitch, DISPLAY, LIST

_rmargin

Defines the right margin for ? and ?? command output when _wrap is true.

Syntax _rmargin = *<expN>*

<expN> The column number of the right margin. The valid range is 0 to 255, inclusive. You can specify a fractional number for *<expN>* to position output accurately with a proportional font.

Default The default for _rmargin is 79.

Description Use _rmargin to set the right margin for output from the ? and ?? commands. The value of _rmargin must be greater than the value of _lmargin or _lmargin +

_indent. For example, if _lmargin and _indent are both set to 5, _rmargin must be greater than 10 to display at least one column of output.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of _ppitch. See the table in the description of _ppitch, which lists _ppitch values. The height of each character cell is determined by the size of the font.

If you change the value of _rmargin, dBASE takes the current value of _ppitch into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

Example The following example sets wrap on and then changes the left margin and displays text:

```
savewrap=_wrap           // save last wrap setting
_wrap=true               // must be true for alignment
savelmargin=_lmargin     // save last alignment setting
_lmargin=5
savermargin=_rmargin     // save last alignment setting
_rmargin=20
? "New York, Chicago and Boston are cold in wintertime."
// Now the text wraps around between columns 5 and 20
_rmargin=savermargin     // restore the previous margin
_lmargin=savelmargin     // restore the previous margin
_wrap=savewrap           // reset wrap
```

See Also ?, ??, _alignment, _indent, _lmargin, _ploffset, _wrap, SET MARGIN

_tabs

Sets one or more tab stops for output from the ? and ?? commands.

Syntax _tabs = <expC>

<expC> The list of column numbers for tab stops. If you set more than one tab stop, the numbers must be in ascending order and separated by commas. Enclose the entire list in quotation marks. You can specify fractional numbers for <expC> to position output accurately with a proportional font.

Default The default for _tabs is an empty string ("").

Description Use _tabs to define a series of tab stops. If _wrap is true, dBASE ignores tab stops equal to or greater than _rmargin.

When you direct output to the printer, dBASE maps each character according to the *coordinate plane*, a two-dimensional grid. The coordinate plane is divided into character cells whose widths depend on the value of _ppitch. See the table

in the description of `_ppitch`, which lists `_ppitch` values. The height of each character cell is determined by the size of the font.

If you change the value of `_tabs`, dBASE takes the current value of `_ppitch` into consideration when calculating the cell width of the coordinate plane. This happens regardless of whether you're printing with proportional or monospaced fonts.

If you send a tab character, `CHR(9)`, with `?` or `??`, dBASE expands it to the amount of space required to reach the next tab stop. If the tab character you send is past the last tab stop, dBASE ignores it, displaying output starting in the current column.

Example The following program makes two tab stops at columns 5 and 20. The array `A` is displayed beginning at the first tab stop, and its position in the array (`i`) is displayed at the second tab stop:

```
_tabs="5,20"    // tab stops at columns 5 and 20
a = { "One", "Two" }
for i=1 to 2
    ? chr(9),A[i],chr(9),ltrim(str(i))
endfor
// chr(9) is equivalent to the tab key
// produces:
//      One           1
//      Two           2
```

See Also `?, ??, _indent, _lmargin, _rmargin, _wrap, CHR(), MODIFY COMMAND`

_wrap

Determines if streaming output wraps between margins specified by `_lmargin` and `_rmargin`.

Syntax `_wrap = <expL>`

<expL> The logical expression *true* or *false*.

Default The default for `_wrap` is *false*, which disables wrapping.

Description Set `_wrap` to *true* to wrap streaming output from commands such as `?`, `??`, and `LIST` within the margins you specify with `_lmargin` and `_rmargin`.

When you enable wrapping, *Visual* dBASE wraps text onto the next line, breaking between words or numbers, when the output reaches the right margin. When you disable wrapping, *Visual* dBASE extends text beyond the right margin, moving to the next line only when a carriage return and linefeed combination (CR/LF) occurs in the text.

The print formatting commands `_alignment`, `_indent`, `_lmargin`, and `_rmargin` require `_wrap` to be *true*.

When `_wrap` is *true*, *Visual* dBASE stores streaming output in a buffer until it finishes displaying or printing the current line. If you generate output with the `? command`, follow it with another `? command` to force the last line of text to print.

Example

```
_lmargin=5
_rmargn=15
string="Now is the time for all men and women to come to..."
_wrap=false
? string
// wrap false displays as:
// Now is the time for all men and women to come to...
_wrap=true
? string
// wrap true displays as:
//   Now is the
//   time for
//   all men and
//   women to
//   come to...
```

See Also `?, ??, _alignment, _indent, _lmargin, _ploffset, _rmargin, PRINTJOB...ENDPRINTJOB`

Extending *Visual* dBASE with DLLs, OLE and DDE

The classes and elements described in this chapter allow you to extend *Visual* dBASE to work with Dynamic Link Libraries (DLLs) and other Windows resources, as well as communicate directly with other Windows programs through both the Object Linking and Embedding (OLE) and Dynamic Data Exchange (DDE) mechanisms.

class DDELink

Initiates and controls a DDE link between *Visual* dBASE and a server application, allowing *Visual* dBASE to send instructions and data-exchange requests to the server.

Syntax [*<oRef>* =] new DDELink()

<oRef> A variable or property in which to store a reference to the newly created DDELink object.

Properties The following tables list the properties, events, and methods of the DDELink class.

| Property | Default | Description |
|------------------|---------|---|
| <i>className</i> | DDELINK | Identifies the object as an instance of the DDELink class (this property is described in Chapter 5, "Core language.") |
| <i>server</i> | | The name of the server you specified with the <i>initiate</i> () method |

| Property | Default | Description |
|----------------|---------|--|
| <i>timeout</i> | 1000 | Determines the amount of time in milliseconds that <i>Visual</i> dBASE waits for a transaction before returning an error |
| <i>topic</i> | | The name of the topic you specified with the <i>initiate()</i> method |

| Event | Parameters | Description |
|-------------------|------------------------------|--|
| <i>onNewValue</i> | <item expC>, <value expC> | When an item in the server application changes |

| Method | Parameters | Description |
|--------------------|--------------------------------|---|
| <i>advise()</i> | <item expC> | Requests that the server notify the client when the item in the server changes |
| <i>execute()</i> | <cmd expC> | Sends instructions to the server in its own language |
| <i>initiate()</i> | <server expC>, <topic expC> | Starts a conversation with a DDE server application |
| <i>peek()</i> | <item expC> | Retrieves a data item stored by the server |
| <i>poke()</i> | <item expC>, <value exp> | Sends a data item to the server |
| <i>reconnect()</i> | | Restores a DDE link that was terminated with <i>terminate()</i> |
| <i>release()</i> | | Explicitly removes the DDELink object from memory (this property is described in Chapter 5, "Core language.") |
| <i>terminate()</i> | | Terminates the link with the server application |
| <i>unadvise()</i> | <item expC> | Asks the server to stop notifying the DDELink object when an item in the server changes |

Description Use a DDELink object to open a channel of communication (known as a *DDE link*) between *Visual* dBASE and an external Windows application (known as a *server*).

You can exchange data and instructions through this link, making the two applications work together. For example, you could use a DDELink object to open, send data to, format and print a document in your word processor, or open a spreadsheet and then exchange and update table data. You can also run separate *Visual* dBASE sessions and use one as a DDE client and the other as a server (as shown in the class DDELink and class DDETopic examples).

A DDE link is established with the *initiate()* method. If the server application isn't already running, *initiate()* attempts to start it. If the link attempt is unsuccessful, *initiate()* returns *false*.

Example This sample application uses a DDE link to send data to a DDE server and optionally receive notification updates. The server program, shown in the class DDETopic example, runs in a second instance of *Visual* dBASE. For a

demonstration of multiple-client DDE interaction, create and run another form using this same code, but change the DDE topic from "TEST" to any other name. At the same time, try linking another DDE client to the *Visual* dBASE server program using some other Windows program, such as Microsoft Word (code for a Word 95 macro to do just that appears at the bottom of this topic).

```

** END HEADER -- do not remove this line
//
// Generated on 02/24/98
//
parameter bModal
local f
f = new stockclientForm()
if (bModal)
    f.mdi = false // ensure not MDI
    f.readModal()
else
    f.open()
endif

class stockclientForm of FORM
    with (this)
        onOpen = class::FORM_ONOPEN
        onClose = class::FORM_ONCLOSE
        scaleFontBold = false
        height = 6
        left = 4
        top = 2
        width = 42
        text = "Stock Client 2000"
        autoCenter = true
        mdi = false
    endwith

    this.NOTIFYTEXT = new TEXT(this)
    with (this.NOTIFYTEXT)
        height = 1
        left = 2
        top = 2
        width = 38
        border = true
        colorNormal = "BtnText"
        fontSize = 8
        text = "Welcome to Stock Client 2000"
        borderStyle = 7// Client
    endwith

    this.SHARES = new SPINBOX(this)
    with (this.SHARES)
        height = 1
        left = 2
        top = 4
        width = 12
        picture = "99999"
    endwith
endclass

```

```
class DDELink
```

```
    step = 10
    rangeMax = 10000
    rangeMin = 1
    fontSize = 8
    value = 10
    rangeRequired = true
    borderStyle = 7// Client
endwith

this.BUYBUTTON = new PUSHBUTTON(this)
with (this.BUYBUTTON)
    onClick = class::BUYBUTTON_ONCLICK
    height = 1
    left = 18
    top = 4
    width = 10
    text = "&Buy"
    fontSize = 8
    group = true
    value = false
endwith

this.SELLBUTTON = new PUSHBUTTON(this)
with (this.SELLBUTTON)
    onClick = class::SELLBUTTON_ONCLICK
    height = 1
    left = 30
    top = 4
    width = 10
    text = "&Sell"
    fontSize = 8
    group = true
    value = false
endwith

this.NOTIFYBUYCHECKBOX = new CHECKBOX(this)
with (this.NOTIFYBUYCHECKBOX)
    onChange = class::NOTIFYBUYCHECKBOX_ONCHANGE
    height = 0.8636
    left = 2
    top = 1
    width = 16
    text = "Notify on Buy"
    colorNormal = "WindowText/BtnFace"
    fontSize = 8
    value = false
    group = true
endwith

this.NOTIFYSELLCHECKBOX = new CHECKBOX(this)
with (this.NOTIFYSELLCHECKBOX)
    onChange = class::NOTIFYSELLCHECKBOX_ONCHANGE
    height = 0.8636
    left = 24
```



```

top = 1
width = 16
text = "Notify on Sell"
colorNormal = "WindowText/BtnFace"
fontSize = 8
value = false
group = true
endwith

function BUYBUTTON_onClick
    form.ddeClientObj.poke( "Buy", "" + form.shares.value )
    return

function form_onClose
    form.ddeClientObj.terminate()
    form.ddeClientObj.release() // Destroys parent reference to form
    return

function form_onOpen
    this.ddeClientObj = new StockDDELink()
    with this.ddeClientObj
        if initiate( "STOCKSERVER", "TEST" )
            parent = this
            timeout = 2000
            msgbox( "Connecting to " + server ;
                + ". Account holder: " + topic ;
                + ". Current holdings: " + peek( "Buy" ),,
                "Stock Client 2000")
        else
            msgbox( "Could not connect to StockServer", ;
                "Connection failed", 16 )
            form.close()
        endif
    endwith

function NOTIFYBUYCHECKBOX_onChange
    if this.value
        form.ddeClientObj.advise( "Buy" )
    else
        form.ddeClientObj.unAdvise( "Buy" )
    endif
    return

function NOTIFYSELLCHECKBOX_onChange
    if this.value
        form.ddeClientObj.advise( "Sell" )
    else
        form.ddeClientObj.unAdvise( "Sell" )
    endif
    return

function SELLBUTTON_onClick
    form.ddeClientObj.poke( "Sell", "" + form.shares.value )
    return

```

```
class DDETopic
```

```
endclass
```

```
class StockDDELink of DDELink  
  this.parent = null
```

```
  function onNewValue( item, value )  
    this.parent.notifyText.text := "Notified of changes in: " + item + ;  
                                   " now: " + value
```

```
endclass
```

This Word for Windows 95 macro also lets you communicate with the *Visual* dBASE DDE stock server program described in the class DDETopic example. You can use this macro from within Word at the same time as you use the *Visual* dBASE DDE client application described above to communicate with the *Visual* dBASE server running in a separate instance.

```
'Word 95 macro to communicate with VdB Stock Server dde sample  
Sub MAIN  
  DDETerminateAll  
  channel = DDEInitiate("STOCKSERVER", "TEST")  
  DDEPoke channel, "sell", "150"  
  DDETerminate channel  
End Sub
```

See Also class DDETopic, class OleAutoClient

class DDETopic

Determines the actions taken when *Visual* dBASE receives requests from a DDE client.

Syntax [*<oRef>* =] new DDETopic(*<topic expC>*)

<oRef> A variable or property in which to store a reference to the newly created DDETopic object. This object reference must be returned by the *_app* object's *onInitiate* event handler.

<topic expC> The name of the topic to which the DDETopic object responds.

Properties The following tables list the properties, events, and methods of the DDETopic class.

| Property | Default | Description |
|------------------|----------|--|
| <i>className</i> | DDETOPIC | Identifies the object as an instance of the DDETopic class (this property is described in Chapter 5, "Core language.") |
| <i>topic</i> | | The DDETopic object's topic |

| Event | Parameters | Description |
|-------------------|------------------------------|--|
| <i>onAdvise</i> | <item expC> | After an external application creates a hot link |
| <i>onExecute</i> | <cmd expC> | When a client application sends a command to <i>Visual</i> dBASE |
| <i>onPeek</i> | <item expC> | When the client requests a value from <i>Visual</i> dBASE |
| <i>onPoke</i> | <item expC>, <value expC> | When the client sends a new value for a <i>Visual</i> dBASE item |
| <i>onUnadvise</i> | <item expC> | After a client removes a hot link from a particular item |

| Method | Parameters | Description |
|--------------------|-------------|--|
| <i>notify</i> () | <item expC> | Notifies all interested client applications that a <i>Visual</i> dBASE item was changed |
| <i>release</i> () | | Explicitly removes the DDETopic object from memory (this property is described in Chapter 5, "Core language.") |

Description Use a DDETopic object to determine what *Visual* dBASE does for a client application when *Visual* dBASE is the server in a DDE link. *Visual* dBASE may act as a DDE server for one or more *topics*; client applications must specify the topic they are interested in when they create the DDE link. (A client application may link to more than one topic at a time.)

As a server application, *Visual* dBASE accepts either a generic command string, or a named item-value pair from a client application. It must respond to requests for a named data item, and notify interested client applications when a item value changes.

You usually create a DDETopic object in an initiation-handler routine, which you assign to the *onInitiate* event of the *_app* object. The initiation-handler executes when a client application requests a DDE link with *Visual* dBASE and no DDETopic object exists in memory for the desired topic. Each newly-created DDETopic object must be returned by the *onInitiate* event handler; that object is automatically stored internally to respond to client requests on that topic.

When *Visual* dBASE acts as a DDE server, the topics and item names it maintains internally are not case-sensitive; topic and item names that match existing names (regardless of case) will be changed to those names before being passed to events. When writing event handlers, be aware that the names may vary in case—use UPPER() or LOWER() to make the names consistent in the program logic.

For information on using *Visual* dBASE as a client application, see class DDELink.

Example The following program creates a DDE service in an instance of *Visual* dBASE. This server handles information passed to it from multiple DDE clients, including those set up in the *Visual* dBASE application (using a separate *Visual* dBASE instance) and Microsoft Word macro shown in the the class DDELink example.

class DDETopic

```
set procedure to program(1) additive

_app.ddeServiceName = "STOCKSERVER"
_app.onInitiate = InitStockDDE
_app.framewin.text = "Stock Server 2000"
? "Ready for DDE server requests for:", _app.ddeServiceName

function InitStockDDE(newTopic)
    // DDETopic object not in memory, create it
    local x
    x = new StockDDETopic(newTopic)
    ? "Initialized server for topic:", newTopic
    return x // Must return requested topic

class StockDDETopic(t) of DDETopic(t)
    this.adviseList = new AssocArray()
    this.value      = 100

    function onAdvise(item)
        ? this.topic, "ADVISE on changes in:", item
        this.adviseList[ upper( item ) ] = null // Create element with dummy value

    function onExecute(cmd)
        ? this.topic, "EXECUTE:", cmd
        // This server does not support any commands

    function onPeek(item)
        ? this.topic, "PEEK:", item
        return this.value

    function onPoke(item /* Buy or Sell */, value /* Shares */)
        ? this.topic, "POKE:", item, value
        if upper(item) = "SELL"
            this.value -= val( value )
        elseif upper(item) = "BUY"
            this.value += val( value )
        endif
        // Notify clients if item in adviseList
        if this.adviseList.isKey( upper( item ) )
            ? this.topic, "NOTIFY change in:", item
            this.notify( upper( item ) ) // Implicitly calls onPeek to get value
        endif
        ? "Value is now:", this.value

    function onUnadvise(item)
        if this.adviseList.isKey( upper( item ) )
            ? this.topic, "cancel ADVISE on changes in:", item
            this.adviseList.removeKey( upper( item ) )
        endif

endclass
```

See Also class DDELink

class OleAutoClient

Creates an OLE2 controller that attaches to an OLE2 server.

Syntax [*oRef* =] new OleAutoClient(<server expC>)

<oRef> A variable or property in which you want to store a reference to the newly created OleAutoClient object.

<server expC> The name of the OLE Automation server. The name is of the form, "app.object"; for example, "word.application"

Properties The properties, events, and methods of each instance of the OleAutoClient class depend on the attached OLE automation server.

Description OLE automation allows you to control another application, an OLE automation server, through an OLE automation client. For example, with a full-featured word processor as an OLE automation server, you could do the following all on the server machine:

- Start the word processor
- Open an order form
- Fill in data that was entered in a client browser
- Fax that order form to a customer
- Close the word processor

With *Visual* dBASE as the host for the OLE automation client, you could control the entire process from a browser. You don't even need the word processor to be installed on the *Visual* dBASE client, just on the *Visual* dBASE server machine.

Visual dBASE's dynamic object model is a natural host for OLE automation clients. Because there is no need to declare the capabilities of the OLE automation server as you would with a statically linked language, you can specify any OLE Automation server at run time, and use whatever capabilities it has.

Once you create the OleAutoClient object, the properties, events, and methods the OLE automation server provides are accessed through the OleAutoClient object, just as with stock *Visual* dBASE objects. You can also *inspect()* the OleAutoClient object's properties.

Example These examples show how to use OleAutoClient to gather a list of Microsoft Word "Normal" template macros and run a Word macro from *Visual* dBASE. Examples are provided for both Word 95 and Word 97.

Word 95 example 1:

```
// Assign a new Word.Basic object.
// If Word is already running, the object uses the running instance.
// If Word is not running, an instance is created, but hidden.
// To make the instance visible, use the Word AppShow() command.
```

class OleAutoClient

```
w = new OleAutoClient("word.basic")
w.AppShow()

// Clear the Command window,
// list all macros and descriptions in the Normal template
clear
for m = 1 to w.countmacros(0)
    n = w.macroname(m, 0)
    ? n
    ? w.macrodsc(n)
next count

// to run a macro named "Macro1"
w.toolsmacro("Macro1",true)

// Dismiss the object. If Word was already open when this routine was run, it remains
open but releases the object. If you opened Word with this routine, the object is
released and Word is removed from the task list. If you also made Word visible with
AppShow(), the app is closed.
w= ""
```

Word 95 example 2. Scenario: A form offers lists containing two corresponding associative arrays of Word documents and available printers. When the user selects a document and a printer, a single button opens the selected Word document, shows Word's printer set up dialog, prints the document and then closes the document.

```
function printButton_onClick()
    local w
    w = new OleAutoClient( "word.basic" )
    w.FileOpen( this.form.aDocs[ this.form.docSelect.value ] )
    w.FilePrintSetup( this.form.aPrinter[ this.form.printerSelect.value ] )
    w.FilePrint()
    w.FileClose()
```

Word 97 example (equivalent to 95 example 1):

```
// Assign a new Word application object.
// If Word is already running, the object uses the running instance.
// If Word is not running, an instance is created, but hidden.
// To show it, you can use Word's Visible property.

w = new OleAutoClient("Word.Application.8")
w.Visible = true

// Clear the Command window,
// list the names of all macros (grouped into VBComponents in Word 97)
// in the Normal template.
clear
for i = 1 to w.NormalTemplate.VBProject.VBComponents.Count
    ? w.NormalTemplate.VBProject.VBComponents.Item(i).Name
next i
w.Application.Run("Macro1")
w.Application.Quit()
```

advise()

Creates a DDE hot link to enable the passing of notification messages whenever a specified topic item changes in the server application.

Syntax <oRef>.advise(<item expC>)

<oRef> A reference to the DDELink object that wants to get notified.

<item expC> The name of the topic item you want to monitor.

Property of DDELink

Description Use the *advise()* method to create a *hot link* to an item in a server topic. A hot link requires the server application to notify *Visual dBASE* when the value of a specified item changes.

A server topic can be anything that the server application understands, but it is most often a document, spreadsheet, or database that you specified when you called the *initiate()* method to create the DDE link with the external application. If, for example, you use *initiate()* to open a spreadsheet application as the server application and a worksheet as a topic, you could specify any item or range in the worksheet as the topic item you want monitored for changes.

The DDELink object's *onNewValue* event will fire with the item name and new value as parameters whenever the named item changes.

Example See class DDELink.

See Also *initiate()*, *onNewValue*, *unadvise()*

execute()

Sends a command string to a DDE server application in its own language.

Syntax <oRef>.execute(<cmd expC>)

<oRef> A reference to the DDELink object that has the link.

<cmd expC> Command or macro to send to the DDE server application.

Property of DDELink

Description Use the *execute()* method to send commands to DDE server applications.

The command string must be in the language of the server application, or any other string that the server expects.

Be sure to enclose commands in the delimiters required by the server application. For example, Quattro Pro commands must be enclosed in braces

{ }), while Word for Windows 95 commands are enclosed in brackets ([]). Some applications accept multiple commands separated by brackets. For information, consult your DDE server documentation.

Before you can send a command string to a server, you must open the server application, open the document, and establish a DDE link. For information on establishing DDE links, see *initiate()*.

Example This example opens a Word for Windows 95 document, prints it, and closes the connection.

```
w = new ddelink()
w.initiate("winword", "c:\my documents\myletter.doc")
w.execute(" [FILEPRINT] ")
w.terminate()
w = " "
```

See Also *initiate()*, *poke()*

EXTERN

Declares a prototype for a non-*Visual* dBASE function contained in a DLL file.

Syntax EXTERN [CDECL | PASCAL | STDCALL] <return type> <function name>
 ([<parameter type> [, <parameter type> ...]])
 <filename>

or

EXTERN [CDECL | PASCAL | STDCALL] <return type> <user-defined function name>
 ([<parameter type> [, <parameter type> ...]])
 <filename>
 FROM <export function name expC> | <ordinal number expN>

Because you create a function prototype with EXTERN, parentheses are required as with other functions.

CDECL | PASCAL | STDCALL Sets the function calling convention. The default is STDCALL.

<function name> The export name of the function. The export name of an external function is contained in the DEF file associated with the DLL file that holds the function, or explicitly exported in the source code.

Note With STDCALL and CDECL, function names are case-sensitive.

<return type> and <parameter type> A keyword representing the data type of the value returned by the function, and the data type of each argument you send to the function, respectively. The following tables list the keywords you can use.

Parameters or return values

| Keyword | as pointer | <i>Visual</i> dBASE data type | Data type size |
|----------------|-------------------|--------------------------------------|-----------------------|
| CINT | CPTR CINT | Numeric | 4 bytes (32 bits) |
| CLONG | CPTR CLONG | Numeric | 4 bytes (32 bits) |
| CSHORT | CPTR CSHORT | Numeric | 2 bytes (16 bits) |
| CCHAR | | Numeric | 1 byte (8 bits) |
| | CSTRING | String | Null-terminated |
| CHANDLE | | Numeric | 4 bytes (32 bits) |
| CUINT | CPTR CUINT | Numeric | 4 bytes (32 bits) |
| CULONG | CPTR CULONG | Numeric | 4 bytes (32 bits) |
| CUSHORT | CPTR CUSHORT | Numeric | 2 bytes (16 bits) |
| CUCHAR | | Numeric | 1 byte (8 bits) |
| CFLOAT | CPTR CFLOAT | Numeric | 4 bytes (32 bits) |
| CDOUBLE | CPTR CDOUBLE | Numeric | 8 bytes (64 bits) |
| CLDOUBLE | CPTR CLDOUBLE | Numeric | 10 bytes (80 bits) |
| CLOGICAL | CPTR CLOGICAL | Logical | 1 byte (8 bits) |
| CVOID | | none | N/A |

Parameters only

| Keyword | as pointer | <i>Visual</i> dBASE data type | Data type size |
|----------------|-------------------|--------------------------------------|-----------------------|
| | CPTR | String | |
| ... | | N/A | |

In most cases, if the function expects a pointer as a parameter, *Visual* dBASE will pass a pointer to the value. If a function returns a pointer, *Visual* dBASE will get the value at the pointer and convert it into the appropriate *Visual* dBASE data type.

CCHAR is actually a numeric data type, representing a single-byte value. When passing a CCHAR parameter, you may pass a string; *Visual* dBASE sends the ASCII value of the first character in the string. The return value is always a number.

If the function has no parameters or returns no value, declare the data type as CVOID.

You may use the ... parameter declaration if the calling convention is STDCALL to designate a variable number of parameters.

Using strings *Visual* dBASE is a Unicode application; using strings is more complicated in 32-bit programming than in 16-bit programming. Many Windows API functions have both an A (ANSI) and W (wide-character) version. The A functions use single-byte characters, and the W versions use

double-byte characters. When calling the A version of a function, always use CSTRING. When calling the W version of a function, always use CPTR. When you use CSTRING, *Visual* dBASE will convert the Unicode string it uses into an ANSI string when passing it to the function. After the function call, it converts the ANSI string back into Unicode. With CPTR, no conversion takes place.

When strings (CSTRING or CPTR) are used as parameters, they are always passed and read back by length. You may include null characters in the string. When a return value is declared as CSTRING, it is read as null-terminated. You cannot use CPTR as a return value.

Using structures If the function expects a pointer to a structure, declare the parameter as CPTR. Use a string to represent the structure contents. Because *Visual* dBASE strings are Unicode, you must use the String object's *getByte()* and *setByte()* methods to read and write the structure, byte-by-byte.

<filename> The name of the DLL file in which the external function is stored. If the extension is omitted, the default is DLL. The file name of any DLL that you load in memory must be unique; for example, you can't load SCRIPT.DLL and SCRIPT.FON into memory concurrently, even though they have different file-name extensions.

If the DLL file is not already loaded into memory, EXTERN loads it automatically. If the DLL file is already in memory, EXTERN increments the reference counter. Therefore, it isn't necessary to execute LOAD DLL before using EXTERN.

The reference counter is incremented only the first time, regardless of how many times you execute the LOAD DLL and EXTERN statements.

You may include a path in <filename>. If you omit the path, *Visual* dBASE looks in the following directories for the DLL by default:

- 1 The directory containing VDB.EXE, or the directory in which the .EXE file of your compiled application is located.
- 2 The current directory.
- 3 The 32-bit Windows system directory (for example, C:\WIN95\SYSTEM).
- 4 The 16-bit Windows system directory, if present (for example, C:\WINDOWS\SYSTEM).
- 5 The Windows directory (for example, C:\WINNT)
- 6 The directories in the PATH environment variable

The path specification is necessary only when the DLL file is not in one of these directories.

<user-defined function name> The name you give to the external function instead of the export name. This is usually used to rename the A or W version of a function to the generic name. When you specify <user-defined function name> (instead of <function name>), you must use the FROM clause to identify the function in the DLL file.

FROM <export function name expC> | <ordinal number expN> Identifies the function in the DLL file specified by <filename>. <export function name expC> identifies the function by its name. <ordinal number expN> identifies the function with a number.

Description Use EXTERN to declare a prototype for an external function written in a language other than *Visual* dBASE. A prototype tells *Visual* dBASE to convert its arguments to data types the external function can use, and to convert the value returned by the external function into a data type *Visual* dBASE can use.

To call an external DLL function, first prototype it with EXTERN. Then, using the name of the function you specified with EXTERN, call the function as you would any *Visual* dBASE function. You must prototype an external function before you can call that function in *Visual* dBASE.

The external function may be in any 32-bit DLL, such as the Windows API or a third-party DLL file. Although most library code is contained in files with extensions of DLL, such code can be held in EXE files, or even in DRV or FON files.

Example Suppose you want to add the Cascade option to your menubar's Window menu. This ability is not built-in, but it can be done easily through the Windows API. To cascade the windows, you send a message to the MDI client window. To get the MDI client window, use the Windows GetParent() function. The SendMessage() function is used to send the message. You add the following to the Header of your .MNU file:

```
if type( "GetParent" ) # "FP"
    extern CHANDLE GetParent( CHANDLE ) User32
endif
if type( "SendMessage" ) # "FP"
    extern CLONG SendMessage( CHANDLE, CUINT, CWORD, CLONG ) User32 from "SendMessageA"
endif
#define WM_MDICASCADE 0x0227
```

The .MNU file is executed when you assign the file to a form's *menuFile* property, which includes the code in the Header. The two functions are prototyped with EXTERN. First, the TYPE() function is used to see if the function has already been EXTERNEd. If so, there's no need to do it again.

The GetParent() function is straightforward: it takes a window handle and returns the handle of the window's parent. A form's window handle is contained in its *hWnd* property. SendMessage() is a bit more complicated. It has both an A version and a W version. The W version does not work in Windows 95, so the A version is used. Note that the function name after the FROM is in quotes—it has to be a character or numeric expression—while the prototyped function name does not. SendMessage() takes a handle to the window, the message, and two parameters to the message. It returns a result value.

Finally, the #define preprocessor directive is used to define a manifest constant for the cascade message number. This #define may be found in the WINUSER.H file, one of the Windows header files in the \Include subdirectory.

`initiate()`

Once all the setup is done, calling the function is easy. The *onClick* handler for the Cascade menu item is the codeblock:

```
{; SendMessage( GetParent( form.hWnd ), WM_MDICASCADE, 0, 0 )}
```

The manifest constant in the codeblock is replaced at compile-time. If you examine the codeblock, you will see the number, not the manifest constant.

See Also `LOAD DLL`, `RELEASE DLL`, `TYPE()` (page 5-50)

initiate()

Starts a conversation with an external application or aliased DDE server.

Syntax `<oRef>.initiate(<server expC>, <topic expC>)`

<oRef> A reference to the DDELink object through which you want to initiate the DDE link.

<server expC> The executable filename of the server application (normally the .EXE extension isn't necessary) or the alias name of a running DDE server.

<topic expC> Name of a built-in DDE topic, document, or other topic.

Property of DDELink

Description Use *initiate()* to open a channel of communication (known as a *DDE link*) between *Visual* dBASE and a running external Windows application or aliased DDE server.

If you call an external application with *initiate()*, *Visual* dBASE tries to open the application if it is not already running. *initiate()* returns *true* if the connection is successful, and *false* if the connection attempt fails.

To close the DDE link, use *terminate()*.

Example See class DDELink.

See Also *reconnect()*, *terminate()*

LOAD DLL

Initiates a DLL file.

Syntax `LOAD DLL [<path>] <DLL filename>`

[<path>] <DLL name> The name of the DLL file. *<path>* is the directory path to the DLL file in which the external function is stored.

Description Use LOAD DLL to make the resources of a DLL file available to your application.

Note *Visual* dBASE 7 uses 32-bit DLLs only; it cannot use 16-bit DLLs.

You can also use LOAD DLL to check for the existence of a DLL file. For example, you can use the ON ERROR command to execute an error trapping routine each time the LOAD DLL command can't find a specified DLL file.

LOAD DLL does not use the *Visual* dBASE path to find DLL files. Instead, it searches the following directories:

- 1 The directory containing VDB.EXE, or the directory in which the .EXE file of your compiled application is located.
- 2 The current directory.
- 3 The 32-bit Windows system directory (for example, C:\WIN95\SYSTEM).
- 4 The 16-bit Windows system directory, if present (for example, C:\WINDOWS\SYSTEM).
- 5 The Windows directory (for example, C:\WINNT)
- 6 The directories in the PATH environment variable

A DLL file is a precompiled library of external routines written in non-*Visual* dBASE languages such as C and Pascal. A DLL file can have any extension, although most have extensions of .DLL.

When you initialize a DLL file with LOAD DLL, *Visual* dBASE can access its resources; however, it doesn't become resident in memory until your program or another Windows program uses its resources.

To access a DLL function, create a *Visual* dBASE function prototype with EXTERN. Then, using the name you specified with EXTERN, call the routine as you would any *Visual* dBASE function.

Example The following example uses LOAD DLL to initialize an image resource from a .DLL file:

```
load dll MyPicts.dll
define form Pics from 2,2 TO 20,40
define image MyPict of Pics;
    property dataSource "resource MyPicts.dll 1001", top 5, left 5
open form Pics
```

See Also EXTERN, RELEASE DLL

notify()

Notifies all interested client applications that a *Visual* dBASE item was changed.

Syntax <oRef>.notify(<item expC>)

<oRel> A reference to the DDETopic object in which the item changed.

<item expC> Name of the item that has changed.

Property of DDETopic

Description Use *notify()* in a DDE server program to tell all interested client applications that an item in the *Visual* dBASE server was changed.

Client applications ask to be notified of changes by calling their equivalent of the DDELink object's *advise()* method. *Visual* dBASE automatically maintains an internal list of these clients so that when the *notify()* method is called, the appropriate DDE message is sent to each client, if any. For a *Visual* dBASE client, that message fires the *onNewValue* event.

onPoke event handlers often call *notify()* when an external application sends *Visual* dBASE a *poke* request. For example, a Quattro Pro data-exchange application might use its {POKE} command to send *Visual* dBASE a value, causing the *onPoke* event handler to execute. The *onPoke* routine could insert the value into a field, then execute *notify()* to inform Quattro Pro that the field changed.

When *notify()* is called, the *onPeek* method is called implicitly to get the value of the item to pass to the client applications.

Example See class DDETopic.

See Also *advise()*, *onNewValue*, *onPeek*, *unadvise()*

onAdvise

Event fired after an external application requests a DDE hot link to an item in a *Visual* dBASE server topic.

Parameters **<item expC>** The data item for which the external application wants to be advised when changes are made.

Property of DDETopic

Description Use *onAdvise* in a DDE server program to respond to a request for a hot link and to determine which *Visual* dBASE data item the link applies to. To implement a hot link in a DDETopic object, use the *notify()* method to advise all interested clients whenever the item changes.

Visual dBASE automatically maintains an internal list of all clients that asked to be notified on changes in a particular item (when the client calls their equivalent of the DDELink object's *advise()* method). This makes *onAdvise* supplemental. For example, you can track those items for which there has been a notification request. Whenever an item changes, you can call *notify()* only if someone has requested notification.

Item names are often held in tables or arrays to handle multiple hot links. For example, a client application might request a hot link to a field, passing the field name through the *<item>* parameter. Each time, the *onAdvise* event handler places the field name in an array.

The *onPoke* event handler would search this array each time a field is changed; if the name of the changed field is found in the array, the routine could call the *notify()* method.

Example See class DDETopic.

See Also *advise()*, *notify()*, *onUnadvise*, *unadvise()*

onExecute

Event fired when a client application sends a command string to a DDE server program.

Parameters *<cmd expC>* The command string sent by the client application.

Property of DDETopic

Description Use the *onExecute* property to perform an action when the client application sends a directive to *Visual* dBASE. This directive can be any string of characters. For example, a stock trading routine might receive either of two character strings, "BUY" or "SELL". The routine could use an IF statement to perform one action or another accordingly. A web browser could take a URL as a command, and display that URL.

Example See class DDETopic.

See Also *execute()*, *onPeek*, *onPoke*

onNewValue

Event fired when a hot-linked item in a DDE server changes.

Parameters *<item expC>* Identifies the server item that was changed.
<value expC> The new value of the hot-linked server item.

Property of DDELink

Description Use *onNewValue* to perform an action when a hot-linked server item is changed. A hot link, which you create with the *advise()* method, tells the server to notify *Visual* dBASE when the item changes.

Note that the value of the item is always converted to a string.

Example See class DDELink.

See Also *advise()*, *unadvise()*

onPeek

Event fired when a client application tries to read an item from a DDE server application.

Parameters *<item expC>* The data item the client application wants to read.

Property of DDETopic

Description Use the *onPeek* property to send a value to a client application when the client application makes a *peek* request.

The *onPeek* event handler must RETURN the value of the requested item. *Visual* dBASE automatically handles the internal DDE mechanism to pass the value back to the client.

onPeek is also called implicitly by the DDETopic object's *notify()* method so that the DDE server can send an item's name and value when it is changed.

Example See class DDETopic.

See Also *onPoke*, *peek()*

onPoke

Event fired when a client application attempts to send a value for a DDE server item.

Parameters *<item expC>* The name of the item that identifies the value.

<value expC> The desired value

Property of DDETopic

Description Use the *onPoke* event to receive a named value from a client application. For example, the *<item expC>* could identify a cell in a spreadsheet, and the *<value exp>* is the value to store in that cell. Another example would be for the *<item expC>* to contain a command, and the *<value exp>* to act as a parameter to that command.

Note If a client established a hot link before sending the data, you should execute the *notify()* method in the *onPoke* event handler, thus informing the client application(s) that a change occurred.

Example See class DDETopic.

See Also *advise()*, *notify()*, *onExecute*, *onPeek*, *poke()*

onUnadvise

Event fired after *Visual* dBASE is requested to stop notifying the client application when a *Visual* dBASE data item changes.

Parameters **<item expC>** The data item for which the external application no longer wants to be advised when changes are made.

Property of DDETopic

Description Use *onUnadvise* in a DDE server program to respond to a request to cancel a hot link.

Visual dBASE automatically maintains an internal list of all clients that asked to be notified on changes in a particular item, and will automatically remove a client when the client calls their equivalent of the DDELink object's *unadvise()* method. This makes *onUnadvise* supplemental. If you have been doing your own tracking in an *onAdvise* event, you will want to undo that in the *onUnadvise* event handler.

Example See class DDETopic.

See Also *advise()*, *onAdvise*, *unadvise()*

peek()

Retrieves a data item from a DDE server.

Syntax *<oRef>.peek(<item expC>)*

<oRef> A reference to the DDELink object that has the link.

<item expC> The name of the desired item.

Property of DDELink

Description Use the *peek()* method to read data from a DDE server topic.

peek() takes a data item in the server topic. This item can be any single element, such as a field in a table or a cell in a spreadsheet. For example, you can read

cell C2 of Page A in a Quattro Pro spreadsheet file by passing the *<item>* parameter "A:C2".

Example See class DDELink.

See Also *poke()*

PLAY SOUND

Plays a sound stored in a .WAV file or a binary field.

Syntax PLAY SOUND FILENAME *<filename>* | ? | *<filename skeleton>* |
or
PLAY SOUND BINARY *<binary field>*

FILENAME *<filename>* | ? | *<filename skeleton>* |
or BINARY *<binary field>* Specifies the sound file or binary field. PLAY SOUND FILENAME ? and PLAY SOUND FILENAME *<filename skeleton>* display a search dialog to let you select a sound file (the .WAV extension is assumed, but you can specify otherwise). PLAY SOUND BINARY *<binary field>* plays the sound stored in a binary field.

Description Use PLAY SOUND in your programs to run .WAV files or audio data stored in binary fields.

Example This example includes a button that lets the user play stored audio data:

```
local f
f = new pictures ()
f.Open()

class pictures of form
with (this)
  escExit = true
  view = "pictures.sql"
  this.colorNormal = "BG/B"
  this.text = "Pictures Form"
  this.width = 76.00
  this.top = 0.00
  this.left = 0.00
  this.height = 30.00
  this.minimize = false
  this.maximize = false
  this.onOpen = {;create session}

define pushbutton sound of this;
property;
  onClick {;play sound binary pictures->sound},;
  text "Sound",;
  width 18.00,;
  top 5.00,;
```

```

left          1.00;;
height        3.00;;
fontSize      16.00;;
fontName "Courier"
// Additional object definitions
endclass

```

See Also SET PATH TO

poke()

Sends data to a DDE server.

Syntax <oRef>.poke(<item expC>, <value expC>)

<oRef> A reference to the DDELink object that has the link.

<item expC> The name of the desired item.

<value expC> The value you want to send (as a string).

Property of DDELink

Description Use the *poke()* method to write data to a DDE server application.

For example, a data-exchange program could start a session in Quattro Pro, open one of its spreadsheet files, and use *poke()* to write a value into one of its cells.

Example See class DDELink.

See Also *execute()*, *peek()*

reconnect()

Attempts to restart a terminated conversation with a DDE server application.

Syntax <oRef>.reconnect()

<oRef> A reference to the DDELink object that had the link.

Property of DDELink

Description Use *reconnect()* to restore a DDE link that was terminated with the *terminate()* method. It returns *true* if successful; *false* if not.

When you terminate a DDE link with *terminate()*, you can restore it with *reconnect()*. When you terminate a link with the *release()* method, however, the link can't be restored and must be recreated with another DDELink object.

Example

```
// Create DDeLink as property of _app object so that it persists
_app.stockDdeLink = new DdeLink()
if _app.stockDdeLink.initiate("STOCKSERVER","StockHolder")
    // Subsequent program operations completed; link
    // no longer required
    _app.stockDdeLink.terminate()
else
    // Link failed; display error
    msgbox( "Cannot link to STOCKSERVER", "DDE Connection failed", 16 )
endif

// Later in program, DDE link object needed again
if _app.stockDdeLink.reconnect()
    // Link OK
else
    // Link failed; display error
    msgbox( "Cannot link to STOCKSERVER", "DDE Connection failed", 16 )
endif
```

See Also *initiate(), release(), terminate()*

RELEASE DLL

Deactivates DLL files.

Syntax RELEASE DLL <DLL filename list>

Description Use RELEASE DLL when you debug a DLL file or a *Visual* dBASE application. For example, you must deactivate a DLL file and activate it again each time you change one of its routines.

A DLL file is a precompiled library of external routines written in non-*Visual* dBASE languages such as C and Pascal. A DLL file can have any extension, although most have extensions of .DLL. You activate a DLL file with the EXTERN or LOAD DLL commands.

Example The following example demonstrates the command sequence for using RELEASE DLL as a trouble shooting tool:

```
load dll mydll.dll
// ... test DLL operation
release dll mydll.dll
// ... change .DLL or C program
load dll mydll.dll
// ... test again
release dll mydll.dll
```

See Also EXTERN, LOAD DLL

RESOURCE()

Returns a character string from a DLL file.

Syntax RESOURCE(<resource id>, <DLL filename expC>)

<resource id> A numeric value that identifies the character string resource.

<DLL filename expC> The name of the DLL file.

Description Use RESOURCE() to generate a character string from a resource in a DLL file. The character string must be less than 32K; a character string longer than this is truncated.

RESOURCE() is often useful for internationalizing applications without changing program code. For example, you can store in a DLL file all character strings that might need translation from one language to another, and your application can retrieve them at run time with the RESOURCE() function. To modify the application for another language, translate the strings and store them in a new DLL file, in the same order and with the same resource IDs as their counterparts in the original DLL file. Then substitute the new DLL for the original one.

Example This example shows how RESOURCE() can be used to change languages for international development:

```
#define ENGLISH 1
#define SPANISH 2
apLanguage = SPANISH
myGreeting = resource(SPANISH, "greeting.dll")
clear
? myGreeting
```

See Also EXTERN, LOAD DLL, RELEASE DLL

RESTORE IMAGE

Displays an image stored in a file or a binary field.

Syntax RESTORE IMAGE FROM
 <filename> | ? | <filename skeleton> | BINARY <binary field>
 [TIMEOUT <expN>]
 [TO PRINTER]
 [[TYPE] <file type>]

FROM <filename> | ? | <filename skeleton> | BINARY <binary field> Identifies the file or binary field to restore the image from. RESTORE IMAGE FROM ? and RESTORE IMAGE FROM <filename skeleton> display the Open Source File dialog box, which lets the user select a file. <filename> is the name of an image file; RESTORE IMAGE assumes a .BMP extension and file type unless you

specify otherwise. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. RESTORE IMAGE FROM BINARY <binary field> displays the image stored in a binary field. You store an image in a binary field with the REPLACE BINARY command.

TIMEOUT <expN> Specifies the number of seconds the image is displayed onscreen.

TO PRINTER Sends the image to the printer as well as to the screen.

[TYPE] <file type> Specifies a bitmap image format, and assumes a .BMP format and filename extension if none is given. The word TYPE is optional. See class Image for a list of image formats supported by *Visual* dBASE.

Description Use RESTORE IMAGE to display a bitmap image that was generated and saved in any of the supported image file formats. The image is displayed in a window.

Example The following example defines a form and list box for selection of an aircraft model and uses RESTORE IMAGE to display a graphic from the memo field Image of the selected record:

```
use AIRCRDB order AIRCRAFT
select AIRCRDB
define form AC
    property
        top 5,
        left 2,
        height 13,
        width 30,
        text "Aircraft",
        sizeable true
define listbox Model OF AC
    property
        top 1,
        left 2,
        height 11,
        width 26,
        dataSource "field Aircrdb->Aircraft" ,
        onSelChange {; restore image from binary image timeout 10 }
AC.open()
```

See Also DEFINE, REPLACE BINARY

server

Holds the name of a DDE server application.

Property of DDELink

Description The read-only *server* property contains the DDE service name of server application that you established a DDE link to.

See Also *initiate(), topic*

terminate()

Terminates a conversation with a DDE server application.

Syntax <oRef>.terminate()

<oRef> A reference to the DDELink object whose connection to terminate.

Property of DDELink

Description Use *terminate()* to close a DDE link between *Visual* dBASE and a server application.

terminate() stops communication between *Visual* dBASE and the server application, but doesn't close the server application itself.

When you terminate a DDE link with *terminate()*, you can restore it with *reconnect*. When you terminate the link with the *release()* method, the link can't be restored and you need to create another DDELink object again.

Example See class DDELink.

See Also *initiate(), reconnect()*

timeout

Specifies the amount of time in milliseconds that *Visual* dBASE waits on a transaction before returning an error.

Property of DDELink

Description Use *timeout* to set a limit on the length of time *Visual* dBASE waits for a DDE transaction to complete successfully.

Errors sometimes occur when *Visual* dBASE tries to exchange data with a server application or send instructions to it. Each time an attempt is made, *Visual* dBASE waits for the amount of time you specify (in milliseconds) with *timeout*. When the transaction fails to complete in the allotted time, *Visual* dBASE generates an error.

When using DDE over a network, you may need to set *timeout* to a larger value.

Example See class DDELink.

See Also *advise()*, *execute()*, *initiate()*, *peek()*, *poke()*, *unadvise()*

topic

Holds the name of the topic of a DDELink or DDETopic object.

Property of DDELink, DDETopic

Description The *topic* property of a DDELink object contains the name of the topic for which you established a DDE link.

The *topic* property of a DDETopic object distinguishes the object from other DDETopic objects. For example, a *Visual* dBASE server application might create two DDETopic objects, one with a topic of NASDAQ and the other with a topic of AMEX:

```
xServer1 = new DdeTopic("NASDAQ")
xServer2 = new DdeTopic("AMEX")
```

The *topic* property is read-only.

See Also *initiate()*, *server*

unadvise()

Asks the server to stop notifying the client when an item in the server document changes.

Syntax <oRef>.unadvise(<item expC>)

<oRef> A reference to the DDELink object that no longer wants notification.

<item expC> The name of a topic item previously hot-linked using *advise()*.

Property of DDELink

Description Use the *unadvise()* method to disconnect a hot link to a DDE server item. A hot link, which you create with the *advise()* method, provides a way for the server to inform the client when a specified item, such as a field or a spreadsheet cell, has changed.

Example See class DDELink.

See Also *advise()*

Chapter 20

IDE

This section of the *Language Reference* describes language elements that you use within the *Visual dBASE* integrated development environment (IDE) to programmatically create, modify, compile and build applications.

BUILD

Creates a Windows executable file (.EXE) from your *Visual dBASE* object files and resources.

Syntax BUILD FROM <project or response file name>
or
BUILD <filename list> [ICON <icon filename>] [SPLASH <bitmap filename>] [TO <executable filename>]

FROM <project or response file name> Name of a *Visual dBASE* project or response file that contains the names of all object files and resources that are to be linked into your executable. If no extension is provided, .PRJ is assumed.

<filename list> List of compiled program elements, separated by commas. If you provide a filename without an extension, .PRO (compiled program) is assumed.

ICON <icon filename> Optional icon (.ICO) file used to identify your program in the Windows environment (e.g., when minimized or listed in the Windows Explorer or a program group).

SPLASH <bmp format filename> Optional bitmap (.BMP) file that displays while your program loads.

TO <executable filename> The name of the Windows executable file (.EXE) to create. If not specified, the base file name of the named project or response file (or the first file name in <filename list>) is used.

CLEAR ALL

Description Use the BUILD command to link compiled *Visual* dBASE program elements and supporting resources (such as bitmaps and icons) into a Windows executable (.EXE) file.

Though the new project file format is the default for build specifications, support for response (.RSP) files is offered for backward compatibility.

See Also COMPILE, MODIFY PROJECT

CLEAR ALL

Releases all user-defined memory variables and closes all open files.

Syntax CLEAR ALL

Description CLEAR ALL combines the CLEAR MEMORY and CLOSE ALL commands, releasing all user-defined memory variables, closing all open tables in the current workset, and all other files. For more information, see CLEAR MEMORY and CLOSE ALL.

Note CLEAR ALL does not explicitly release objects. However, if the only reference to an object is in a variable, releasing the variable with CLEAR ALL in turn releases the object.

Use CLEAR ALL during development to clear all variables (and any objects that rely on those references) and close all files to reset your working environment. Because of the event-driven nature of *Visual* dBASE, CLEAR ALL is generally not used in programs.

See also CLEAR MEMORY (page 5-7), CLOSE ALL

CLOSE ALL

Closes (almost) all open files.

Syntax CLOSE ALL

Description CLOSE ALL closes almost all open files, including:

- All databases opened by the Navigator and with OPEN DATABASE
- All tables opened (with USE) in all work areas in the current workset
- All files opened with low-level file functions, or a File object
- All procedure and library files opened with SET PROCEDURE and SET LIBRARY
- Any text streaming file opened by SET ALTERNATE

It does not close:

- The printer file specified by the SET PRINTER TO command

- Tables or databases opened through the data access objects

Use CLOSE ALL during development to close all files, resetting your working environment, without affecting any variables. To close all files and release all variables, use CLEAR ALL. Because of the event-driven nature of *Visual* dBASE, CLOSE ALL is generally not used in programs.

See Also CLEAR ALL, CLOSE ALTERNATE, CLOSE DATABASES, CLOSE PRINTER, CLOSE PROCEDURE, CLOSE TABLES, CREATE SESSION

COMPILE

Compiles program files (.PRG, .WFM), creating object code files (.PRO, .WFO).

Syntax COMPILE <filename 1> | <filename skeleton>
[AUTO]
[LOG <filename 2>]

<filename 1> | <filename skeleton> The file(s) to compile. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory only. If you specify a file without including its extension, *Visual* dBASE assumes .PRG.

AUTO The optional AUTO clause causes the compiler to detect automatically which files are called by your program, and to recompile those files.

LOG <filename 2> Logs the files that were compiled, and any compiler errors or warning messages to <filename 2>. The default extension for the log file is .TXT.

Description Use COMPILE to explicitly compile or recompile program files without loading or executing them. *Visual* dBASE automatically compiles program source files into object (bytecode) files when they are loaded (with SET PROCEDURE or SET LIBRARY) or executed (with DO or the call operator). The compiled object files are created in the same directory as the source code files.

The file is compiled with coverage information if SET COVERAGE is ON or the file contains the

```
#pragma coverage(on)
```

directive.

When you compile a program, *Visual* dBASE detects any syntax errors in the source file and either logs the error in the LOG file, or displays an error message corresponding to the error in a dialog box that contains three buttons:

- *Cancel* cancels compilation (equivalent to pressing *Esc*).
- *Ignore* cancels compilation of the program containing the syntax error but continues compilation of the rest of the files that match <filename skeleton> if you specified a skeleton.

- *Fix* lets you fix the error by opening the source code in an editing window, positioning the insertion point at the point where the error occurred.

See the Help for information about compiling *Visual* dBASE programs into stand-alone executable files.

See Also CLEAR PROGRAM, DO, SET COVERAGE, SET DEVELOPMENT, SET PROCEDURE

CONVERT

Adds a _dbaselock field to a table for storing multiuser lock information.

Syntax CONVERT [TO <expN>]
TO <expN> Specifies the length of the multiuser information field to add to the current table. The <expN> argument can be a number from 8 to 24, inclusive. The default is 16.

Description Use CONVERT to add a special _dbaselock field to the structure of the current table. In general, CONVERT is a one-time operation required for each table that is shared in a multi-user environment.

Use the option TO <expN> to specify the length of the field. If you issue CONVERT without the TO <expN> option, the width of the field is 16. If you want to change the length of the _dbaselock field after using CONVERT, you can issue CONVERT again on the same table. To view the contents of the _dbaselock field, use LKSYS().

Note You must use the table exclusively (USE...EXCLUSIVE) before issuing CONVERT. Any records marked as deleted will be lost during the CONVERT. The _dbaselock field contains the following values:

| | |
|-------|--|
| Count | A 2-byte hexadecimal number used by CHANGE() |
| Time | A 3-byte hexadecimal number that records the time a lock was placed |
| Date | A 3-byte hexadecimal number that records the date a lock was placed |
| Name | A 0- to 16-character representation of the login name of the user who placed a lock, if a lock is active |

The count, time, and date portions of the _dbaselock field always make up its first 8 characters. If you accept the default 16-character width of the _dbaselock field, the login name is truncated to 8 characters. If you set the field width to fewer than 16 characters, the login name is truncated the necessary amount. If you set the width of <expN> to 8 characters, the login name doesn't appear at all.

Every time a record is updated, *Visual* dBASE rewrites the count portion of `_dbaselock`. If you issue `CHANGE()`, *Visual* dBASE reads the count portion from disk and compares it to the previous value it stored in memory when the record was initially read. If the values are different, another user has changed the record, and `CHANGE()` returns *true*. For more information, see `CHANGE()`.

`LKSYS()` returns the login name, date, and time portions of the `_dbaselock` field. If you place a file lock on the table containing the `_dbaselock` field, the value in the `_dbaselock` field of the first record contains the information used by `CHANGE()` and `LKSYS()`. For more information, see `LKSYS()`.

Note `CONVERT` doesn't affect SQL databases or Paradox tables.

Example After creating the DBF table `Company`, the `CONVERT` command is used in the Command window to add the `_dbaselock` field:

```
use COMPANY exclusive
display structure           // Note structure
convert to 24              // Include 16 bytes for user name
display structure           // Note added _dbaselock field with 24 byte size
```

See Also `CHANGE()`, `FLOCK()`, `LKSYS()`, `LOCK()`, `NETWORK()`, `REINDEX`, `RLOCK()`, `SET DELETED`, `SET EXCLUSIVE`, `SET LOCK`, `SET REPROCESS`, `UNLOCK`, `USE`

CREATE

Opens the Table designer to create or modify a table interactively.

Syntax `CREATE`
`[<filename> | ? | <filename skeleton>`
`[[TYPE] PARADOX | DBASE]`
`[WIZARD | EXPERT [PROMPT]]`

<filename> | ? | <filename skeleton> The name of the table you want to create. `CREATE ?` and `CREATE <filename skeleton>` display a dialog box, in which you can specify the name of a new table. The `<filename>` follows the standard Xbase DML table naming conventions detailed on page 12-1.

If you don't specify a name, the table remains untitled until you save the file. If you specify an existing table name, *Visual* dBASE asks whether you want to overwrite it. If you reply no, nothing further happens.

[TYPE] PARADOX | DBASE Overrides the default table type set by `SET DBTYPE`. The `TYPE` keyword is included for readability only; it has no effect on the operation of the command. Specifying `PARADOX` creates a Paradox table with a `.DB` extension. Specifying `DBASE` creates a DBF table with a `.DBF` extension.

WIZARD | EXPERT [PROMPT] If the `PROMPT` clause is used, a dialog appears asking if you want to use the Table designer or the Table wizard. You can then

invoke either the designer or the wizard. The WIZARD clause without PROMPT causes the Table wizard to be invoked. You may use the keyword EXPERT instead of WIZARD.

Description CREATE opens the Table designer, an interactive environment in which you can create or modify the structure of a table, or the Table wizard, a tool that guides you through the process of creating tables. The type of table you create depends on the *<filename>* you specify, or the current database and the current setting of SET DBTYPE.

Create a table by defining the name, type, and size of each field. For more information on using the Table designer, see the *Developer's Guide*.

To modify an existing table, use the MODIFY STRUCTURE command.

See Also COPY STRUCTURE, DISPLAY STRUCTURE, LIST STRUCTURE, MODIFY STRUCTURE

CREATE COMMAND

Displays a specified program file for editing, or displays an empty editing window.

Syntax CREATE COMMAND [*<filename>* | ? | *<filename skeleton>*]

<filename>* | ? | *<filename skeleton> The file to display and edit. The ? and *<filename skeleton>* options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. If you specify a file without including its extension, *Visual* dBASE assumes .PRG. If you issue CREATE COMMAND without an option, *Visual* dBASE displays an untitled empty editing window.

Description Use CREATE COMMAND to create new or edit existing program files. Use DO to execute program files.

If you're creating a new program file, CREATE COMMAND displays an empty editing window. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY COMMAND command to edit an existing file without being asked whether you want to modify it.

By default, CREATE COMMAND launches the *Visual* dBASE Source Editor. You can specify an alternate editor by using the SET EDITOR command or by changing the EDITOR setting in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the EDITOR parameter directly in VDB.INI.

Note *Visual* dBASE compiles programs before running them, and assigns the compiled files the same name as the original, but with the letter "O" as the last letter in the filename extension. For example, the compiled version of

SALESRPT.PRГ would be SALESRPT.PRO. If SALESRPT.PRO already exists, it is overwritten. For this reason, avoid using filename extensions ending in "O" in directories containing compiled programs.

See Also DO, CREATE FILE, SET DEVELOPMENT, SET EDITOR

CREATE DATAMODULE

Opens the Data Module designer.

Syntax CREATE DATAMODULE
[<filename> | ? | <filename skeleton>]
[CUSTOM] | [WIZARD | EXPERT [PROMPT]]

<filename> | ? | <filename skeleton> The file to display and edit. The default extension is .DMD. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE DATAMODULE without an option, *Visual* dBASE creates an untitled empty data module.

CUSTOM Invokes the Custom Data Module designer instead of the Data Module designer. The default extension is .CDM instead of .DMD.

WIZARD | EXPERT [PROMPT] If the PROMPT clause is used, a dialog appears asking if you want to use the Data Module designer or the Data Module wizard. You can then invoke either the designer or the wizard. The WIZARD clause without PROMPT causes the Table wizard to be invoked. You may use the keyword EXPERT instead of WIZARD.

You cannot combine the CUSTOM and WIZARD options; there is no Custom Data Module wizard.

Description Use CREATE DATAMODULE to open the Data Module designer and create new or edit existing data modules. The Data Module designer automatically generates *Visual* dBASE program code that defines the data in the data module, and stores this code in an editable source code file with a .DMD extension. Use a DataModRef object to use a data module.

If you're creating a new data module, CREATE DATAMODULE displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY DATAMODULE command to edit an existing file without being asked whether you want to modify it.

CREATE FILE

Displays a specified text file for editing, or displays an empty editing window.

Syntax CREATE FILE [<filename> | ? | <filename skeleton>]

Description CREATE FILE is identical to CREATE COMMAND, except that it defaults to displaying and editing text files, which have a .TXT extension (instead of program files, which have a .PRG extension).

See Also CREATE COMMAND, SET EDITOR

CREATE FORM

Opens the Form designer to create or modify a form.

Syntax CREATE FORM
[<filename> | ? | <filename skeleton>]
[CUSTOM] | [WIZARD | EXPERT [PROMPT]]

<filename> | ? | <filename skeleton> The form to create or modify. The default extension is .WFM. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE FORM without an option, *Visual* dBASE creates an untitled empty form.

CUSTOM Invokes the Custom Form designer instead of the Form designer. The default extension is .CFM instead of .WFM.

WIZARD | EXPERT [PROMPT] If the PROMPT clause is used, a dialog appears asking if you want to use the Form designer or the Form wizard. You can then invoke either the designer or the wizard. The WIZARD clause without PROMPT causes the Form wizard to be invoked. You may use the keyword EXPERT instead of WIZARD.

You cannot combine the CUSTOM and WIZARD options; there is no Custom Form wizard.

Description Use CREATE FORM to open the Form designer or Form wizard and create or modify a form interactively. The Form designer automatically generates *Visual* dBASE program code that defines the contents and format of a form, and stores this code in an editable source code file with a .WFM extension. DO the .WFM file to run the form.

If you're creating a new form, CREATE FORM displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY FORM command to edit an existing file without being asked whether you want to modify it.

You may invoke the Custom Form designer by specifying the CUSTOM keyword. A custom form is stored in a .CFM file, and does not have the standard bootstrap code that instantiates and opens a form when the file is executed. It is intended to be used as a base class for other forms. A single .CFM

file may contain more than one custom form class definition. If there is more than one form class in the .CFM file, *Visual dBASE* presents a list of classes to modify.

By default, the Form designer creates a class made up of the name of the file plus the word “Form”. For example, when creating STUDENT.WFM, the form class is named StudentForm. The Custom Form designer uses the word “CForm” instead; for example, in SCHOOL.CFM, the form class is named SchoolCForm.

See the *Developer’s Guide* for instructions on using the Form designer.

See Also CREATE COMMAND, DO

CREATE LABEL

Opens the Label designer to create or modify a label file.

Syntax CREATE LABEL
[<filename> | ? | <filename skeleton>]
[WIZARD | EXPERT [PROMPT]]

<filename> | ? | <filename skeleton> The label file to create or modify. The default extension is .LAB. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual dBASE* looks for the file in the current directory. If you issue CREATE LABEL without an option, *Visual dBASE* creates an untitled label file.

WIZARD | EXPERT [PROMPT] If the PROMPT clause is used, a dialog appears asking if you want to use the Label designer or the Label wizard. You can then invoke either the designer or the wizard. The WIZARD clause without PROMPT causes the Label wizard to be invoked. You may use the keyword EXPERT instead of WIZARD.

Description Use CREATE LABEL to open the Label designer and create new or edit existing labels. The Label designer automatically generates *Visual dBASE* program code that defines the contents and format of the labels, and stores this code in an editable source code file with a .LAB extension. DO the .LAB file to print the labels.

If you're creating a new label file, CREATE LABEL displays an empty design surface. If you specify an existing file, *Visual dBASE* asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY LABEL command to edit an existing file without being asked whether you want to modify it.

See Also CREATE REPORT, DO

CREATE MENU

Opens the Menu designer to create or modify a menu file.

Syntax CREATE MENU [<filename> | ? | <filename skeleton>]

<filename> | ? | <filename skeleton> The menu file to create or modify. The default extension is .MNU. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE MENU without an option, *Visual* dBASE creates an untitled menu file.

Description Use CREATE MENU to open the Menu designer and create new or edit existing menus. The Menu designer automatically generates *Visual* dBASE program code that defines the contents of a menu, and stores this code in an editable source code file with a .MNU extension. To use the menu, assign the .MNU file name as the *menuFile* property of a form, or

DO <.MNU file> WITH <form reference>

to assign the menu to the form. The Menu designer always creates a menu named “root”, so that when assigned to a form, it is referenced as *form.root*.

If you're creating a new menu file, CREATE MENU displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY MENU command to edit an existing file without being asked whether you want to modify it.

See Also CREATE POPUP, *menuFile*

CREATE POPUP

Opens the Popup Menu designer to create or modify a popup menu file.

Syntax CREATE POPUP [<filename> | ? | <filename skeleton>]

<filename> | ? | <filename skeleton> The popup menu file to create or modify. The default extension is .POP. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE POPUP without an option, *Visual* dBASE creates an untitled popup menu file.

Description Use CREATE POPUP to open the Popup Menu designer and create new or edit existing popup menus. The Popup Menu designer automatically generates *Visual* dBASE program code that defines the contents of a popup menu, and

stores this code in an editable source code file with a .POP extension. To assign the popup menu to a form, create the popup as a property of the form with:

```
DO <.POP file> WITH <form reference>, <property name>
```

then assign the popup object to the form's *popupMenu* property.

If you're creating a new popup menu file, CREATE POPUP displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY POPUP command to edit an existing file without being asked whether you want to modify it.

See Also CREATE MENU, *popupMenu*

CREATE PROJECT

Syntax CREATE PROJECT

Description CREATE PROJECT displays the New Project dialog box, where you can name and locate a new project. The new project then becomes the current project in the Project Explorer.

Use MODIFY PROJECT to open existing project.

See Also MODIFY PROJECT

CREATE QUERY

Opens a new or existing query in the SQL designer.

Syntax CREATE QUERY [<filename> | ? | <filename skeleton>]

<filename> | ? | <filename skeleton> The SQL query file to create or modify. The default extension is .SQL. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE QUERY without an option, *Visual* dBASE creates an untitled SQL query file.

Description Use CREATE QUERY to open the SQL designer and create new or edit existing SQL queries. The SQL designer automatically generates an SQL statement that defines the query, and stores this statement in an editable source code file with a .SQL extension. The .SQL file can be run directly from the Navigator or used as the *sql* property of a Query object.

If you're creating a new SQL query file, CREATE QUERY displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY

QUERY command to edit an existing file without being asked whether you want to modify it.

See Also *sql*

CREATE REPORT

Opens the Report designer to create or modify a report.

Syntax CREATE REPORT
 [<filename> | ? | <filename skeleton>]
 [CUSTOM] | [WIZARD | EXPERT [PROMPT]]

<filename> | ? | <filename skeleton> The report to create or modify. The default extension is .REP. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory. If you issue CREATE FORM without an option, *Visual* dBASE creates an untitled empty report.

CUSTOM Invokes the Custom Report designer instead of the Report designer. The default extension is .CRP instead of .REP.

WIZARD | EXPERT [PROMPT] If the PROMPT clause is used, a dialog appears asking if you want to use the Report designer or the Report wizard. You can then invoke either the designer or the wizard. The WIZARD clause without PROMPT causes the Report wizard to be invoked. You may use the keyword EXPERT instead of WIZARD.

You cannot combine the CUSTOM and WIZARD options; there is no Custom Report wizard.

Description Use CREATE Report to open the Report designer or Report wizard and create or modify a report interactively. The Report designer automatically generates *Visual* dBASE program code that defines the contents and format of a report, and stores this code in an editable source code file with a .REP extension. DO the .REP file to run the report.

If you're creating a new report, CREATE REPORT displays an empty design surface. If you specify an existing file, *Visual* dBASE asks whether you want to modify it. If you reply no, nothing further happens. Use the MODIFY REPORT command to edit an existing file without being asked whether you want to modify it.

You may invoke the Custom Report designer by specifying the CUSTOM keyword. A custom report is stored in a .CRP file, and does not have the standard bootstrap code that instantiates and renders a report when the file is executed. It is intended to be used as a base class for other reports. A single .CRP file may contain more than one custom report class definition. If there is more than one report class in the .CRP file, *Visual* dBASE presents a list of classes to modify.

See the *Developer's Guide* for instructions on using the Report designer.

See Also CREATE COMMAND, DO

DEBUG

Opens the *Visual* dBASE Debugger.

Syntax DEBUG
[<filename> | ? | <filename skeleton> [WITH <parameter list>]]

<filename> | ? | <filename skeleton> The program file to debug. DEBUG ? and DEBUG <filename skeleton> display the Open Source File dialog box, from which you can select a file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. If you specify a file without including its extension, *Visual* dBASE assumes .PRG.

WITH <parameter list> Specifies expressions to pass as parameters to a program. For information about parameter passing, see the description of PARAMETERS.

Description Use DEBUG to open the Debugger and view or control program execution interactively. You must issue DEBUG in the Command window; the command has no effect in a program. If you issue DEBUG without any options, *Visual* dBASE opens the Debugger without loading a program file. (You can load a file to debug from the Debugger.)

To debug a function, open the program file that contains the function, and set a breakpoint at the FUNCTION or PROCEDURE line. When the function is called, the debugger will appear, at the breakpoint that you set.

If an unhandled exception or error occurs during program execution, the standard error dialog gives you the option of opening the Debugger at the line where the error occurred.

For more information, see the *Developer's Guide*, which describes the Debugger in detail.

See Also DISPLAY COVERAGE, ON ERROR, RESUME, SET COVERAGE, SUSPEND

DISPLAY COVERAGE

Displays the contents of a coverage file in the results pane of the Command window.

Syntax DISPLAY COVERAGE <filename1> | ? | <filename skeleton 1>
[ALL]
[SUMMARY]

[TO FILE <filename2> | ? | <filename skeleton 2>]
[TO PRINTER]

<filename1> | ? | <filename skeleton 1> The coverage file for the desired program. The ? and <filename skeleton 1> options display a dialog box from which you can select a coverage file. If you specify a file without including its path, *Visual* dBASE looks for the file in the current directory, then in the path you specify with SET PATH. If you specify a file without including its extension, *Visual* dBASE assumes .COV.

ALL Includes the coverage files, if any, for all other program files that could be called by the main program file, adding to the display:

- The total number of logical blocks exercised in all the program files combined
- The percentage of logical blocks exercised in all the program files combined

SUMMARY Excludes the logical blocks that were exercised. Without SUMMARY, both the logical blocks that were exercised, and the logical blocks *not* exercised are displayed. Use the SUMMARY option to find code that still needs to be exercised.

TO FILE <filename2> | ? | <filename skeleton 2> Directs output to <filename2> in addition to the results pane of the Command window. By default, *Visual* dBASE assigns a .TXT extension to <filename2> and saves the file in the current directory. The ? and <filename skeleton 2> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer in addition to the results pane of the Command window.

Description

A coverage file contains the results of the coverage analysis of a program file. You cause *Visual* dBASE to analyze the execution of any code in a program file by compiling the program file with coverage, either by having SET COVERAGE ON when the program is compiled, or with the #pragma coverage(on) directive in the program file. A coverage file is created whenever any code in the program file is executed.

The coverage file has the same name as the program file, and changes the last letter of the extension to the letter "V"; unless the file is a .PRG, in which case the coverage file has an extension of .COV. For example, the coverage file for GRADES.PRG is GRADES.COV, and the coverage file for STUDENTS.WFM is STUDENTS.WFV.

The coverage file accumulates statistics whenever any code in the program file is executed. You will usually want to make sure that all logical blocks in your code have been exercised. You may erase the coverage file to restart the coverage analysis totals.

DISPLAY COVERAGE displays the results of the coverage analysis:

- Each logical block, and how many times it was exercised
- The total number of blocks, and the number of blocks that were tested
- The percentage of blocks tested

DISPLAY COVERAGE pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information. Use the TO FILE clause to send the information to a file. Use the TO PRINTER clause to send the information to the printer. In either case, you can use SET CONSOLE OFF to suppress the display of the information in the results pane.

DISPLAY COVERAGE is the same as LIST COVERAGE, except that LIST COVERAGE does not pause with the first window of information but rather continuously lists the information until complete. This makes LIST COVERAGE more appropriate for outputting to a file or printer.

See Also #pragma, SET COVERAGE

DISPLAY MEMORY

Displays information about memory variables in the results pane of the Command window.

Syntax DISPLAY MEMORY
[TO FILE <filename> | ? | <filename skeleton>]
[TO PRINTER]

TO FILE <filename> | ? | <filename skeleton> Directs output to the text file <filename>, in addition to the results pane of the Command window. By default, Visual dBASE assigns a .TXT extension to <filename> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer in addition to the results pane of the Command window.

Description Use DISPLAY MEMORY to display the contents and size of a memory variable list. If you haven't used ON KEY or to reassign the F7 key, pressing F7 when the Command window has focus is a quick way to execute DISPLAY MEMORY.

DISPLAY MEMORY displays information about both user-defined and system memory variables. The following information on user-defined memory variables is displayed.

- Name
- Scope (public, private, local, static or hidden)
- Data type
- Value
- Number of active memory variables
- Number of memory variables still available for use
- Number of bytes of memory used by character variables
- Number of bytes of memory still available for user character variables
- Name of the program that initialized private memory variables

The following information on system memory variables is displayed.

- Name
- Scope (public, private, or hidden)
- Data type
- Current value

DISPLAY MEMORY pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information. Use the TO FILE clause to send the information to a file. Use the TO PRINTER clause to send the information to the printer. In either case, you can use SET CONSOLE OFF to suppress the display of the information in the results pane.

DISPLAY MEMORY is the same as LIST MEMORY, except that LIST MEMORY does not pause with the first window of information but rather continuously lists the information until complete. This makes LIST MEMORY more appropriate for outputting to a file or printer.

See Also CLEAR MEMORY, RESTORE, SAVE, STORE, RELEASE

DISPLAY STATUS

Displays information about the current *Visual* dBASE environment in the results pane of the Command window.

Syntax DISPLAY STATUS
[TO FILE <filename> | ? | <filename skeleton>]
[TO PRINTER]

TO FILE <filename> | ? | <filename skeleton> Directs output to the text file <filename>, in addition to the results pane of the Command window. By default, *Visual* dBASE assigns a .TXT extension to <filename> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer in addition to the results pane of the Command window.

Description Use DISPLAY STATUS to identify open tables and index files and to check the status of the SET commands. DISPLAY STATUS shows information related to the current session only.

If you haven't used ON KEY, SET, or SET FUNCTION to reassign the *F6* key, pressing *F6* when the Command window has focus is a quick way to execute DISPLAY STATUS.

DISPLAY STATUS displays the following information:

- Name and alias of open tables in each work area, and for each table:
 - Whether that table is the table in the currently selected work area
 - The language driver and character set of each open table
 - Names of all open indexes and their index key expressions in each work area

- Master index, if any, in each work area
- Locked records in each work area
- Database relations in each work area
- Filter conditions in each work area
- The name of the SET LIBRARY file, if any
- The name of all open SET PROCEDURE files
- SET PATH file search path
- SET DEFAULT drive setting
- Current work area
- SET PRINTER setting
- Current language driver and character set
- DBTYPE setting
- Numeric settings for SET MARGIN, SET DECIMALS, SET MEMOWIDTH, SET TYPEAHEAD, SET ODOMETER, SET REFRESH, and SET REPROCESS
- The current directory
- ON KEY, ON ESCAPE, and ON ERROR settings
- SET ON/OFF command settings
- Programmable function key and SET FUNCTION settings

DISPLAY STATUS pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information. Use the TO FILE clause to send the information to a file. Use the TO PRINTER clause to send the information to the printer. In either case, you can use SET CONSOLE OFF to suppress the display of the information in the results pane.

DISPLAY STATUS is the same as LIST STATUS, except that LIST STATUS does not pause with the first window of information but rather continuously lists the information until complete. This makes LIST STATUS more appropriate for outputting to a file or printer.

See Also SET(), SETTO()

DISPLAY STRUCTURE

Displays the field definitions of the specified table.

Syntax DISPLAY STRUCTURE
 [IN <alias>]
 [TO FILE <filename> | ? <filename skeleton>]
 [TO PRINTER]

IN <alias> Identifies the work area of the open table whose structure you want to display rather than that of the current table. For more information, see “Aliases” on page 12-2.

TO FILE <filename> | ? | <filename skeleton> Directs output to the text file <filename>, in addition to the results pane of the Command window. By default, Visual dBASE assigns a .TXT extension to <filename> and saves the file in the current directory. The ? and <filename skeleton> options display a dialog box in which you specify the name of the target file and the directory to save it in.

TO PRINTER Directs output to the printer in addition to the results pane of the Command window.

Description Use DISPLAY STRUCTURE to view the structure of the current or a specified table in the results pane of the Command window. DISPLAY STRUCTURE displays the following information about the current or specified table:

- Name of the table
- Type of table (Paradox, dBASE, or SQL)
- Table type version number
- Number of records
- Date of last update (DBF only)
- Fields
 - Field number
 - Field name (if SET FIELDS is ON, the greater-than symbol (>) appears next to each field specified with the SET FIELDS TO command)
 - Type
 - Length
 - Whether there is an index on that field
- Number of bytes per record (the sum of field lengths; for DBF includes one additional byte reserved for storing the asterisk that marks a record as deleted)

Multiply the total number of bytes per record by the number of records in the table to estimate the size of a DBF table (excluding the size of the table header).

DISPLAY STRUCTURE pauses when the results pane is full and displays a dialog box prompting you to display another screenful of information. Use the TO FILE clause to send the information to a file. Use the TO PRINTER clause to send the information to the printer. In either case, you can use SET CONSOLE OFF to suppress the display of the information in the results pane.

DISPLAY STRUCTURE is the same as LIST STRUCTURE, except that LIST STRUCTURE does not pause with the first window of information but rather continuously lists the information until complete. This makes LIST STRUCTURE more appropriate for outputting to a file or printer.

Neither DISPLAY STRUCTURE nor LIST STRUCTURE permit modification of an existing table structure. To alter the structure, use MODIFY STRUCTURE.

Example To display the structure of the Fish table in the \Samples directory:

```
use FISH
display structure
```

The following is displayed in the results pane of the Command window:

```
Structure for table  C:\Program Files\dBASE\Visual dBASE 75\Samples\fish.dbf
Table type          DBASE
Version             7
Number of rows      10
Last update         10/18/97
-----
Field  Field Name                                Type      Length  Dec  Index
```

| | | | | | |
|-------------|-------------|---------------|-----|---|---|
| 1 | ID | AUTOINCREMENT | 4 | | Y |
| 2 | Name | CHARACTER | 30 | | Y |
| 3 | Species | CHARACTER | 40 | | Y |
| 4 | Length CM | NUMERIC | 20 | 4 | N |
| 5 | Description | MEMO | 10 | | N |
| 6 | OLE Graphic | OLE | 10 | | N |
| ----- | | | | | |
| ** Total ** | | | 115 | | |

See Also MODIFY STRUCTURE

HELP

Activates the *Visual* dBASE Help system.

Syntax `HELP [<help topic>]`

<help topic> The Help topic you access with HELP.

Description Use the HELP command in the Command window to get information on *Visual* dBASE.

Visual dBASE locates the first Help topic in the index beginning with *<help topic>*. If only one topic with the index entry is found, that topic is displayed. If there are multiple matches, Help displays a dialog box to let you choose the topic. If there is no match, the Help index is opened with *<help topic>* as the current search value.

Pressing *F1* gives you context-sensitive help based on the control or window that currently has focus, or text that is highlighted in the Command window or Source Editor.

Example Here are some examples of using HELP in the command window.

```
help extern
help lockretryinterval
```

Note that the topic is not case-sensitive.

INSPECT()

Opens the Inspector, a window that lists object properties and lets you change their settings.

Syntax `INSPECT(<oRef>)`

<oRef> A reference to the object that you want to inspect.

LIST...

Description Use INSPECT() to examine and change object properties directly. For example, during program development you can use INSPECT() to evaluate objects and experiment with different property settings.

The Inspector is modeless, and doesn't affect program execution.

Note You can access the Inspector from the Form designer by pressing *F11*.

You can get help on any property in the Inspector by selecting the property and pressing *F1*.

See Also DISPLAY MEMORY, DISPLAY STATUS

LIST...

Lists information in the results pane of the Command window without pausing.

Syntax LIST COVERAGE <filename1> | ? | <filename skeleton 1>
[ALL]
[SUMMARY]
[TO FILE <filename2> | ? | <filename skeleton 2>]
[TO PRINTER]

LIST MEMORY
[TO FILE <filename> | ? | <filename skeleton>]
[TO PRINTER]

LIST STATUS
[TO FILE <filename> | ? | <filename skeleton>]
[TO PRINTER]

LIST STRUCTURE
[IN <alias>]
[TO FILE <filename> | ? | <filename skeleton>]
[TO PRINTER]

Description The LIST commands listed above are the same as their DISPLAY command counterparts, except that LIST commands do not pause with the first window of information but rather continuously list the information until complete. This makes the LIST versions more appropriate for outputting to a file or printer.

See Also DISPLAY COVERAGE, DISPLAY MEMORY, DISPLAY STATUS, DISPLAY STRUCTURE

MODIFY...

Modifies the corresponding file.

Syntax MODIFY COMMAND [<filename> | ? | <filename skeleton>]
 MODIFY DATAMODULE [<filename> | ? | <filename skeleton>]
 MODIFY FILE [<filename> | ? | <filename skeleton>]
 MODIFY FORM [<filename> | ? | <filename skeleton>]
 MODIFY LABEL [<filename> | ? | <filename skeleton>]
 MODIFY MENU [<filename> | ? | <filename skeleton>]
 MODIFY POPUP [<filename> | ? | <filename skeleton>]
 MODIFY QUERY [<filename> | ? | <filename skeleton>]
 MODIFY REPORT [<filename> | ? | <filename skeleton>]

Description The MODIFY commands listed above operate the same as their CREATE command counterparts, except that if the specified file exists it is modified without prompting. For more information, see the corresponding CREATE commands.

See also CREATE COMMAND, CREATE DATAMODULE, CREATE FILE, CREATE FORM, CREATE LABEL, CREATE MENU, CREATE POPUP, CREATE QUERY, CREATE REPORT

MODIFY PROJECT

Opens an existing project in the Project Explorer.

Syntax MODIFY PROJECT [<filename> | ? | <filename skeleton>]
 <filename> | ? | <filename skeleton> The project file to open. The default extension is .PRJ. The ? and <filename skeleton> options display a dialog box from which you can select a file. If you specify a file without including its path, Visual dBASE looks for the file in the current directory.

If you issue MODIFY PROJECT without an option, the New Project dialog is displayed, as if you issued CREATE PROJECT.

Description MODIFY PROJECT opens the specified .PRJ file in the Project Explorer, making it the current project.

See Also CREATE PROJECT

MODIFY STRUCTURE

Allows you to modify the structure of the current table.

Syntax MODIFY STRUCTURE

Description Use MODIFY STRUCTURE to change the structure of the current table by adding or deleting fields, or changing a field name, width, or data type. Issuing the MODIFY STRUCTURE command opens the Table designer, an interactive environment in which you can create or modify the structure of a table.

Before allowing changes to the structure of a dBASE table, *Visual* dBASE makes a backup of the original table assigning the file a .DBK extension. *Visual* dBASE then creates a new table file with the .DBF extension and copies the modified table structure to that file. When you've finished modifying a table structure, *Visual* dBASE copies the content of the backup file into the new structure. If data is accidentally truncated or lost, you can recover the original data from the .DBK file. Before modifying the structure of a table, make sure that you have sufficient disk space to create the backup file plus any temporary storage required to copy records between the two tables (approximately twice the size of the original table).

If a table contains a memo field, MODIFY STRUCTURE also creates a backup memo file to store the original memo field data. This file has the same name as the table, but is given a .TBK extension.

You shouldn't change a field name and its width or type at the same time. If you do, *Visual* dBASE won't be able to append data from the old field, and your new field will be blank. Change the name of a field, save the file, and then use MODIFY STRUCTURE again to change the field width or data type.

Also, don't insert or delete fields from a table and change field names at the same time. If you change field names, MODIFY STRUCTURE appends data from the old file by using the field position in the file. If you insert or delete fields as well as change field names, you change field positions and could lose data. You can, however, change field widths or data types at the same time as you insert or delete fields. In those cases, since MODIFY STRUCTURE appends data by field name, the data will be appended correctly.

Visual dBASE successfully converts data between a number of field types. If you change field types, however, keep a backup copy of your original file, and check your new files to make sure the data has been converted correctly.

If you convert numeric fields to character fields, *Visual* dBASE converts numbers from the numeric fields to right-aligned character strings. If you convert a character field to a numeric field, *Visual* dBASE converts numeric characters in each record to digits until it encounters a non-numeric character. If the first character in a character field is a letter, the converted numeric field will contain zero.

You can convert logical fields to character fields, and vice versa. You can also convert character strings that are formatted as a date (for example, mm/dd/yy or mm-dd-yy) to a date field, or convert date fields to character fields. You can't convert logical fields to numeric fields.

In general, *Visual* dBASE attempts to make a conversion you request, but the conversion must be a sensible one or data may be lost. Numeric data can easily be handled as characters, but logical data, for example, cannot become numeric.

To convert incompatible data types (such as logical to numeric), first add a new field to the file, use REPLACE to convert the data, then delete the old field.

If you modify the field name, length, or type of any fields that have an associated tag in the production (.MDX) file, the tag is rebuilt. If any indexes are open when you modify a table structure, *Visual* dBASE automatically closes those indexes when saving the modified table. You should re-index the table after you modify its structure.

Example The following example uses MODIFY STRUCTURE in the Command window to change the structure of a table:

```
use clients in select() exclusive
modify structure
close databases
```

The structure of a table can also be displayed but not changed by using the following commands in the Command window:

```
use clients in select() noupdate
display structure
close databases
```

See Also APPEND, APPEND MEMO, COPY STRUCTURE, CREATE, DISPLAY STRUCTURE, LIST STRUCTURE, REPLACE

SET

Displays a dialog box for viewing and changing the values of many SET commands. The changed values are stored in the VDB.INI file.

Syntax SET

Description Use SET to view and change settings interactively, instead of typing individual SET commands such as SET TALK ON in the Command window.

Note Any changes you make to settings by using SET are automatically saved to VDB.INI. This means that the settings will be in effect each time you start *Visual* dBASE. If you want to change the value of SET commands only temporarily, issue individual SET commands in the Command window or in a program.

Issuing SET is the same as choosing the Properties | Desktop menu option.

See Also DISPLAY STATUS, SET(), SETTO(), individual SET commands

SET BELL

Turns the computer bell on or off and sets the bell frequency and duration.

Syntax SET BELL ON | off

SET BLOCKSIZE

SET BELL TO
[<frequency expN>, <duration expN>]

<frequency expN> The frequency of the bell tone in cycles per second, which must be an integer from 37 to 32,767, inclusive.

<duration expN> The duration of the bell tone in milliseconds, which must be an integer from 1 to 2000 (two seconds), inclusive.

Description When SET BELL is ON, *Visual* dBASE produces a tone when you fill a data entry field or enter invalid data. SET BELL TO determines the frequency and duration of this tone, unless the computer is running Windows 95 and has a sound card. In that case, the Windows Default sound is played (through the sound card) instead of the tone.

Displaying CHR(7) in the results pane of the Command window sounds the “bell” whether SET BELL is ON or OFF.

SET BELL TO with no arguments sets the frequency and duration to the default values of 512 Hertz (cycles per second) for 50 milliseconds.

Example The following examples typed in the Command window set the bell to high and low pitch and short and long durations:

```
set bell to 50,1500    // A long, very low bell
? chr(7)              // Ring the bell (or Windows Default sound)
set bell to 10000,30   // Short, very high pitched
? chr(7)              // Ring the bell (or Windows Default sound)
```

See Also CHR(), SET CONFIRM

SET BLOCKSIZE

Changes the default block size of memo field and .MDX index files.

Syntax SET BLOCKSIZE TO <expN>

<expN> A number from 1 to 63 that sets the size of blocks used in memo and .MDX index files. (The actual size in bytes is the number you specify multiplied by 512.)

Default The default for SET BLOCKSIZE is 1 (for compatibility with dBASE III PLUS). To change the default, update the BLOCKSIZE setting in VDB.INI.

Description Use SET BLOCKSIZE to change the size of blocks in which *Visual* dBASE stores memo field files and .MDX index files on disk. The actual number of bytes used in blocks is <expN> multiplied by 512. Instead of using SET BLOCKSIZE, you can set the block size used for memo and .MDX index files individually, by using SET MBLOCK and SET IBLOCK commands.

After the block size is changed, memo fields created with the COPY, CREATE, and MODIFY STRUCTURE commands have the new block size. To change the block size of an existing memo field file, use the SET BLOCKSIZE command to change the block size and then copy the table containing the associated memo field to a new file. The new file then has the new block size.

Example The following example uses SET BLOCKSIZE to create another table that is a copy of Clients but has a memo blocksize of 1024 bytes embedded in its structure instead of the default of 512 bytes:

```
use Clients
? set("blocksize")           // Returns 1, each memo block = 512 bytes
set blocksize to 2
copy to Clients2
use Clients2
? set("blocksize")           // Returns 2
list files like *.DBT         // Note file size larger than Clients.DBT
close databases
```

See Also COPY, COPY INDEXES, CREATE, MODIFY STRUCTURE, INDEX, REINDEX, REPLACE, SET(), SET IBLOCK, SET MBLOCK

SET COVERAGE

Determines whether program files are compiled with coverage.

Syntax SET COVERAGE on | OFF

Description A coverage file is a binary file containing cumulative information on how many times, if any, *Visual* dBASE enters and exits (and thus fully executes) each logical block of a program. Use SET COVERAGE as a program development tool to determine which program lines *Visual* dBASE executes and doesn't execute each time you run a program.

A program file is either compiled with coverage or not. To disable coverage analysis, the file must be recompiled with coverage off.

There are two ways to control compilation with coverage. The first way is with SET COVERAGE, which can be either ON or OFF. The second way is with the coverage #pragma in the program file. The #pragma directive overrides the SET COVERAGE setting.

If a file is compiled with coverage enabled, *Visual* dBASE creates a new coverage file or updates an existing one. When *Visual* dBASE creates a coverage file, it uses the name of the program file, and changes the last letter of the extension to the letter "V"; unless the file is a .PRG, in which case the coverage file has an extension of .COV. For example, the coverage file for GRADES.PRG is GRADES.COV, and the coverage file for STUDENTS.WFM is STUDENT.WFV.

To view the contents of a coverage file, use `DISPLAY COVERAGE` or `LIST COVERAGE`. If the coverage file reveals that some lines aren't executing, you can respond by changing the program or the input to the program to make the lines execute. In this way, you can make sure that you test all lines of code in the program.

Coverage analysis divides a program into logical blocks. A logical block doesn't include commented lines or programming construct command lines such as `IF` and `ENDIF`. It does, however, include command lines within programming construct command lines. If your program doesn't contain any programming constructs (like `IF`, `DO WHILE`, `FOR...ENDFOR`, `SCAN...ENDSCAN`, `LOOP`, `DO CASE`, `DO...UNTIL`), the program has only one logical block consisting of all uncommented command lines.

The coverage file identifies a logical block by its corresponding program line number(s):

```

Line 1  * UPDATES.PRG
Line 2  SET TALK OFF Block 1 (Lines 2-3)
Line 3  USE Customer INDEX Salespers
Line 4  SCAN
Line 5      DO CASE
Line 6      CASE Salesper = "S-12"
Line 7          SELECT 2 Block 2 (Lines 7-8)
Line 8          USE S12
Line 9      CASE Salesper = "L-5"
Line 10         SELECT 2 Block 3 (Lines 10-11)
Line 11         USE L5
Line 12      CASE Salesper = "J-25"
Line 13         SELECT 2 Block 4 (Lines 13-14)
Line 14         USE J25
Line 15      ENDCASE
Line 16      DO Changes Block 5 (Lines 16-17)
Line 17      SELECT 1
Line 18  ENDSCAN
Line 19  CLOSE ALL Block 6 (Lines 19-20)
Line 20  SET TALK ON

```

Visual dBASE writes the coverage file to disk when the program is unloaded from memory or when you issue a `LIST COVERAGE` or `DISPLAY COVERAGE`. To unload a program from memory, use `CLEAR PROGRAM`.

See Also `#pragma`, `CLEAR PROGRAM`, `COMPILE`, `DEBUG`, `DISPLAY COVERAGE`, `SET DEVELOPMENT`

SET DESIGN

Determines whether `CREATE` and `MODIFY` commands can be executed.

Syntax `SET DESIGN ON | off`

Default The default for SET DESIGN is ON. To change the default, set the DESIGN parameter in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the DESIGN parameter directly in VDB.INI.

Description When SET DESIGN is ON, *Visual* dBASE lets you use CREATE and MODIFY commands to create and modify tables, forms, labels, reports, text, and queries. To prevent users of your applications from creating and modifying these types of files, issue SET DESIGN OFF in your programs.

If you issue SET DESIGN ON or OFF in a subroutine, the setting is effective only during execution of that subroutine.

Example The default setting of SET DESIGN is usually ON. In this example the default setting in VDB.INI is OFF so that a user cannot use CREATE and MODIFY:

```
// In vdb.ini
[OnOffCommandSettings]
design=OFF
```

See Also CREATE, CREATE FORM, CREATE LABEL, CREATE REPORT, MODIFY COMMAND, MODIFY FILE, MODIFY STRUCTURE

SET DEVELOPMENT

Determines whether *Visual* dBASE automatically compiles a program, procedure, or format file when you change the file and then execute it or open it for execution.

Syntax SET DEVELOPMENT ON | off

Default The default for SET DEVELOPMENT is ON. To change the default, set the DEVELOPMENT parameter in VDB.INI. To do so, either use the SET command to specify the "Ensure Compilation" setting interactively, or enter the DEVELOPMENT parameter directly in VDB.INI.

Description When SET DEVELOPMENT is ON and you execute a program file with DO, or open a procedure or format file, *Visual* dBASE compares the time and date stamp of the source file and the compiled file. If the source file has a later time and date stamp than the compiled file, *Visual* dBASE recompiles the file.

When SET DEVELOPMENT is ON and you change a source program, procedure, or format file with MODIFY COMMAND, *Visual* dBASE erases the corresponding compiled file. When you then execute the program or open the procedure or format file, *Visual* dBASE recompiles it.

When SET DEVELOPMENT is OFF, *Visual* dBASE doesn't compare time and date stamps, and executes or opens existing compiled program, procedure, or format files. When you modify a source file and then open or execute it, *Visual* dBASE first looks for a compiled file in memory and executes it if found. If no compiled file is in memory, *Visual* dBASE looks for a compiled disk file and executes it if found. If no compiled file is found, *Visual* dBASE compiles the file.

When you DO a program, open a procedure file with SET PROCEDURE, or open a format file with SET FORMAT, *Visual* dBASE always looks for, opens, and executes a compiled file. Therefore, if *Visual* dBASE can't find a compiled version of a source file when you execute or open the source, *Visual* dBASE compiles the file regardless of the SET DEVELOPMENT setting.

During program development, when you're editing files often, you should turn SET DEVELOPMENT ON. This ensures that you're always executing an up-to-date compiled file.

Turn SET DEVELOPMENT OFF when you no longer plan to change any source code. Turning SET DEVELOPMENT OFF speeds up program execution because *Visual* dBASE doesn't have to check time and date stamps. You might want to set the DEVELOPMENT parameter to OFF in the VDB.INI file you distribute with your compiled code.

See Also CLEAR PROGRAM, COMPILE, DO, SET PROCEDURE

SET EDITOR

Specifies the text editor to use when creating and editing programs and text files.

Syntax SET EDITOR TO
[<expC>]

<expC> The expression you would enter at the DOS prompt or as the Windows command line to start the editor, usually the name of the editor's executable file (.EXE) or a Windows .PIF file. If <expC> doesn't include the file's full path name, *Visual* dBASE looks for the file in the current directory, then in the DOS path.

Default The default for SET EDITOR is the *Visual* dBASE internal Source editor. To specify a different default editor, set the EDITOR parameter in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the EDITOR parameter directly in VDB.INI.

Description Use SET EDITOR to specify an editor other than the default *Visual* dBASE Source editor to use when creating or editing text files. The file name you specify can be any text editor that produces standard ASCII text files. The specified editor opens when you issue CREATE/MODIFY FILE or CREATE/MODIFY COMMAND. If you issue SET EDITOR TO without a file name for <expC>, *Visual* dBASE returns to the default editor.

You can use SET EDITOR to specify a .PIF file, which is a Windows file that controls the Windows environment for a DOS application, or a Windows .EXE file. Start the DOS editor by running the .PIF file rather than the .EXE. When you issue commands that run in a DOS window, *Visual* dBASE loads COMMAND.COM using VDB.PIF in the _dbwinhome directory. You can use the Windows PIF Editor to customize the settings in VDB.PIF. For more information about .PIF files, see your Windows documentation. If there is not

enough memory available to access an external editor, *Visual* dBASE returns an "Unable to execute DOS" error message.

If the text editor you specify is already in use when you open a memo or file for editing, a second instance of the editor starts.

Example The following example changes the default editor to Brief, to Write, the Windows editor and back to the *Visual* dBASE editor:

```
set editor to "c:\brief\b"
// now c:\brief does not need to be in the path statements
// modify command now accesses Brief.
modify command temp.prg
set editor to
modify command temp
// Reverts to Visual dBASE editor
```

You might not have sufficient RAM to access an external editor in which case *Visual* dBASE gives an "Unable to execute DOS" error message.

See Also `_dbwinhome`, `MEMORY()`, `MODIFY COMMAND`, `MODIFY FILE`

SET IBLOCK

Changes the default block size used for new .MDX files.

Syntax SET IBLOCK TO <expN>

<expN> A number from 1 to 63 that sets the size of index blocks allocated to new .MDX files. The default value is 1. (The actual size in bytes is the number you specify multiplied by 512 bytes; however, the minimum size of a block is 1024 bytes.) To change the default, update the IBLOCK setting in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the IBLOCK parameter directly in VDB.INI.

Description Use SET IBLOCK to change the size of blocks in which *Visual* dBASE stores .MDX files on disk to improve the performance and efficiency of indexes. You can specify a block size from 1024 bytes to approximately 32K. The IBLOCK setting overrides any previous block size defined by the SET BLOCKSIZE command or specified in the VDB.INI file. After the block size has been changed, new .MDX index files are created with the new block size.

Multiple index (.MDX) files are composed of individual index blocks (or *nodes*). Nodes contain the value of keys corresponding to individual records and provide the information to locate the appropriate record for each key value. Since the IBLOCK setting determines the size of nodes, the setting also determines the number of key values that can fit into each node. When a single node can't contain all the key values in an index, *Visual* dBASE creates one or more parent nodes. These intermediate nodes also contain key values. Instead of pointing to record numbers, however, intermediate nodes point to *leaf nodes* or other lower-level intermediate nodes. If you increase the size of index blocks

and create a new .MDX file, the new and larger leaf nodes contain more key values.

Whether you can improve performance by storing key values in larger or smaller nodes depends on several factors: the distribution of data, if tables are linked together, the length of key values, the value of INDEXBYTES, and the type of operation requested. Typically, every .MDX file contains more than one index tag. Finding the best setting for a given .MDX file requires experimentation because the best size for one index tag might not be the best size for another.

The following is a list of basic principles governing index performance.

- Since nodes might not be sequential, *Visual* dBASE reads only one node at a time from the disk. Reading more than one node is usually inefficient, because typically the second node is not the next node in the sequential list.
- Once a node is read into memory, *Visual* dBASE attempts to store it there for later use. The amount of space devoted to caching the index nodes is determined by the setting of INDEXBYTES.
- When users link several tables together, for example, with SET RELATION, performance is better if all the relevant nodes for the tables are in memory simultaneously. For example, if a large node for table B pushes out the previously read node for table A, *Visual* dBASE must find and read the table A node again from disk when the node for table A needs to be used again. If both nodes remain in memory, performance can be improved.
- When tables have many identical key values, *Visual* dBASE might have to store them in many nodes. In this situation, performance might be improved by increasing the node size so that *Visual* dBASE reads fewer nodes from disk to load the same number of key values into memory.
- Small node sizes can cause performance degradation. This occurs because as nodes are read in and out, *Visual* dBASE attempts to cache them all. When the small nodes are removed from memory by more recently read nodes, they leave unused spaces in memory that are too small to contain larger nodes. Over time, memory can become fragmented, resulting in slower performance.

Example This example creates two .MDXs containing identical data but with different IBLOCK settings and consequently, different file sizes:

```
close data
delete file Co1.mdx
delete file Co2.mdx
// remove any previous .mdx
// create the MDXs
use Company exclusive
set iblock to 2
index on CompCode tag CompCode of Co1
index on Company tag Company of Co1
index on City tag City of Co1
set iblock to 20
```

```

index on CompCode tag CompCode of Co2
index on Company tag Company   of Co2
index on City tag City         of Co2
dir co?.mdx

```

Two .MDXs, Co1.MDX and Co2.MDX are created with different IBLOCK settings. CO1 and CO2 will have different file sizes because their block lengths are different.

See Also COPY, COPY INDEXES, CREATE, MODIFY STRUCTURE, INDEX, REINDEX, REPLACE, SET(), SET BLOCKSIZE, SET MBLOCK

SET MBLOCK

Changes the default block size of new memo field (.DBT) files.

Syntax SET MBLOCK TO <expN>

<expN> A number from 1 to 512 that sets the size of blocks used to store new memo (.DBT) files. (The actual size in bytes is the number you specify multiplied by 64.)

Default The default value for SET MBLOCK is 8 (or 512 bytes). To change the default, update the MBLOCK setting in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the MBLOCK parameter directly in VDB.INI.

Description Use SET MBLOCK to change the size of blocks in which *Visual* dBASE stores new memo field (.DBT) files on disk. You can specify a block size from 64 bytes to approximately 32K. The MBLOCK setting overrides any previous block size defined by the SET BLOCKSIZE command or specified in the VDB.INI file. After the block size has been changed, new memo .DBT files are created with the new block size. *Visual* dBASE stores data in each memo field in a group made up of as many blocks as needed.

After the block size is changed, memo fields created with the COPY, CREATE, and MODIFY STRUCTURE commands have the new block size. To change the block size of an existing memo field file, use the SET BLOCKSIZE command to change the block size and then copy the table containing the associated memo field to a new file. The new file then has the new block size.

When the block sizes are large and the memo contents are small, memo (.DBT) files contain unused space and become larger than necessary. If you expect the contents of the memo fields to occupy less than 512 bytes (the default size allocated), set the block size to a smaller size to reduce wasted space. If you expect to store larger pieces of information in memo fields, increase the size of the block.

SET MBLOCK is similar to the older SET BLOCKSIZE command except for two advantages:

- You can allocate different block sizes for memo field and index data, whereas SET BLOCKSIZE requires the same block size for both. To allocate block sizes for index data, use SET IBLOCK.
- You can specify smaller blocks with SET MBLOCK than with SET BLOCKSIZE. SET BLOCKSIZE creates blocks in increments of 512 bytes, compared to 64 bytes with SET MBLOCK.

Example The following example uses SET MBLOCK to create another table that is a copy of Clients but has a memo blocksize of 256 bytes embedded in its structure versus the default of 512 bytes. This technique applies if memo entries are normally less than 256 bytes and you want to minimize wasted space in the .DBT file:

```
use Clients
? set("mblock")
// Returns default of 8;each memo block = 512 bytes
set mblock to 4
copy to Clients2
use Clients2
? set("mblock")
// Returns 4
list files like Clients*.dbt
// Note that Clients2.dbt is smaller than Clients.dbt
close databases
```

See Also CREATE, MODIFY STRUCTURE, REPLACE, SET(), SET BLOCKSIZE, SET IBLOCK

SET TALK

Determines whether *Visual* dBASE displays messages in the status bar, or displays memory variable assignments in the results pane of the Command window.

Syntax SET TALK ON | off

Default The default for SET TALK is ON. To change the default, set the TALK parameter in VDB.INI. To do so, either use the SET command to specify the setting interactively, or enter the TALK parameter directly in VDB.INI.

Description When SET TALK is ON, *Visual* dBASE uses the current SET ODOMETER setting to indicate when operations such as COUNT and SORT are in progress in the status bar. It also displays the results of memory variable assignments (using STORE or =) in the results pane of the Command window.

Depending on the amount of memory your system has and the amount of memory particular operations require, issuing SET TALK OFF might improve the performance of some operations.

Use SET TALK with SET ALTERNATE to send SET TALK output to a file or printer rather than to the results pane of the Command window.

When SET TALK is ON, *Visual* dBASE reports the results of the BUILD command in a dialog box. If SET TALK is OFF, nothing happens when BUILD is successful.

Example This example shows the effect of SET TALK ON and SET TALK OFF while creating a memory variable:

```
Oldtalk=SET("TALK")
set talk on
First="Susan"                // Susan
Last="O'Shenko"             // O'Shenko
Name=Last+", "+First
set talk off
First="Tom"
Last="Frost"
Name=Last+", "+First
? Name
// Name will be set to "Frost, Tom" but this will not
// be displayed in the status bar
set talk &Oldtalk
```

When TALK is OFF, the assignment of "Tom" to First and "Frost" to Last and "Frost, Tom" to Name are not displayed.

In the following example Count displays to the status bar when TALK is ON:

```
close all
use company
set talk on                 // Talk on
count to Recs               // Count displays in status bar
set talk off                // Talk off
count to Recs               // No display
```

See Also SET ALTERNATE, SET CONSOLE, SET ODOMETER, STORE

Everything Else (Except Preprocessor)

This section includes *Visual* dBASE language elements that pertain to errors, security, and locale.

ACCESS()

Returns the access level of the current user for DBF table security.

Syntax ACCESS()

Description In DBF table security, an access level is assigned to each user. The access level is a number from 1 to 8, with 1 being the highest level of access. Use ACCESS() to build security into an application. The access level returned can be used to test privileges assigned with PROTECT. If a user is not logged in to the application, ACCESS() returns 0 (zero).

If you write programs that use encrypted files, check the user's access level early in the program. If ACCESS() returns zero, your program might prompt the user to log in, or to contact the system administrator for assistance.

For more information, see PROTECT.

See Also LOGOUT, PROTECT, SET ENCRYPTION, USER()

ANSI()

Returns a character string that is the ANSI equivalent of a character expression using the current global character set.

Syntax ANSI(<expC>)

<expC> The character expression to convert to ANSI characters.

Description Each character in a string is represented by a byte value from 0 to 255. (Because *Visual* dBASE is a Unicode application it is actually more complicated than this internally, but this is how things appear to the programmer.) The character that each number represents is determined by the current *character set*. The same series of bytes may represent different characters with different character sets. Conversely, the same character may be represented by a different byte value in different character sets.

Windows uses the ANSI (American National Standards Institute) character set. *Visual* dBASE supports that character set, and multiple OEM (Original Equipment Manufacturer) character sets, which are identified by a *code page* number. Each character set, along with other country-specific information, is represented in *Visual* dBASE by a *language driver*. The classic IBM extended character set—the one with box drawing characters used in text screens and the MS-DOS Command Prompt—is an OEM character set, represented by the DB437US0 language driver, the default language driver for the United States.

There are more characters in use than will fit in a 256-character character set; therefore some characters are present in some character sets but not in others. While the lower 128 characters of these character sets are always identical (they match the standard 7-bit ASCII characters), the upper 128 characters (sometimes referred to as high-ASCII characters) may differ. Sometimes the same characters have different byte values. For example, the lowercase a-umlaut (ä) is character 132 in the DB437US0 OEM character set, and character 228 in the ANSI character set.

Use the ANSI() and OEM() functions to convert characters between the ANSI character set and an OEM character set, represented by the current global language driver.

Note If the current language driver is an ANSI language driver, like DBWINUS0, then DB437US0 is used as the OEM character set.

ANSI() treats the byte values of the characters in <expC> as OEM characters, and attempts to convert them to the equivalent characters in the ANSI character set. OEM() does the reverse. If no direct conversion is possible, then the characters are converted to similar-looking characters.

Example Suppose you are calling an EXTERNed function SomeFunc() from a DLL that expects a string parameter to be an ANSI string. You are using DB437US0 as your global language driver, so you use the ANSI() function to do the conversion:

```
extern CVOID SomeFunc( CSTRING ) SomeDLL.DLL
SomeFunc( ansi( cParameter ))
```

See Also ASC(), CHR(), LDRIVER(), OEM()

CANCEL

Halts program execution.

Syntax CANCEL

Description Use CANCEL to cancel program execution in the midst of a process. You can also issue CANCEL in the Command window when a program is suspended (with SUSPEND) to cancel execution of the suspended program.

While procedural applications are characterized by deeply nested subroutines that wait for user actions, applications in *Visual* dBASE are event-driven; objects sit on-screen, waiting for something to happen. While waiting for an event, no programs are being executed. When an event occurs, the event handler is fired, and when that's done, *Visual* dBASE goes back to waiting for events. Issuing CANCEL will halt the current event handler thread, but does cause *Visual* dBASE to stop responding to events. To do that the object itself must be removed from the screen or destroyed.

A process that is halted simply stops; no message or exception is generated. In the main routine of a process, issuing CANCEL has the same effect as issuing RETURN: the process is terminated. A program or thread halted by CANCEL performs the standard cleanup for a completed process. All local and private memory variables are cleared. Control returns to the object that started the process, if it's still available; usually a form, menu, or the Command window.

See Also DO, QUIT, RESUME, RETRY, RETURN, SUSPEND

CERROR()

Returns the number of the last compiler error.

Syntax CERROR()

Description Use CERROR() before executing a new program to test whether the source code compiles successfully. If no compiler error occurs, CERROR() returns 0. CERROR() is updated each time you or dBASE compile a program or format file. CERROR() isn't affected by warning messages generated by compiling.

Use CERROR() in a program file. If you issue ? CERROR() in the Command window, it returns 0. (This is because dBASE is compiling the "? CERROR()" command itself, which does not cause a compiler error.)

See the table in the description of ERROR() that compares ERROR(), MESSAGE(), DBERROR(), DBMESSAGE(), SQLERROR(), SQLMESSAGE(), and CERROR().

Example The following program segment uses CERROR() in a DO WHILE loop to make the user edit the program until it compiles successfully:

CHARSET()

```
DO WHILE .T.
  Clear
  MODIFY COMMAND USER.PRG
  ON ERROR ? ERROR(), MESSAGE(), CERROR()
  COMPILE USER.PRG
  IF CERROR()>0
    ?
    WAIT "Your program didn't compile. Press a key to edit your .PRG."
  LOOP
ELSE
  EXIT
ENDIF
ENDDO
```

See Also COMPILE, DBERROR(), DBMESSAGE(), ERROR(), MESSAGE(), ON ERROR

CHARSET()

Returns the name of the character set the current table or a specified table is using. If no table is open and you issue CHARSET() without an argument, it returns the global character set in use.

Syntax CHARSET([<alias>])

<alias> A work area number (1 through 225), letter (A through J), or alias name. The work area letter or alias name must be enclosed in quotes.

Description Use CHARSET() to learn which character set the current table or a specified table is using. If you don't pass CHARSET() an argument, it returns the name of the character set of the current table or, if no tables are open, the global character set in use. CHARSET() also returns information on Paradox and SQL databases.

The character set a table's data is stored in depends on the language driver setting that was in effect when the table was created. With *Visual* dBASE, you can choose the language driver that applies to your dBASE data in the [CommandSettings] section in the VDB.INI file.

The value CHRSET() returns is a subset of the value LDRIVER() returns. For more information, see LDRIVER().

Example This example shows the CHARSET() function and a sample response:

```
? CHARSET()           && Returns DOS:437
```

See Also ANSI(), LDRIVER(), OEM(), SET LDCHECK

DBERROR()

Returns the number of the last BDE error.

Syntax DBERROR()

Description DBERROR() returns the BDE error number of the last BDE error generated by the current table. To learn the BDE error message itself, use DBMESSAGE().

See the table in the description of ERROR() that compares ERROR(), MESSAGE(), DBERROR(), DBMESSAGE(), SQLERROR(), SQLMESSAGE(), and CERROR().

See Appendix C for a listing of all error messages.

Example The following example uses ON ERROR to branch to an error procedure that uses DBERROR() to return what BDE error has occurred during the BROWSE and DBMESSAGE() to return what it means:

```
USE Clients
ON ERROR DO Recovery
COPY TO TEMP
USE TEMP
BROWSE

PROCEDURE Recovery
CLOSE DATABASES
CLEAR
IF ERROR()=239
? "The BDE error was error number: " + STR(DBERROR())
? "Which means: " + DBMESSAGE()
ELSE
? "No BDE error encountered"
ENDIF
RETURN
```

See Also CERROR(), DBMESSAGE(), ERROR(), MESSAGE(), SQLERROR(), SQLMESSAGE()

DBMESSAGE()

Returns the error message of the last BDE error.

Syntax DBMESSAGE()

Description DBMESSAGE() returns the error message of the most recent BDE error.

See the table in the description of ERROR() that compares ERROR(), MESSAGE(), DBERROR(), DBMESSAGE(), CERROR(), SQLERROR(), and SQLMESSAGE().

See online Help for a listing of all error messages.

Example See DBERROR()

See Also CERROR(), DBERROR(), ERROR(), MESSAGE(), SQLERROR(), SQLMESSAGE()

ERROR()

Returns the number of the most recent dBASE error.

Syntax ERROR()

Description Use ERROR() to determine the error number when an error occurs. ERROR() is initially set to 0. ERROR() returns an error number when an error occurs, and remains set to that number until one of the following happens:

- Another error occurs
- RETRY is issued
- The subroutine in which the error occurs completes execution

The following table compares the functionality of CERROR(), DBERROR(), DBMESSAGE(), ERROR(), MESSAGE(), SQLERROR(), and SQLMESSAGE().

| Function | Returns |
|---------------|-----------------------|
| CERROR() | Compiler error number |
| DBERROR() | BDE error number |
| DBMESSAGE() | BDE error message |
| ERROR() | dBASE error number |
| MESSAGE() | dBASE error message |
| SQLERROR() | Server error number |
| SQLMESSAGE() | Server error message |

See online Help for a listing of all error codes.

Example See ON ERROR

See Also CERROR(), DBERROR(), DBMESSAGE(), MESSAGE(), ON ERROR, RETRY, SQLERROR(), SQLMESSAGE()

ID()

Returns the name of the current user on a *local area network* (LAN) or other multiuser system.

Syntax ID()

Description ID() accepts no arguments and returns the name of the current user as a character string. ID() returns an empty string when you call it on a single-user system or when a user name isn't registered on a multiuser system.

Example The following example keeps track of the last network user to update a record. It updates a network database and puts ID(), the current user, into a field called USER:

```
PROCEDURE OkToChange && Updates a network database and logs user name
IF ID() <> ""
    REPLACE NAME WITH cName, USER with ID()
ELSE
    CLEAR
    ? "You have lost your network connection. Data not saved."
    WAIT
ENDIF
RETURN
```

See Also CONVERT, LKSYS(), NETWORK()

LDRIVER()

Returns the name of the language driver the current table or a specified table is using. If no table is open and you issue LDRIVER() without an argument, it returns the global language driver in use.

Syntax LDRIVER([<alias>])

<alias> A work area number (1 through 225), letter (A through J), or alias name. The work area letter or alias name must be enclosed in quotes.

Description Use LDRIVER() to learn which language driver the current table or a specified table is using. If you don't pass LDRIVER() an argument, it returns the name of the language driver of the current table or, if no tables are open, the global language driver in use. LDRIVER() also returns information on Paradox and SQL databases.

The language driver associated with a table depends on the DOS code page or the BDE language driver setting that was in effect when the table was created. With *Visual* dBASE, you can choose the language driver that applies to your dBASE data in the [CommandSettings] section in the VDB.INI file. For example, you can load a German language driver to work with a table created while that driver was active.

Example This example shows the LDRIVER() function and a sample response:

```
? LDRIVER() && DB437US0
```

This example first closes all tables and obtains the global language driver. Then it opens a table and checks whether the table was created with the global language driver. If not, a warning is displayed:

`LINENO()`

```
CLOSE ALL
SET LDCHECK OFF
* this program replaces the LDCHECK alert message
GlobalDriver=LDRIVER()
USE Customer
TableLangDriver=LDRIVER()
SET EXACT ON
IF GlobalDriver<>TableLangDriver
  ? "Warning: this table was created"+ "with a different language driver"
  ? "Global Language Driver: "+GlobalDriver
  ? DBF()+" Language Driver: "+TableLangDriver
  WAIT
ENDIF
SET EXACT OFF
```

See Also `ANSI()`, `CHARSET()`, `OEM()`, `SET LDCHECK`

LINENO()

Returns the number of the current program line in the current program, procedure, or user-defined function (UDF).

Syntax `LINENO()`

Description Use `LINENO()` to track program flow. Use it in conjunction with `PROGRAM()` to learn when a program executes a given line of code. You can also use `LINENO()` with `ON ERROR` to find out which line produces an error.

`LINENO()` is meaningful only when issued from within a program, procedure, or UDF. When issued in the Command window, `LINENO()` returns 0.

`LINENO()` always returns the actual program line number; the number doesn't reflect the order in which the line executes within the program.

Example See `ON ERROR`

See Also `ERROR()`, `MESSAGE()`, `PROGRAM()`, `RESUME`, `SUSPEND`

LOGOUT

`LOGOUT` logs out the current user and sets up a new log-in dialog.

Syntax `LOGOUT`

Description `LOGOUT` logs out the current user from the current session and sets up a new log-in dialog when used with `PROTECT`. The `LOGOUT` command enables you to control user sign-in and sign-out procedures. The command forces a logout and prompts for a login.

When the command is processed, a log-in dialog appears. The user can enter a group name, log-in name, and password. The PROTECT command establishes log-in verification functions and sets the user access level.

LOGOUT closes all open tables, their associated files, and program files.

If PROTECT has not been used, and no DBSYSTEM.DB file exists, the LOGOUT command is ignored.

See also PROTECT, QUIT

MEMORY()

Returns the amount of currently available memory.

Syntax MEMORY([<expN>])

<expN> Any number, which causes MEMORY() to return the amount of available physical memory.

Description Use MEMORY() to determine how much memory is available in the system. It returns the amount in kilobytes (1024 bytes).

In Windows, available memory is a combination of physical memory (RAM installed in the computer) and virtual memory (disk space used to simulate memory).

When called with no parameters, MEMORY() returns the total amount of available memory: the amount of unused physical memory plus the amount of disk space available for virtual memory. By default, Windows 95 sets no maximum for virtual memory, so MEMORY() will return free physical memory plus free disk space on the hard drive used for virtual memory. On Windows NT, the size of the paging file used for virtual memory is set to a reasonable size.

When called with any numeric parameter, MEMORY() returns the amount of free physical memory. The amount of free physical memory can vary greatly, depending on what the system is doing or has just finished doing. For example, you may have more free physical memory right after viewing and dismissing a dialog box, since the memory that was used to display the dialog box is momentarily unallocated.

Visual dBASE's About dialog box displays the amount of free memory in bytes.

Example The following example warns if the user has less than a megabyte of RAM available:

```
IF MEMORY() >= 1000
  ? "You have at least a megabyte of RAM"
ELSE
  ?? "Warning: You have less than a megabyte of RAM:"
```

MESSAGE()

```
    ? MEMORY( ) , "Kb"  
ENDIF
```

See Also none

MESSAGE()

Returns the error message of the most recent dBASE error.

Syntax MESSAGE()

Description Use MESSAGE() with other error-trapping commands and functions, such as ON ERROR, RETRY, and ERROR(), to substitute specific responses and actions for dBASE default responses to errors.

MESSAGE() is initially set to an empty string. MESSAGE() returns an error message when an error occurs, and remains set to that error message until one of the following happens:

- Another error occurs
- RETRY is issued
- The subroutine in which the error occurs completes execution

To learn the BDE error message of the last BDE error generated by the current table, use DBMESSAGE().

See the table in the description of ERROR() that compares CERROR(), ERROR(), MESSAGE(), DBERROR(), DBMESSAGE(), SQLERROR(), and SQLMESSAGE().

See online Help for a listing of all dBASE error messages.

Example See ON ERROR

See Also CERROR(), DBERROR(), DBMESSAGE(), ERROR(), ON ERROR, RETRY, SQLERROR(), SQLMESSAGE()

NETWORK()

Returns .T. if *Visual* dBASE is running on a system in which a *local area network* (LAN) card or other multiuser system card has been installed.

Syntax NETWORK()

Description Use NETWORK() to determine if a program might be running in a network environment. For example, your program might need to do something in a network environment that it doesn't need to do in a single-user environment, such as issue USE with the EXCLUSIVE option.

NETWORK() returns .T. if a network card is installed; it doesn't determine whether a user is currently running *Visual* dBASE in a network environment. To determine whether a user is actually working in a network environment, use ID().

Example This example uses NETWORK() to test if the user has a network card installed. The user will SET EXCLUSIVE ON only when it might be needed. Without a network card, SET EXCLUSIVE can be ON permanently:

```
IF NETWORK()
    SET EXCLUSIVE OFF && set ON as needed
ELSE
    SET EXCLUSIVE ON && No network
ENDIF
```

See Also GETENV(), OS(), USE

OEM()

Returns a character string using the current global language driver that is the equivalent of an ANSI character expression.

Syntax OEM(<expC>)

<expC> The ANSI character expression to convert into characters in the global language driver.

Description OEM() is the inverse of ANSI(). For more information, see ANSI().

Example Suppose you have a Windows text file that you want to convert to OEM text that is readable by someone using MS-DOS. You convert each line using the OEM() function:

```
function convertToOEM( cSourceFile, cDestFile )
    local fSource, fDest
    fSource = new File()
    fDest = new File()
    fSource.open( cSourceFile )
    fDest.create( cDestFile )
    do while not fSource.eof()
        fDest.puts( oem( fSource.gets() ) )
    enddo
    fDest.close()
    fSource.close()
```

See Also ASC(), ANSI(), CHR(), LDRIVER()

ON ERROR

Executes a specified statement when an error occurs.

Syntax ON ERROR [<statement>]

<statement> The statement to execute when an error occurs. ON ERROR without a <statement> option disables any previous ON ERROR <statement>.

Description Use ON ERROR as a global error handler for unexpected conditions. For localized error handling—that is, for situations where you expect something might fail, like trying to open a file—use TRY...ENDTRY instead. ON ERROR also acts as a global CATCH; if there is no CATCH for a particular class of exception, an error occurs, which can be handled by ON ERROR.

When ON ERROR is active, *Visual* dBASE doesn't display its default error dialog; it executes the specified <statement>. To execute more than one statement when an error occurs, make the <statement> DO a program file, or call a function or method. In either case, the code that is executed in response to the error is known as the *ON ERROR handler*.

The ON ERROR handler usually uses the ERROR(), MESSAGE(), PROGRAM(), and LINE() functions to determine what the error is and where it occurred. In most applications, the only safe response to an unexpected condition is to log the error and quit the application. In some cases, you may be able to fix the problem and use the RETRY command to retry the statement that caused the error; or RETURN from the ON ERROR handler, which skips the statement that caused the error and executes the next statement.

While *Visual* dBASE is executing an ON ERROR statement, that particular ON ERROR <statement> statement is disabled. Thus, if another error occurs during the execution of <statement>, *Visual* dBASE responds with its default error dialog. You can, however, set another ON ERROR handler inside a routine called with ON ERROR.

SET("ON ERROR") returns the current ON ERROR <statement>.

Avoid using a *Visual* dBASE command recursively with ON ERROR.

Example Suppose you have an application management object assigned as the *core* property of the global *_app* object. The following ON ERROR command specifies a particular method of that object to act as a global error handler; a method call is a valid statement. It passes all relevant error information to the method as parameters:

```
on error _app.core.globalErrorTrap( program(), line(), error(), message() )
```

The ON ERROR handler can then display an error message and terminate the application, like this:

```
function globalErrorTrap( cProg, nLine, nError, cMsg )
    local c
    #define CHAR_CR chr(13)
```

```

c =          cMsg + CHAR_CR + CHAR_CR + ;
    "In: "    + cProg + CHAR_CR + ;
    "Line: "  + nLine + CHAR_CR + CHAR_CR + ;
    "If this error persists, contact program vendor."
msgbox( c, "Unexpected Application Error [" + nError + "]", 16 )
quit

```

See Also ERROR(), LINE(), MESSAGE(), PROGRAM(), RETRY, RETURN, SET ERROR, TRY

ON NETERROR

Executes a specified command when a multiuser-specific error occurs.

Syntax ON NETERROR [<command>]

<command> The command to execute when a multiuser-specific error occurs. To execute more than one command when such an error occurs, issue ON NETERROR DO <filename>, where <filename> is a program or procedure file containing the sequence of commands to execute. ON NETERROR without a <command> option disables any previous ON NETERROR <command> statement.

Description Use ON NETERROR to control a program's response to multiuser-specific errors. For example, in a multiuser environment on a *local area network* (LAN), an error can occur when two users attempt to alter the same record in a shared table at the same time, or when one user attempts to open a shared table that another user already has open for exclusive use.

ON NETERROR is similar to ON ERROR, except that ON ERROR responds to all run-time errors regardless of whether they're multiuser-specific. You can use ON ERROR to handle both single-user and multiuser errors, or you can use ON NETERROR to handle just multiuser errors. If you issue both ON ERROR and ON NETERROR, then ON ERROR responds to just single-user errors, leaving ON NETERROR to respond to multiuser errors.

While dBASE is executing an ON NETERROR command, that particular ON NETERROR <command> statement is disabled. Thus, if another multiuser-specific error occurs during the execution of <command>, dBASE responds with its default error messages. You can, however, set another ON NETERROR condition inside a subroutine called with ON NETERROR.

You should avoid using a dBASE command recursively with ON NETERROR.

Example The following example uses ON NETERROR in case the Clients table cannot be opened exclusively:

```

SET PROCEDURE TO PROGRAM(1) ADDITIVE
ON NETERROR Do NetErr
USE Clients EXCLUSIVE
* If Clients cannot be opened exclusively then

```

PROGRAM()

* the subroutine NetErr will be called.

```
PROCEDURE NETERR  
WAIT "Multi-user problem"
```

See Also DO, ERROR(), MESSAGE(), ON ERROR, RETRY, RETURN, SET EXCLUSIVE, SET REPROCESS, SQLERROR(), SQLMESSAGE()

PROGRAM()

Returns the name of the currently executing program, procedure, or user-defined function (UDF).

Syntax PROGRAM([<expN>])
<expN> Any number.

Description PROGRAM() returns the name of the lowest level executing subroutine—program, procedure, or UDF. PROGRAM() returns an empty string ("") when no program or subroutine is executing.

PROGRAM(*expN*) returns the full path name of the program that is currently running, which may be different from the name of the lowest level executing subroutine. This is shown in the following example.

```
SET PROCEDURE TO program1  
** Inside PROGRAM1.PRG is PROCEDURE procedure1  
** If procedure1 is running, note the following:  
? PROGRAM() returns PROCEDURE1  
? PROGRAM(expN) returns C:\VISUALDB\PROGRAM1.PRG.
```

You can issue PROGRAM() in the Command window if a program is suspended with SUSPEND. For example, if Program A calls Procedure B, and Procedure B is suspended, issuing PROGRAM() in the Command window returns the name of Procedure B; issuing PROGRAM(*expN*) in the Command window returns the full path name of the file containing Procedure B.

You can also use PROGRAM() with ON ERROR and LINENO() to identify the subroutine that was executing and the exact program line number at which the error occurred.

PROGRAM() returns the name of the subroutine in uppercase letters. PROGRAM() doesn't include a file-name extension even if the subroutine is a separate file, while PROGRAM(*expN*) always includes a file-name extension.

Example See ON ERROR

See Also DEBUG, LINENO(), ON ERROR, PROCEDURE, RESUME, SET PROCEDURE, SUSPEND

PROTECT

Creates and maintains DBF table security.

Syntax PROTECT

Description This command is issued within *Visual* dBASE by the database administrator, who is responsible for data security. PROTECT works in a single user or multiuser environment.

PROTECT is optional. Once you create table security, you may force all users to login when *Visual* dBASE or your *Visual* dBASE .EXE is started, or require login only when attempting to open an encrypted table.

This command displays a multi-page dialog. This dialog is the same dialog that is displayed when choosing dBASE table security in the File | Database Administration dialog box. The first time you use PROTECT, the system prompts you to enter and confirm an administrator password.

Warning Remembering the administrator password is essential. You can access the security system only if you can supply the password. Once established, the security system can be changed only if you enter the administrator password when you call PROTECT. Keep a hard copy of the database administrator password in a secured area. There is no way to retrieve a password from the system.

Once you enter the administrator password, you may setup and modify DBF table security.

The DBSYSTEM.DB file PROTECT builds and maintains a password system file called DBSYSTEM.DB, which contains a record for each user who accesses a PROTECTED system. Each record, called a user profile, contains the user's log-in name, account name, password, group name, and access level. When a user attempts to start *Visual* dBASE (if *Visual* dBASE is configured to require a log-in to start the program), or attempts to access an encrypted table (if *Visual* dBASE is configured to require a log-in when an encrypted table is accessed), *Visual* dBASE looks for a DBSYSTEM.DB file. You can specify a location for this file in the [CommandSettings] section of VDB.INI:

```
DBSYSTEM=C:\VISUALDB\BIN
```

If there is no DBSYSTEM entry in VDB.INI, *Visual* dBASE looks for the file in the same directory in which VDB.EXE is located. If it finds the file, it initiates the log-in process. If it does not find the file, there is no log-in process.

DBSYSTEM.DB is maintained as an encrypted file. Keep a record of the information contained in DBSYSTEM.DB, as well as a current backup copy of the file. If the DBSYSTEM.DB file is deleted or damaged and no backup is available, the database administrator will need to reinitialize PROTECT using the same administrator password and group names as before, or the data will be unrecoverable.

See Also ACCESS(), LOGOUT, SET ENCRYPTION, USER()

RESUME

Restarts program execution at the command line following the one at which program execution was suspended.

Syntax RESUME

Description RESUME causes dBASE to resume execution of a program that is suspended. You can suspend program execution by issuing SUSPEND. If you have not assigned a value to ON ERROR, you can also choose to suspend a program when an error occurs.

To restart program execution, enter RESUME in the Command window. The program file resumes execution at the line immediately following the line that caused it to become suspended. If you want to re-execute the line that caused an error, perhaps because you fixed the condition that caused the error, retype the program line at the command line before issuing RESUME.

Example See SUSPEND

See Also CANCEL, ON ERROR, RETRY, RUN, SUSPEND

RETRY

Returns control from a subroutine to the command line of the calling routine or Command window that called the subroutine.

Syntax RETRY

Description Use RETRY to re-execute a command—for example, one that resulted in an error. RETRY returns program control to the calling command. RETRY clears the memory variables created by the subroutine.

RETRY is valid only in program files.

You can use RETRY with ON ERROR to give the user more chances to resolve an error condition. Using RETRY with ON ERROR resets ERROR() to zero.

Example The following example uses the Recover procedure when an error is detected with ON ERROR. If the Clients table is already open in another work area when the USE command is executed, an error is returned. The Recover procedure uses CLOSE DATABASES to insure that all tables are closed and RETRY returns program flow to the USE command:

```
ON ERROR DO Recover
USE Clients EXCLUSIVE
```

```

BROWSE
ON ERROR
CLOSE DATABASES

PROCEDURE Recover
WAIT "An error has occurred.;
    Press any key to retry.."
CLOSE DATABASES
RETRY

```

See Also ERROR(), MESSAGE(), ON ERROR, RESUME, RETURN

SET ENCRYPTION

Establishes whether a newly created dBASE table is encrypted if PROTECT is used.

Syntax SET ENCRYPTION ON | off

Default The default for SET ENCRYPTION is ON.

Description This command determines whether copied dBASE tables (that is, tables created through the COPY, JOIN, and TOTAL commands) are created as encrypted tables. An encrypted table contains data encrypted into another form to hide the contents of the original table. An encrypted table can only be read after the encryption has been deciphered or copied to another table in decrypted form.

To access an encrypted table, you must enter a valid user name, group name, and password after the login screen prompts. Your authorization and access level determine whether you can or cannot copy an encrypted table. After you access the table, SET ENCRYPTION OFF to copy the table to a decrypted form. You need to do this if you wish to use EXPORT, COPY STRUCTURE EXTENDED, MODIFY STRUCTURE, or options of the COPY TO command.

Note Encryption works only with dBASE (.DBF) tables. Encryption works only with PROTECT. If you do not enter dBASE or access the table through the log-in screen, you will not be able to use encrypted tables.

All encrypted tables used concurrently in an application must have the same group name.

Encrypted tables cannot be JOINed with unencrypted tables. Make both tables either encrypted or unencrypted before JOINing them.

You can encrypt any newly created table by assigning the table an access level through PROTECT.

See also COPY TO, PROTECT, SET()

SET ERROR

Specifies one character expression to precede error messages and another one to follow them.

Syntax SET ERROR TO
[<preceding expC> [, <following expC>]]

<preceding expC> An expression of up to 33 characters to precede error messages. dBASE ignores any characters after 33.

<following expC> An expression of up to 33 characters to follow error messages. dBASE ignores any characters after 33. If you want to specify a value for <following expC>, you must also specify a value or empty string ("") for <preceding expC>.

Default The default for the message that precedes error messages is "Error: ". The default for the message that follows error messages is an empty string. To change the default, set the ERROR parameter in VDB.INI, using the following format:

```
ERROR = <preceding expC> [, <following expC>]
```

Description Use SET ERROR to customize the beginnings and endings of run-time error messages. SET ERROR TO without an argument resets the beginnings and endings to the default values.

SET ERROR is similar to ON ERROR; both can be used to customize error messages. SET ERROR, however, can only specify expressions to precede and follow a standard dBASE error message, while ON ERROR can specify the message itself. Also unlike ON ERROR, SET ERROR can't call a procedure that carries out a series of commands.

Example Use SET ERROR to customize error messages.

```
SET ERROR TO "Oops! - ", " - Please fix this."
? "a" = 1                                && generate a runtime error
SET ERROR TO
```

See Also ERROR(), MESSAGE(), ON ERROR

SET LDCHECK

Enables or disables language driver ID checking.

Syntax SET LDCHECK ON | off

Default The default for SET LDCHECK is ON. To change the default, set the LDCHECK parameter in VDB.INI.

Description Use SET LDCHECK to disable or enable dBASE's capability to check for language driver compatibility. This capability is important if you work with dBASE tables created with different dBASE configurations or different international versions of dBASE because it warns you of conflicting language drivers.

Language drivers determine the character set and sorting rules that dBASE uses, so if you create a dBASE table with one language driver and then use that file with a different language driver, some of the characters will appear incorrectly and you may get incorrect results when querying data.

Example See LDRIVER()

See Also CHARSET(), LDRIVER()

SET LDCONVERT

Determines whether data read from and written to character and memo fields is transliterated when the table character set does not match the global language driver.

Syntax SET LDCONVERT ON | off

Default The default for SET LDCONVERT is ON. To change the default, set the LDCONVERT parameter in VDB.INI.

Description Use SET LDCONVERT to determine whether the contents of character and memo fields in tables created with a given language driver in effect, are converted to match the language driver in effect at the time the fields are read or written to.

Language drivers determine the character set and sorting rules that dBASE uses, so if you create a dBASE table with one language driver and then use that file with a different language driver, some of the characters will appear incorrectly and you may get incorrect results when querying data.

In general, SET LDCONVERT should be ON to insure that dBASE behaves as expected when using data created under disparate language drivers.

See Also CHARSET(), LDRIVER(), SET LDCHECK

SQLERROR()

Returns the number of the last server error.

Syntax SQLERROR()

Description Use SQLERROR() to determine the error number of the last server error. To learn the text of the error message itself, use SQLMESSAGE().

See the table in the description of ERROR() that compares ERROR(), MESSAGE(), DBERROR(), DBMESSAGE(), SQLERROR(), SQLMESSAGE(), and CERROR().

See online Help for a listing of all error messages.

Example The following example uses SQLERROR() and SQLMESSAGE() to return an SQL error number and SQL error message to an ON ERROR routine that displays a MDI form with an error report:

```
ON ERROR DO ErrHndlr WITH ERROR(), MESSAGE(), ;
    SQLERROR(), SQLMESSAGE(), PROGRAM(), LINENO()
SET DBTYPE TO DBASE
OPEN DATABASE CAClients
SET DATABASE TO CAClients
errorCode = SQLEXEC("SELECT Company, City ;
    FROM Company WHERE State_Prov='CA'", "StateCA.DBF")
IF errorCode = 0
    SET DATABASE TO
    USE StateCa
    LIST
ENDIF
RETURN

PROCEDURE ErrHndlr
PARAMETERS nErrorNo, cErrMess, nSQLErrorNo, cSqlErrMess, cProgram, nLineNo
DEFINE FORM HeadsUp FROM 10,20 TO 20,55;
    PROPERTY Text "Heads Up"
DEFINE TEXT Line1 OF HeadsUp AT 2,10 ;
    PROPERTY Text "An Error has occurred", Width 24, ColorNormal "R+/W"
DEFINE TEXT Line2 OF HeadsUp AT 4,2;
    PROPERTY Text ;
    IIF(ERROR()=240,cSqlErrMess,cErrMess), Width 33
DEFINE TEXT Line3 OF HeadsUp AT 5,2;
    PROPERTY Text "Number: " + ;
    IIF(ERROR()=240,STR(nSQLErrorNo),STR(nErrorNo)), Width 24
DEFINE TEXT Line4 OF HeadsUp AT 6,2;
    PROPERTY Text "Program: " + cProgram, Width 22
DEFINE TEXT Line5 OF HeadsUp AT 7,2;
    PROPERTY Text "Line #: " + STR(nLineNo), Width 22
OPEN FORM HeadsUp
```

See Also CERROR(), DBERROR(), DBMESSAGE(), ERROR(), MESSAGE(), ON ERROR, RETRY, SQLMESSAGE()

SQLEXEC()

Executes an SQL statement in the current database or on specified dBASE or Paradox tables.

Syntax SQLEXEC(<SQL statement expC> [,<Answer table expC>])

<SQL statement expC> A character string that contains an SQL statement. The SQL statement must follow server-specific dialect rules for the current database and must be enclosed in quotes. For Paradox and *Visual* dBASE tables, the dialect is the same as that used by the Borland InterBase database server, which is ANSI-compliant. Character strings and SQL or BDE reserved words contained within the SQL statement must also be enclosed in either single or double quotation marks. (Single quotation marks are normally used.).

<Answer table expC> Paradox or DBF table that stores the data returned by an SQL SELECT statement; must also be in quotes. If you specify a file without including its path, *Visual* dBASE creates the file in the current directory, then in the path you specify with SET PATH. If you specify a file without including its extension, *Visual* dBASE assumes the default table type specified with the SET DBTYPE command. If you don't specify a table name, *Visual* dBASE creates a table named Answer with the extension defined by the current DBTYPE setting.

You can also specify the name of an already open database (defined for a file directory location only) as a prefix (enclosed in colons) to the name of the answer table, that is, *:database name:table name*. You cannot specify the location of an answer table on a database server.

Description SQLEXEC() executes a SQL statement in the current database set by SET DATABASE, or if a database is not set, on tables in the current or a specified directory. (You can preface the name of a table with its directory location or specify an already open database by enclosing the database name in colons, for example, *:database name:table name*. If you're using Borland SQL Link to connect to a database server, *Visual* dBASE passes the SQL statement you specify directly to the database server where the database selected by SET DATABASE resides.

When an SQL statement contains SQL or BDE reserved words and you are executing the statement on DBF or Paradox tables, you need to enclose the reserved words in single(') or double (") quotation marks and use SQL table aliases (different than the aliases associated with dBASE tables) to identify fields, for example:

```
SELECT * FROM company.dbf b WHERE b."CHAR" = 'element'
```

You can use table aliases to qualify fields specified in the SELECT, WHERE, GROUP BY, or ORDER BY clauses of SELECT statements. This is particularly useful when querying data from more than one table.

SQLEXEC() returns error codes with the same values as those returned by ERROR() and MESSAGE(); a value of zero indicates that no error occurred as a result of the statement's execution. If an error occurs, you can use DBERROR() and DBMESSAGE() functions to return BDE errors or use the SQLERROR() and SQLMESSAGE() functions to obtain information directly from the database server about the cause of an error. (Also, the ERROR() function returns an error code of 240 if a server error occurs.)

SQLMESSAGE()

Example The following example executes an SQL SELECT statement on the server table Company:

```
SET DBTYPE TO DBASE
OPEN DATABASE CAClients
SET DATABASE TO CAClients
errorCode = SQLEXEC("SELECT Company, City ;
FROM Company WHERE State_Prov='CA'", "StateCA.DBF")
IF errorCode = 0
SET DATABASE TO
USE StateCa
LIST
ENDIF
RETURN
```

See Also DBERROR(), DBMESSAGE(), ERROR(), MESSAGE(), OPEN DATABASE, SET DATABASE, SET DBTYPE, SET PATH, SQLERROR(), SQLMESSAGE()

SQLMESSAGE()

Returns the most recent server error message.

Syntax SQLMESSAGE()

Description Use SQLMESSAGE() to determine the error message of the last server error. To learn the error code, use SQLERROR().

See online Help for a listing of all error messages.

Example See SQLERROR()

See Also CERROR(), DBERROR(), DBMESSAGE(), ERROR(), MESSAGE(), ON ERROR, RETRY, SQLERROR()

SUSPEND

Suspends program execution, temporarily returning control to the Command window.

Syntax SUSPEND

Description SUSPEND lets you interrupt program execution at a specific point, a *break point*. The program remains suspended until you issue RESUME or CANCEL, or until you exit *Visual dBASE*. If you issue RESUME, the program resumes from the break point. If you issue CANCEL, *Visual dBASE* cancels program execution.

While a program is suspended, you can enter commands in the Command window. For example, you can check and change the status of files, memory variables, SET commands, and so on; however, *Visual dBASE* ignores any

changes you make to the program while it is suspended. If you want to correct a suspended program, issue CANCEL, edit the program, and then run it again.

If you initialize memory variables in the Command window while a program is suspended, *Visual* dBASE makes them private at the program level that suspension occurred.

You should not return to a suspended program by issuing DO <filename> in the Command window. If you do so, you will end up with "nested" SUSPEND statements, and may not know that a program is still suspended. If you want to run a suspended program from the beginning, issue CANCEL and then DO <filename>.

The *Visual* dBASE Debugger allows for more complex break points than using SUSPEND.

Example The following program prompts the entry of a 2-letter state abbreviation and lists the clients within that state. If the program fails to return a list of clients, the programmer might insert the SUSPEND command just after the second CLEAR to halt the program so that trouble shooting commands could be issued at the Command window such as: ? mState to determine the value in the variable mState, LIST FOR STATE_PROV = "CA", DISPLAY MEMORY, DISPLAY STATUS. Issue the command RESUME when ready to proceed with the remainder of the program:

```
CLEAR
SET TALK OFF
USE CLIENTS
ACCEPT "Enter 2 letter State abbreviation: " TO mState
CLEAR
SUSPEND                                && To be removed after troubleshooting
? CENTER("Clients in "+UPPER(mState))
?
SCAN FOR State_Prov = UPPER(mState)
? Company, Contact, Startbal
ENDSCAN
RETURN
```

See Also CANCEL, DEBUG, DO, RESUME, QUIT

USER()

Returns the login name of the user currently logged in to a protected system.

Syntax USER()

Description The USER() function returns the log-in name used by an operator currently logged in to a system that uses PROTECT to encrypt files. On a system that does not use PROTECT, USER() returns an empty string.

See Also ACCESS(), PROTECT

VERSION()

Returns the name and version number of the currently running copy of *Visual* dBASE.

Syntax VERSION([<expN>])

<expN> Any number, which causes VERSION() to return extended version information.

Description Although you may be able to use VERSION() in programs to take advantage of version-specific features, the most common use of VERSION() is to get the exact build number of your copy of *Visual* dBASE to see if you have the latest build. When called with no parameters, VERSION() returns a string like:

```
Visual dBASE 7.0
```

with the product name and the version number. If you pass a number, for example VERSION(1), you will get extended build information, like:

```
Visual dBASE 7.0 b1298 (-US970930)
```

which adds the build number after the “b” and the identifier and date of the language resource for that copy of *Visual* dBASE. If you pass the number .89, you will get the build information for the Borland Database Engine used, for example,

```
BDE version: 09/17/97
```

If you are running a *Visual* dBASE executable, the word "Runtime" appears in the string; for example:

```
Visual dBASE 7.0 Runtime
Visual dBASE 7.0 Runtime b1298 (-US970930)
```

Example Suppose you have a class that represents your application. It has a method that checks if the string returned by VERSION() contains the word "Runtime" to determine whether the application is being run in the IDE workbench or as a compiled application:

```
function isRuntime()
    return ( "RUNTIME" $ upper( version() ) )
```

In the method that terminates the application, you either quit or restore the IDE workbench (by calling other methods in the class not shown here):

```
function shutdown()
    if this.isRuntime()
        quit
    else
        this.unloadProcFiles()
        this.restoreWorkbench()
        this.unhookGlobalErrorHandler()
    endif
```

See Also OS()

Preprocessor

When *Visual* dBASE compiles a program file, that file is run through the *preprocessor* before it is actually compiled. The preprocessor is a separate built-in utility that processes the text of the program file to prepare it for compilation. Its duties include

- Stripping out comments from the program file
- Joins lines separated by the line continuation character
- Substituting macro-identifiers and macro-functions with the corresponding replacement text
- Including the text of other files in the program file
- Conditionally excluding parts of the program file so they are not compiled

The preprocessor generates an intermediate file; this is the file that is compiled by *Visual* dBASE's compiler.

While those are the mechanics of the preprocessor, the usage of the preprocessor allows you to

- Replace constants and “magic numbers” in your code with easy-to-read and easy-to-change identifiers
- Create macro-functions to replace complex expressions with parameters
- Use collections of constant identifiers and macro-functions in multiple program files
- Maintain separate versions of your programs, for example debug and production versions, in the same program files through conditional compilation

Visual dBASE's preprocessor is similar to the preprocessor used in the C language. It uses a handful of *preprocessor directives* to control its activities. All preprocessor directives start with the # character and each one must be on its own, single line in the script file.

#define

Defines an identifier (name) for use in controlling program compilation, defining constants, or creating macro-functions.

Syntax `#define <identifier> [<replacement text>]`
 `#define <identifier>(<parameter list>) <replacement text with parameters>`

<identifier> A name. It identifies the text to replace if *<replacement text>* is supplied. The name must start with an alphabetic character and can contain any combination of alphabetic or numeric characters, uppercase or lowercase letters. The identifier is not case-sensitive.

(<parameter list>) Parameter names that correspond to arguments passed to a macro-function that you create with `#define <identifier>(<parameter list>) <replacement text>`. If you specify multiple parameters, separate each with a comma. There cannot be any spaces between the *<identifier>* and the opening parenthesis of (*<parameter list>*), or after any of the parameter names in the *<parameter list>*.

<replacement text> The text you want to use to replace all occurrences of *<identifier>*. If you specify *<replacement text>*, the preprocessor scans each source code line for identifiers and replaces each one it encounters with the specified replacement text. *<replacement text>* can be any text that is part of a dBASE program, such as a string, numeric expression, or series of commands.

Description The `#define` directive defines an identifier and optionally lets you replace text in a program before compilation. Each `#define` definition must begin on a new line and is limited to 4096 characters.

Identifiers are available only to the program in which they are defined. To define identifiers for use in multiple programs, place them in a separate file and use `#include` to include that file as needed.

You must define an identifier in a file with the `#define` directive before you can use it. Once it has been defined, you cannot `#define` it again; you must undefine it first with the `#undef` preprocessor directive.

Use the `#define` directive for the following purposes:

- To declare an identifier and assign replacement text to represent a constant value or a complex expression.
- To create a macro-function.
- To declare an identifier with no replacement text, so you can use it with the `#ifdef` or `#ifndef` directive.
- To declare an identifier with replacement text, so you can use it with the `#if` directive.

The effect of `#define` is similar to a word processor's search-and-replace feature. When the preprocessor encounters a `#define` identifier in the text of a script, it

replaces that identifier with the *<replacement text>*. If there is no *<replacement text>* for that identifier, the identifier is simply removed. For example:

```
#define test 4                // Create identifier with value
? test - 3                    // ( 4 - 3 ) = 1
#undef test                    // #undef to change definition
#define test                    // Create identifier with no value
? test - 3                    // ( - 3 ) = -3
```

Because the preprocessor runs before a program is compiled and performs simple text substitution, the use of `#define` statements can in effect override memory variables, built-in commands and functions, and any other element having the same name as *<identifier>*. This is shown in the following examples.

```
// Overriding a variable
somevar = 25;                // Creates variable
#define somevar 10;          // Until further notice, "somevar" will be replaced
? somevar                    // Compiles argument as "10". Displays 10
#undef somevar                // "somevar" no longer replaced
? somevar                    // Displays 25

// Overriding a function
#define cos(x) (42 + x)       // Function adds 42
? cos(3)                     // Compiles argument as "(42 + 3)". Displays 45
```

To use `#define` directives in WFM and REP files generated by the Form and Report designers, place the directives in the Header section of the file so that the definitions will not be erased by the designer.

Declaring identifiers to represent constants Assign an identifier to represent a constant value or expression when you want the preprocessor to search for and replace all instances of the identifier with the specified value or expression before compilation. When used in this manner, the identifier is known as a *manifest constant*. It's common practice to make the name of the manifest constant all uppercase, with underscores between words so that it stands out in code. For example:

```
#define SECS_PER_HOUR 3600    // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR) // Number of milliseconds per day
```

Note that when using a manifest constant to represent a numeric expression, you should place the entire expression inside parentheses. This prevents possible errors due to the precedence of operators used to evaluate expressions. For example, consider the following calculation:

```
nDays = nTimeElapsed / MSECS_PER_DAY
```

Without parentheses, this statement would compile as:

```
nDays = nTimeElapsed / 1000*24*3600
```

That's incorrect—it divides by 1000 then multiplies by 24 and 3600. (The multiplication and division operators are at the same level of precedence, so the expression is evaluated from left to right.). By placing parentheses around the manifest constant definition as shown, the statement would compile as:

```
nDays = nTimeElapsed / (1000*24*3600)
```

Because of the parentheses, the expression is evaluated correctly: the value of the constant is evaluated first, then used as the divisor.

Manifest constants streamline your code and improve its readability because you can use a single identifier to represent a frequently used constant or a complex expression. In addition, if you need to change the value of a constant in your program, you need to change only the constant definition and not every occurrence of the constant.

To replace an identifier only in parts of a program, insert `#undef <identifier>` into your program where you want the search-and-replace process to stop.

Creating macro-functions When the preprocessor encounters a function call that matches a defined macro-function, it replaces the function call with the replacement text, inserting the arguments of the function call into the replacement text. This is shown in the following example.

```
#define avg(num1,num2) (((num1)+(num2))/2)  // Average two numbers
:
n1=20
n2=40
? avg( n1, n2 ) // Displays 30
```

The arguments in the macro-function call are substituted exactly as they are shown in the macro-function definition. In this example, the last statement compiles as:

```
? (((n1)+(n2))/2)
```

When using a macro-function to perform calculations, always use parentheses to enclose each parameter and the entire expression in the macro-function definition as shown. If you leave them out, errors may result due to the precedence of operators, as shown in these (somewhat contrived) examples:

```
#define avg(num1,num2) (((num1)+(num2))/2)
#define badAvg(num1,num2) (num1+num2)/2
? 12 / avg( 2, 4 ) // 12/(6/2) --> displays 4
? 12 / badavg( 2, 4 ) // 12/6/2 --> displays 1
```

Unlike functions in dBASE, the number of arguments passed from a macro-function call must match the number of parameters defined in your `#define` statement.

Declaring identifiers for conditional compilation In addition to using identifiers for constants and macro-functions in dBASE code, they are used for conditional compilation with the `#if`, `#ifdef`, and `#ifndef` directives.

Defining an identifier without replacement text lets you use it with the `#ifdef` or `#ifndef` directive to test if the identifier exists. When used in this manner, the existence of the identifier acts as a logical flag to either include or exclude code for compilation.

When an identifier is defined with replacement text, you can use comparison operators to check the value of the identifier in an `#if` directive, and conditionally compile code based on the result. You can also use `#ifdef` and `#ifndef` to test for the existence of the identifier.

Nesting preprocessor identifiers You can nest preprocessor identifiers; that is, the replacement text for one identifier may contain other identifiers, as long as those identifiers are already defined, as shown in the following example:

```
#define SECS_PER_HOUR 3600           // Number of seconds per hour
#define MSECS_PER_DAY (1000*24*SECS_PER_HOUR) // Number of milliseconds per day
```

You cannot use the identifier being defined in its replacement text, however.

Example The first example uses a manifest constant to represent a “magic number.” Suppose your application is doing metric conversions. Instead of sprinkling the conversion factor around in your code, which would just be a cryptic number, you can #define it as a manifest constant, which eliminates the possibility that you might get the number wrong in some places and makes your code self-documenting:

```
#define LB_PER_KG 2.2046 // Number of pounds per kilogram
:
nPounds = form.kg.value * LB_PER_KG
```

The second example uses a manifest constant to represent a simple constant in your application. Suppose you’re testing several different techniques to see which one accomplishes the same task the fastest. You need to repeat the task many times to get measurable results, so you use a manifest constant to represent the number of times you want each test to be run. By using a single manifest constant, you can easily change the number of times each test is run and calculate the average time:

```
#define NUM_REPS 10000 // Number of times to repeat each test
:
for n = 1 to NUM_REPS
    // Test 1
endfor
for n = 1 to NUM_REPS
    // Test 2
endfor
:
? "Average time for test 1", time[1] / NUM_REPS
```

The following example uses a manifest constant for a file name that is used in different parts of an application:

```
#define QWK_FILE      "IMF.QWK"
#define MESSAGE_FILE "MESSAGES.DAT"
:
fMsg = new File()
fMsg.create( MESSAGE_FILE )
:
z = new ZipFile( QWK_FILE ) // Create compressed file
z.store( MESSAGE_FILE )    // Store the message file
:
class ZipFile( cFileName ) of File
    // Code to implement ZipFile class
endclass
```

`#else`

This example demonstrates conditional compilation. Two preprocessor identifiers are used: a `DEBUG` flag, and a `BUILD` number:

```
#define DEBUG    // Comment out if not debug version
#define BUILD 35 // Current build number
:
#if BUILD < 20
    // Older code
#else
    // Current code
    #ifdef DEBUG
        // Include DEBUG code
    #endif
#endif
```

See also `#if`, `#ifdef`, `#ifndef`, `#include`, `#undef`

`#else`

Designates an alternate block of code to conditionally compile if the condition specified by an `#if`, `#ifdef`, or `#ifndef` directive is *false*.

Description See `#if`, `#ifdef`, and `#ifndef` for details.

`#endif`

Designates the end of an `#if`, `#ifdef`, or `#ifndef` directive.

Description See `#if`, `#ifdef`, and `#ifndef` for details.

`#if`

Controls conditional compilation of code based on the value of an identifier assigned with `#define`.

Syntax `#if <condition>`
`<statements 1>`
`[#else`
`<statements 2>]`
`#endif`

<condition> A logical expression, using an identifier you've defined, that evaluates to *true* or *false*.

<statements 1> Any number of statements and preprocessor directives. These lines are compiled if `<condition>` evaluates to *true*.

#else <statements 2> Specifies the lines you want to compile if <condition> evaluates to *false*.

Description Use the #if directive to conditionally compile sections of source code based on the value of an identifier. Two other directives, #ifdef and #ifndef, are also used to conditionally include or exclude code for compilation. Unlike the #if directive, however, they test only for the existence of an identifier, not for its value.

The <condition> must be a simple logical condition; that is, a single test using one basic comparison operator (=, ==, <, >, <=, >=, <>). You may nest conditional compilation directives.

If the identifier is not defined, the <condition> always evaluates to *false*.

Conditional compilation is useful when maintaining different versions of the same program, for debugging, and for managing the use of #include files. Using #if for conditional compilation is different than conditionally executing code with an IF statement. With IF, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #if to exclude code you don't want for a particular version of your program, the code is never compiled into byte code.

When *Visual* dBASE's preprocessor processes a file, it internally defines the preprocessor identifier __vdb__ (two underscores on both ends) with the current version number. Earlier versions of *Visual* dBASE used __dbasewin__. Use these two built-in identifiers to manage code that's intended to run on different versions of *Visual* dBASE.

Example The following example demonstrates how you would create code that runs on different versions of *Visual* dBASE, using the built-in identifiers __vdb__ and __dbasewin__:

```
#if __vdb__ >= 7
    // Version 7 code
#else
    #if __dbasewin__ > 5
        && Version 5.5 or 5.6 code
    #else
        && Version 5.0 code
    #endif
#endif
```

Because code that is excluded by #if is never compiled, you can safely use new syntax that might be introduced in a new version. When compiled with an older version of *Visual* dBASE, the new code is ignored. This is different than testing the version returned by the VERSION() function at run time. New syntax would not compile under an older version.

Note that for the pre-version 7 code, the older comment style is used.

See also #define, #ifdef, #ifndef

#ifdef

Controls conditional compilation of code based on the existence of an identifier created with #define.

Syntax `#ifdef <identifier>`
 `<statements 1>`
 `[#else`
 `<statements 2>]`
 `#endif`

<identifier> The identifier you want to test for. *<identifier>* is defined with the #define directive.

<statements 1> Any number of statements and preprocessor directives. These lines are compiled if *<identifier>* has been defined.

#else <statements 2> Specifies the lines to compile if *<identifier>* has not been defined.

Description Use the #ifdef directive to conditionally compile sections of source code. If you've defined *<identifier>* with #define, the code you specify with *<statements 1>* is compiled; otherwise, the code following #else, if any, is compiled.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same program, for debugging purposes, and for managing the use of #include files. Using #ifdef for conditional compilation is different than not executing code with an IF statement. With IF, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #ifdef to exclude code you don't want for a particular version of your program, the code is never compiled into byte code.

Example The following example uses #ifdef to check for a identifier named DEBUG to determine if extra code should be included to display trace information in the results pane of the Command window:

```
#define DEBUG      // Comment out if not debug version
#ifdef DEBUG
    #define TRACE(m) ? m
#else
    #define TRACE(m)
#endif
:
// Process names in list
if form.rowset.first()
do
    TRACE( form.rowset.fields[ "Last name" ].value ) // Display name as we go
    // Do whatever
until not form.rowset.next()
```

endif

The macro-function TRACE() is defined so that if the DEBUG identifier is not defined, all calls to TRACE() are replaced with nothing—they are removed from the code and not compiled. This allows you to use TRACE() as much as you want, and with a simple change in the DEBUG identifier, remove all the code from the compiled byte code, resulting in better performance.

See also #define, #if, #ifndef

#ifndef

Controls conditional compilation of code based on the existence of an identifier assigned with #define.

Syntax `#ifndef <identifier>
 <statements 1>
 [#else
 <statements 2>]
 #endif`

<identifier> The identifier you want to test for. *<identifier>* is defined with the #define directive.

<statements 1> Any number of statements and preprocessor directives. These lines are compiled if *<identifier>* has not been defined.

#else <statements 2> Specifies the lines to compile if *<identifier>* has been defined.

Description Use the #ifndef directive to conditionally compile sections of source code. If you haven't defined *<identifier>* with #define, the code you specify with *<statements 1>* is compiled; otherwise, the code following #else, if any, is compiled.

Use #ifndef if you want to include code only if the identifier is not defined. Otherwise, you can use #ifdef to include code only if the identifier is defined, and #ifdef with its #else option to include different sets of code depending on the existence of the identifier.

You may nest conditional compilation directives.

Conditional compilation is useful when maintaining different versions of the same program, for debugging purposes, and for managing the use of #include files. Using #ifndef for conditional compilation is different than not executing code with an IF statement. With IF, the code still gets compiled into the resulting byte code file, even if it is never executed. By using #ifndef to exclude code you don't want for a particular version of your program, the code is never compiled into byte code.

Example When creating a set of #define directives in an #include file, enclose the entire set inside an #ifndef block and #define a special identifier for that block. For example, here are some lines from the VDBASE.H file that includes #define directives for many of the enumerated values used in *Visual* dBASE:

```
#ifndef VDBASE_H
#define VDBASE_H

// Constants for MSGBOX()
#define OK_BUTTON 0
#define OK_CANCEL_BUTTONS 1
#define ABORT_RETRY_IGNORE_BUTTONS 2
#define YES_NO_CANCEL_BUTTONS 3
#define YES_NO_BUTTONS 4
#define RETRY_CANCEL_BUTTONS 5

#endif
```

If you #include the same file twice in the same program file (which often happens because some #include files #include other files), the #ifndef directive will make sure that #define directives are processed only once. Attempting to #define the same identifier twice causes an error.

See also #define, #if, #ifdef

#include

Inserts the contents of the specified source file (known as an *include* or *header* file) into the current program file at the location of the #include statement.

Syntax #include <filename> | "<filename>"

<filename> | "<filename>" The name of the file, optionally including a full or partial path, whose contents are to be inserted into the current program file. You can specify the file name within or without quotes. An include file typically has an .h file-name extension.

If you specify <filename> without a path, the preprocessor uses the following search order:

- 1 It searches the current directory for the file exactly as you've specified it.
- 2 If you omitted the .h file-name extension, it adds the extension and searches the current directory.
- 3 If it can't find the file in the current directory, it looks in <home directory>\INCLUDE. (The home directory is the one in the _DBWINHOME system memory variable.)
- 4 If it can't find the file in the current directory or <home directory>\INCLUDE, it looks in the directory you specify with the DOS environment variable INCLUDE.

Description The effect of `#include` is as if the contents of the specified file were typed into the current program file at the location of the `#include` statement. The specified file is called an include file. `#include` is used primarily for files which have `#define` directives.

Identifiers are available only to the program in which they are defined. To use a single set of identifiers in multiple programs, save the `#define` statements in a file, then use the `#include` directive to define the identifiers in additional programs.

An advantage of having all the `#define` statements in one file is the ease of maintenance. If you need to modify any of the `#define` statements, you need only change the include file; the program files that use the `#define` statements remain unchanged. After you modify the include file, recompile your program file for the changes to take effect.

To use `#include` directives in WFM and REP files generated by the Form and Report designers, place the directives in the Header section of the file so that the definitions will not be erased by the designer.

Example You may want to set up a standard include file that you use in all your script files that contains manifest constants and macro-functions that you use through your application. For example:

```
// Std.h
#include "VDBASE.H"           // Contains #defines for enums
#define CONFIRM(m,t) (msgbox(m,t,4+32)==BUTTON_YES)
```

Place the STD.H file in *Visual dBASE's* INCLUDE subdirectory so that it's easily accessible. Then at the top of every program, `#include` that file:

```
#include "STD.H"
:
if CONFIRM( "This record will be lost forever! You sure?", "Delete" )
```

See also `#define`

#pragma

Sets compiler options.

Syntax `#pragma <compiler option>`
<compiler option> The compiler option to set.

Description Use the `#pragma` to set compiler options. The only option supported in this version of *Visual dBASE* is:

coverage(ON | OFF) Enables or disables the inclusion of coverage analysis information in the resulting byte code file.

Coverage analysis provides information about which program lines are executed. To provide coverage analysis, a program file must be compiled to include the extra coverage analysis information.

SET COVERAGE controls whether programs are compiled with coverage information. Use `#pragma coverage()` in your program file to override the SET COVERAGE setting for that particular file.

Note Do not specify both `#pragma coverage(ON)` and `#pragma coverage(OFF)` in the same program file. The last `#pragma` takes effect; all others are ignored.

For more information about coverage files, see SET COVERAGE.

Example The following example uses `#pragma` to enable coverage analysis if the `DEBUG` constant has been `#defined`:

```
#ifdef DEBUG
    #pragma coverage(on)
#else
    #pragma coverage(off)
#endif
```

See Also `#define`, SET COVERAGE

#undef

Removes the current definition of the specified identifier previously defined with `#define`.

Syntax `#undef <identifier>`

<identifier> The identifier whose definition you want to remove.

Description The `#undef` directive removes the definition of an identifier previously defined with the `#define` directive. If you use `#define` with *<replacement text>*, the preprocessor replaces all instances of the identifier with the replacement text from the point it encounters that `#define` until it encounters an `#undef` specifying the same identifier. Therefore, to replace an identifier only in parts of a program, insert `#undef <identifier>` into your program where you want the search-and-replace process to stop.

`#undef` is also required if you want to change the *<replacement text>* for an identifier. You cannot use `#define` for an identifier that is already defined. You must `#undef` the identifier first, then specify a new `#define` directive.

Attempting to `#undef` an identifier that is not defined has no effect; no error is generated.

Example In this example, the script file has numerous `#ifdef DEBUG` statements to conditionally compile debug code. You want to use the debug code for only one section in the file, so you `#define DEBUG` at the beginning of the section, and `#undef DEBUG` at the end:


```

:
#define DEBUG
// Some code
#ifdef DEBUG
    // Debug code
#endif
// More code
#undef DEBUG
:
// Some more code
#ifdef DEBUG
    // More debug code
#endif

```

See also `#define`

__vdb__

Identifies the current *Visual* dBASE version number.

Description When *Visual* dBASE's preprocessor processes a file, it internally defines the preprocessor identifier `__vdb__` (two underscores on both ends) with the current version number. Earlier versions of *Visual* dBASE used `__dbasewin__`. Use these two built-in identifiers to manage code that's intended to run on different versions of *Visual* dBASE.

Example The following example demonstrates how you would create code that runs on different versions of *Visual* dBASE, using the built-in identifiers `__vdb__` and `__dbasewin__`:

```

#if __dbasewin__ >= 7
    #if __vdb__ < 8
        // Version 7 or 7.5 code
    #else
        // Later version code
    #endif
#else
    #if __dbasewin__ > 5
        && Version 5.5 or 5.6 code
    #else
        && Version 5.0 code
    #endif
#endif

```

Because code that is excluded by `#if` is never compiled, you can safely use new syntax that might be introduced in a new version. When compiled with an older version of *Visual* dBASE, the new code is ignored. This is different than testing the version returned by the `VERSION()` function at run time. New syntax would not compile under an older version.

Note that for the pre-version 7 code, the older comment style is used.

See also `#if`



ASCII character chart (code page 437)

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|--------|-----|-----|---------|-----|-----|------|-----|-----|------|
| 0 | 00 | <null> | 32 | 20 | <space> | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 01 | A | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 02 | B | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 03 | C | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 04 | D | 36 | 24 | \$ | 68 | 44 | D | 100 | 64 | d |
| 5 | 05 | E | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 06 | F | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 07 | G | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 08 | H | 40 | 28 | (| 72 | 48 | H | 104 | 68 | h |
| 9 | 09 | I | 41 | 29 |) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | J | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | K | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | L | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | M | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | N | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | O | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | P | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Q | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | R | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | S | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | T | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | U | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | V | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | W | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | X | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | Y | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Z | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | [| 59 | 3B | ; | 91 | 5B | [| 123 | 7B | { |
| 28 | 1C | \ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | |
| 29 | 1D |] | 61 | 3D | = | 93 | 5D |] | 125 | 7D | } |
| 30 | 1E | ^ | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | _ | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | |

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|------|-----|-----|------|-----|-----|------|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | À | 224 | E0 | à |
| 129 | 81 | ü | 161 | A1 | í | 193 | C1 | Á | 225 | E1 | á |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | Â | 226 | E2 | â |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | Ã | 227 | E3 | ã |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | Ä | 228 | E4 | ä |
| 133 | 85 | à | 165 | A5 | Ñ | 197 | C5 | Å | 229 | E5 | å |
| 134 | 86 | â | 166 | A6 | ı | 198 | C6 | Æ | 230 | E6 | æ |
| 135 | 87 | ç | 167 | A7 | § | 199 | C7 | Ç | 231 | E7 | ç |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | È | 232 | E8 | è |
| 137 | 89 | ë | 169 | A9 | © | 201 | C9 | É | 233 | E9 | é |
| 138 | 8A | è | 170 | AA | ª | 202 | CA | Ê | 234 | EA | ê |
| 139 | 8B | ï | 171 | AB | « | 203 | CB | Ë | 235 | EB | ë |
| 140 | 8C | î | 172 | AC | ¬ | 204 | CC | Ì | 236 | EC | ì |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | Í | 237 | ED | í |
| 142 | 8E | Ä | 174 | AE | ® | 206 | CE | Î | 238 | EE | î |
| 143 | 8F | Å | 175 | AF | ˆ | 207 | CF | Ï | 239 | EF | ï |
| 144 | 90 | É | 176 | B0 | ° | 208 | D0 | Ð | 240 | F0 | ≡ |
| 145 | 91 | æ | 177 | B1 | ± | 209 | D1 | Ñ | 241 | F1 | ± |
| 146 | 92 | Æ | 178 | B2 | ² | 210 | D2 | Ò | 242 | F2 | ≥ |
| 147 | 93 | ô | 179 | B3 | ³ | 211 | D3 | Ó | 243 | F3 | ≤ |
| 148 | 94 | ö | 180 | B4 | ´ | 212 | D4 | Ô | 244 | F4 | ∫ |
| 149 | 95 | ò | 181 | B5 | µ | 213 | D5 | Õ | 245 | F5 | ∫ |
| 150 | 96 | û | 182 | B6 | ¶ | 214 | D6 | Ö | 246 | F6 | ÷ |
| 151 | 97 | ù | 183 | B7 | · | 215 | D7 | × | 247 | F7 | ≈ |
| 152 | 98 | ÿ | 184 | B8 | ¸ | 216 | D8 | Ø | 248 | F8 | ° |
| 153 | 99 | Ö | 185 | B9 | ¹ | 217 | D9 | Ù | 249 | F9 | • |
| 154 | 9A | Ü | 186 | BA | º | 218 | DA | Ú | 250 | FA | • |
| 155 | 9B | › | 187 | BB | » | 219 | DB | Û | 251 | FB | √ |
| 156 | 9C | £ | 188 | BC | ¼ | 220 | DC | Ü | 252 | FC | n |
| 157 | 9D | ¥ | 189 | BD | ½ | 221 | DD | Ý | 253 | FD | ² |
| 158 | 9E | ₤ | 190 | BE | ¾ | 222 | DE | Þ | 254 | FE | Ý |
| 159 | 9F | f | 191 | BF | ¿ | 223 | DF | ß | 255 | FF | |

B

File structures

This appendix provides information on the structure of the dBASE table (.DBF) and the memo (.DBT) files.

Note Table specifications for *Visual* dBASE apply to level 5 tables only.

Table header and records

A dBASE table file (.DBF) is composed of a header, data records, deletion flags, and an end-of-file marker. The header contains information about the file structure, and the records contain the actual data. One byte of each record is reserved for the deletion flag.

Table header structure

The header structure, detailed in Table B.1 and Table B.2, provides information that *Visual* dBASE uses to maintain the table file.

Table B.1 dBASE table file header

| Byte | Contents | Description |
|-------|---------------|---|
| 0 | 1 byte | Valid <i>Visual</i> dBASE table file; bits 0–2 indicate version number; bit 3 indicates presence of a dBASE IV or <i>Visual</i> dBASE memo file; bits 4–6 indicate the presence of a dBASE IV SQL table; bit 7 indicates the presence of any .DBT memo file (either a dBASE III PLUS type or a dBASE IV or <i>Visual</i> dBASE memo file) |
| 1–3 | 3 bytes | Date of last update; in YYMMDD format |
| 4–7 | 32-bit number | Number of records in the table |
| 8–9 | 16-bit number | Number of bytes in the header |
| 10–11 | 16-bit number | Number of bytes in the record |

Table B.1 dBASE table file header (continued)

| Byte | Contents | Description |
|------------|---------------|--|
| 12–13 | 2 bytes | Reserved; filled with zeros |
| 14 | 1 byte | Flag indicating incomplete dBASE IV transaction ¹ |
| 15 | 1 byte | dBASE IV encryption flag ² |
| 16–27 | 12 bytes | Reserved for multi-user processing |
| 28 | 1 byte | Production .MDX flag; 01H stored in this byte if a production .MDX file exists for this table; 00H stored if no .MDX file exists |
| 29 | 1 byte | Language driver ID |
| 30–31 | 2 bytes | Reserved; filled with zeros |
| 32 – n^3 | 32 bytes each | Field descriptor array (the structure of this array is shown in Table B-2). |
| $n + 1$ | 1 byte | 0DH stored as the field terminator |

1. Flag not used by *Visual* dBASE; in dBASE IV, BEGIN TRANSACTION sets this flag to 01H, END TRANSACTION and ROLLBACK resets it to 00H.
2. Encryption not supported in *Visual* dBASE; in dBASE IV, if flag is set to 01H, the table is encrypted.
3. n is the last byte in the field descriptor array. The size of the array depends on the number of fields in the table file.

Table B.2 Table field descriptor bytes

| Byte | Contents | Description |
|-------|----------|--|
| 0–10 | 11 bytes | Field name in ASCII (zero-filled) |
| 11 | 1 byte | Field type in ASCII (B, C, D, F, G, L, M, or N) |
| 12–15 | 4 bytes | Reserved |
| 16 | 1 byte | Field length in binary |
| 17 | 1 byte | Field decimal count in binary |
| 18–19 | 2 bytes | Reserved |
| 20 | 1 byte | Work area ID |
| 21–30 | 10 bytes | Reserved |
| 31 | 1 byte | Production .MDX field flag; 01H if field has an index tag in the production .MDX file; 00H if field is not indexed |

Table records

The records follow the header in the table file. Data records are preceded by one byte, that is, a space (20H) if the record is not deleted, an asterisk (2AH) if the record is deleted. Fields are packed into records without field separators or record terminators. The end of the file is marked by a single byte, with the end-of-file marker, an OEM Code Page character value of 26 (1AH). You can input OEM code page data as indicated in Table B.3.

Table B.3 Allowable input for dBASE data types

| Data type | Data input |
|------------------------------------|---|
| B (Binary) | All OEM code page characters (stored internally as 10 digits representing a .DBT block number) |
| C (Character) | All OEM code page characters |
| D (Date) | Numbers and a character to separate month, day, and year (stored internally as 8 digits in YYYYMMDD format) |
| G (General or OLE) | All OEM code page characters (stored internally as 10 digits representing a .DBT block number) |
| N (Numeric) and F (Floating Point) | – . 0 1 2 3 4 5 6 7 8 9 |
| L (Logical) | ? Y y N n T t F f (? when not initialized) |
| M (Memo) | All OEM code page characters (stored internally as 10 digits representing a .DBT block number) |

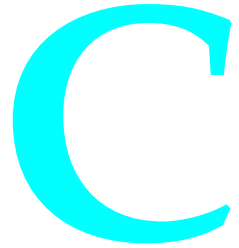
Binary, memo, and OLE fields and .DBT files

Binary, memo, and OLE fields store data in .DBT files consisting of blocks numbered sequentially (0, 1, 2, and so on). SET BLOCKSIZE determines the size of each block. The first block in the .DBT file, block 0, is the .DBT file header.

Each binary, memo, or OLE field of each record in the .DBF file contains the number of the block (in OEM code page values) where the field's data actually begins. If a field contains no data, the .DBF file contains blanks (20H) rather than a number.

When data is changed in a field, the block numbers may also change and the number in the .DBF may be changed to reflect the new location.

Unlike dBASE III PLUS, if you delete text in a memo field (or binary and OLE fields), *Visual* dBASE (like dBASE IV) may reuse the space from the deleted text when you input new text. dBASE III PLUS always appends new text to the end of the .DBT file. In dBASE III PLUS, the .DBT file size grows whenever new text is added, even if other text in the file is deleted.



Error codes

Table C.1 Error codes and associated messages

| Code | Message |
|------|---|
| 1 | Error creating file |
| 2 | Error opening file |
| 3 | Error closing file |
| 4 | End of table encountered |
| 5 | Record out of range |
| 6 | Error positioning in file |
| 7 | File does not exist |
| 8 | File already exists |
| 9 | File already open |
| 10 | Unable to rename file |
| 11 | Structure invalid |
| 12 | Invalid COV file |
| 13 | dBASE IV binary report file not supported - use component builder to convert it |
| 14 | Invalid label file |
| 15 | dBASE IV binary label file not supported - use component builder to convert it |
| 16 | Invalid memory variable file |
| 17 | Invalid PRO / FMO file |
| 18 | Invalid query file |
| 19 | Invalid report file |
| 20 | Invalid driver name or insufficient system resources |
| 21 | Invalid view file |
| 22 | Invalid window file |
| 23 | Operation not allowed for calculated fields |
| 24 | Operation not allowed on read-only files |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|--|
| 25 | Bad field name |
| 26 | Bad field type |
| 28 | Duplicate field name |
| 29 | Error writing file |
| 30 | Not a valid dBASE table |
| 31 | No such record in index |
| 32 | Illegal key |
| 33 | WindowMenu must be on child of MenuBar |
| 34 | No table in use in area |
| 35 | Table is not indexed |
| 36 | Maximum number of fields reached |
| 40 | Field not found |
| 41 | Cyclic RELATION not allowed |
| 42 | Work area already used in relation |
| 44 | Too many RELATIONS in this chain |
| 47 | Too many index files |
| 48 | Invalid order number |
| 49 | No fields were found to process |
| 50 | Field must be a memo field |
| 51 | Field must be a binary field |
| 53 | Tag not found |
| 54 | Unrecognized command verb |
| 55 | Command too large |
| 56 | Expression expected |
| 57 | Expression too large |
| 58 | Too few arguments. Expecting at least |
| 59 | Too many arguments. Expecting at most |
| 61 | Unterminated string |
| 62 | Unbalanced parentheses |
| 63 | Syntax error |
| 67 | Something is missing. Expecting |
| 68 | Unknown keyword |
| 70 | PARAMETERS command must be at top of function or procedure |
| 71 | Invalid name character |
| 74 | ENDIF encountered without preceding IF |
| 75 | Missing ENDIF |
| 76 | ENDDO encountered without preceding WHILE |
| 77 | Missing ENDDO |
| 78 | NEXT encountered without preceding FOR |
| 79 | Missing NEXT |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|---|
| 80 | ENDSCAN encountered without preceding SCAN |
| 81 | Missing ENDSCAN |
| 82 | UNTIL encountered without preceding DO |
| 83 | Missing UNTIL |
| 84 | ENDCASE encountered without preceding CASE |
| 85 | Missing ENDCASE |
| 86 | ENDPRINTJOB command encountered without previous PRINTJOB command |
| 87 | Missing ENDPRINTJOB |
| 88 | ENDCLASS/PROTECT command encountered without previous CLASS command |
| 89 | Missing ENDCLASS |
| 90 | ENDTEXT command encountered without previous TEXT command |
| 91 | Missing ENDTEXT |
| 94 | Loop stack overflow. |
| 95 | Too many nested FOR loops. |
| 96 | Too many nested SCAN loops. |
| 97 | Unallowed phrase/keyword in command |
| 98 | Keyword Repeated |
| 100 | UDF must return a value |
| 101 | Too many dimensions |
| 102 | Too many UDF/PROCEDURES defined in program |
| 103 | Invalid FUNCTION or PROCEDURE name |
| 104 | Invalid CLASS name |
| 105 | Invalid MEMBER name |
| 106 | Program too big to compile |
| 107 | Not enough memory for this operation |
| 108 | In use by another |
| 109 | Record is in use by another |
| 110 | Command requires exclusive use of table |
| 111 | Memory variable space exhausted |
| 112 | Not enough memory for DOS |
| 114 | Filename space exhausted |
| 115 | Only valid in program files |
| 116 | No PARAMETERS statement found |
| 117 | Unmatched parameters |
| 118 | Program not SUSPENDED |
| 119 | No such bar |
| 120 | No such menu |
| 121 | No such pad |
| 122 | No such popup |
| 124 | No such window |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|-------------|--|
| 125 | No such window |
| 126 | No such form object |
| 127 | Menu already active |
| 128 | Popup already active |
| 129 | Unable to add data while constraints active |
| 130 | Windows print file name longer than 31 characters |
| 131 | Printer is either not connected or turned off |
| 132 | Window out of range |
| 133 | Unauthorized access level |
| 134 | No bars defined for popup/pulldown |
| 135 | Bars already defined for popup. |
| 136 | Bars must have a positive value. |
| 139 | Cannot release active |
| 140 | Datasource/Prompt cannot be MEMO/OLE/BINARY |
| 142 | Cannot change property while form is open |
| 143 | Expecting reference to MENU object |
| 144 | First class in menu file is not derived from MENU |
| 145 | Cannot have more than one form object with the same name |
| 146 | Internal stack overflow |
| 147 | Internal stack underflow |
| 148 | Stack overflow |
| 149 | Runtime buffer overflow |
| 150 | String buffer overflow |
| 151 | Attempt to free a bad memory block |
| 152 | Attempt to load a bad icode block |
| 153 | Macros cannot expand flow of control commands |
| 154 | Expanded macro variable does not return a valid identifier |
| 155 | Cannot assign to reserved word NULL |
| 156 | Numbers are not allowed in the CURRENCY symbol |
| 157 | Illegal work area number or alias |
| 158 | Illegal value |
| 159 | UDF or procedure already exists |
| 160 | Unable to execute DOS |
| 161 | Too many nested expressions |
| 162 | Nested views not allowed |
| 163 | Data type mismatch. Expecting |
| 164 | Out of range |
| 167 | Variable undefined |
| 168 | Not an array |
| 169 | Illegal Opcode |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|--|
| 172 | Maximum number of nested FOR NEXT loops exceeded |
| 175 | Maximum number of DO or UDF parameters exceeded |
| 178 | Alias not found |
| 179 | MEMO field not allowed here |
| 180 | ALIAS already in use |
| 181 | Processing would exceed maximum allowed string length |
| 182 | Procedure not found |
| 185 | Illegal file name |
| 186 | Beginning of table encountered |
| 187 | Error reading file |
| 188 | Unexpected type |
| 189 | Printer error |
| 190 | Memory variable already defined - cannot make PUBLIC |
| 191 | CONTINUE without previous LOCATE |
| 192 | Value out of range |
| 193 | Invalid subscript reference |
| 196 | Invalid printer redirection |
| 197 | Cannot execute this command when DESIGN is off |
| 198 | Command not functional in this release of dBASE for Windows |
| 199 | Restricted command: not allowed in this context |
| 200 | Command will never be reached |
| 201 | Command not functional in dBASE for Windows |
| 202 | Extra characters ignored at end of command |
| 203 | Program was previously compiled with SET COVERAGE OFF |
| 206 | Drive not ready |
| 207 | UDF or PROCEDURE not found |
| 209 | Too many files open |
| 210 | Invalid directory |
| 211 | Invalid disk drive |
| 212 | Cannot redefine active menu |
| 214 | No such listbox |
| 215 | Window not active |
| 217 | Symbol table space exhausted. Increase to |
| 218 | SET FIELDS space exhausted |
| 219 | No previous DO WHILE/SCAN/PRINTJOB/FOR to match this command |
| 221 | Too many nested DO/UDF |
| 222 | Maximum number of locked records exceeded |
| 223 | Sharing buffers are full |
| 224 | Error unlocking file |
| 225 | Unmatched #else |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|---|
| 226 | Unmatched #endif |
| 227 | Maximum #ifdef nesting exceeded |
| 228 | Expecting #endif |
| 229 | Preprocessor expansion too large |
| 230 | Include file not found |
| 231 | Table already open |
| 232 | Database already open |
| 234 | Operation not allowed in transaction |
| 236 | Operation not allowed on this table |
| 237 | Index is not open |
| 239 | IDAPI Error |
| 240 | Server Error |
| 241 | Database not opened |
| 242 | Invalid value for convert size (8-24) |
| 243 | Invalid file Handle |
| 244 | IDAPI Not Initialized |
| 245 | Cannot UPDATE a table with itself |
| 246 | Invalid Catalog |
| 247 | Invalid password |
| 248 | Access denied |
| 249 | Can only change draft mode on page boundaries |
| 250 | Can only change page orientation on page boundaries |
| 251 | Already in printjob |
| 252 | Wrong version of IDAPI01.DLL |
| 253 | No print driver selected |
| 254 | No pads defined for |
| 255 | AUTOEXTERN not supported for this database |
| 256 | Output parameter required |
| 257 | Attempting to call a method as a function |
| 258 | Method is not available on object |
| 259 | No Records Selected |
| 260 | Internal Exception Error |
| 261 | Stored procedures not supported |
| 262 | Form cannot be MDI |
| 265 | Resource not found |
| 266 | Cannot load print driver |
| 267 | Cannot paste more than one file |
| 268 | Cannot recognize dropped file |
| 269 | Cannot Paste selection |
| 270 | Cannot Copy selection |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|--|
| 271 | Cannot Package file |
| 272 | Cannot activate object. OLE server is busy |
| 273 | Cannot update linked object |
| 274 | Unknown error saving window contents |
| 275 | Cannot perform operation on static object |
| 276 | Error connecting to OLE server. May be bad object path if a link. |
| 277 | Invalid command verb for OLE object |
| 278 | OLE object error |
| 279 | OLE BLOB field is corrupted |
| 280 | OLE BLOB field data is from an incompatible version |
| 281 | Attempt to access released object |
| 282 | Property is read only |
| 283 | only 1 or 2 dimensional array is allowed in this operation |
| 284 | Report Engine Error |
| 285 | Property not found |
| 286 | Operation not allowed on read-only fields |
| 287 | Operation not allowed on read-only tables |
| 288 | An Editor or Viewer of a memo field is still open |
| 289 | Not member of Class or Base class |
| 290 | SUPER not allowed when THIS is undefined |
| 291 | Unable to open form |
| 292 | Unable to create control |
| 293 | No such form |
| 294 | The system registry does not contain an OLE server for a file with extension |
| 295 | OLE: cannot create link |
| 298 | Constant is already #defined |
| 299 | Field must be an OLE field |
| 301 | Position not in window |
| 302 | Invalid Color |
| 303 | Parameter type '...' can only be used with CDECL calling convention |
| 304 | DLL does not support Multiple Instances |
| 305 | Error loading DLL |
| 306 | Extern |
| 307 | Parent is not a REPORT |
| 308 | Parent is not a PAGETEMPLATE or BANDBODY |
| 309 | Error Saving VBX |
| 310 | BINARY field not allowed here |
| 311 | OLE field not allowed here |
| 312 | Popup too small |
| 313 | Error creating palette |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|-------------|--|
| 314 | OLE Error |
| 315 | only 1 dimensional array is allowed in this operation |
| 316 | Incomplete link specification |
| 317 | Selected tables cannot be related |
| 318 | Too many symbols in this module |
| 319 | Cannot create directory |
| 320 | Invalid table name |
| 321 | Invalid preprocessor identifier |
| 322 | DLL not Loaded |
| 323 | Invalid key label |
| 324 | Only allowed in function or procedure scope |
| 325 | Not a valid table |
| 326 | Port not configured for a printer |
| 327 | #includes nested too deeply |
| 328 | Index expressions not allowed for INTEGRITY rules |
| 329 | Related records still exist in alias |
| 330 | SET KEY active in alias |
| 331 | Relation using CONSTRAIN |
| 332 | No matching parent record |
| 333 | Operation not allowed across different databases or table types |
| 334 | Key already exists in parent |
| 335 | First class in .WFM file is not derived from FORM |
| 336 | Relation expression and active index expression must be the same |
| 337 | Memo file does not exist |
| 338 | Production index file does not exist |
| 339 | Invalid file privileges |
| 340 | Form already open |
| 341 | Error reading from binary field |
| 342 | Must convert report before modifying |
| 343 | Class does not exist |
| 344 | Fix or remove errors before running query |
| 345 | PRIMARY must start with first field |
| 346 | Fields must be in consecutive order |
| 347 | Report writer has not been installed |
| 348 | VBX dlls cannot be RELEASED |
| 349 | VBStream file Missing or Corrupt |
| 350 | Cannot JOIN table with itself |
| 351 | Cannot assign to reserved word THIS |
| 352 | OLE Unknown interface |
| 353 | OLE Member Not found |

Table C.1 Error codes and associated messages (continued)

| Code | Message |
|------|--|
| 354 | OLE Parameter Not found |
| 355 | OLE Data Type Mismatch |
| 356 | OLE Unknown name |
| 357 | OLE No Named arguments |
| 358 | OLE Bad Variable Type |
| 359 | OLE Dispatch Exception |
| 360 | OLE Overflow |
| 361 | OLE Invalid Subscript |
| 362 | OLE Unknown Class |
| 363 | OLE Array is locked |
| 364 | OLE Bad parameter count |
| 365 | OLE Parameter not optional |
| 366 | OLE Bad call |
| 367 | OLE Not a collection |
| 368 | OLE Unknown error |
| 369 | OLE Object does not support automation |
| 370 | OLE Unable to create object |
| 371 | OLE Class name not in registry |
| 372 | In use by another |
| 373 | Record is in use by another |
| 374 | Property is not accessible |

Table C.2 Error codes in alphabetical order

| Message | Code |
|--|------|
| #includes nested too deeply | 327 |
| Access denied | 248 |
| ALIAS already in use | 180 |
| Alias not found | 178 |
| Already in printjob | 251 |
| An Editor or Viewer of a memo field is still open | 288 |
| Attempt to access released object | 281 |
| Attempt to free a bad memory block | 151 |
| Attempt to load a bad icode block | 152 |
| Attempting to call a method as a function | 257 |
| AUTOEXTERN not supported for this database | 255 |
| Bad field name | 25 |
| Bad field type | 26 |
| Bars already defined for popup. | 135 |
| Bars must have a positive value. | 136 |
| Beginning of table encountered | 186 |
| BINARY field not allowed here | 310 |
| Can only change draft mode on page boundaries | 249 |
| Can only change page orientation on page boundaries | 250 |
| Cannot activate object. OLE server is busy | 272 |
| Cannot assign to reserved word NULL | 155 |
| Cannot assign to reserved word THIS | 351 |
| Cannot change property while form is open | 142 |
| Cannot Copy selection | 270 |
| Cannot create directory | 319 |
| Cannot execute this command when DESIGN is off | 197 |
| Cannot have more than one form object with the same name | 145 |
| Cannot JOIN table with itself | 350 |
| Cannot load print driver | 266 |
| Cannot Package file | 271 |
| Cannot paste more than one file | 267 |
| Cannot Paste selection | 269 |
| Cannot perform operation on static object | 275 |
| Cannot recognize dropped file | 268 |
| Cannot redefine active menu | 212 |
| Cannot release active | 139 |
| Cannot UPDATE a table with itself | 245 |
| Cannot update linked object | 273 |
| Class does not exist | 343 |
| Command not functional in dBASE for Windows | 201 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|---|------|
| Command not functional in this release of dBASE for Windows | 198 |
| Command requires exclusive use of table | 110 |
| Command too large | 55 |
| Command will never be reached | 200 |
| Constant is already #defined | 298 |
| CONTINUE without previous LOCATE | 191 |
| Cyclic RELATION not allowed | 41 |
| Data type mismatch. Expecting | 163 |
| Database already open | 232 |
| Database not opened | 241 |
| Datasource/Prompt cannot be MEMO/OLE/BINARY | 140 |
| dBASE IV binary label file not supported - use component builder to convert it | 15 |
| dBASE IV binary report file not supported - use component builder to convert it | 13 |
| DLL does not support Multiple Instances | 304 |
| DLL not Loaded | 322 |
| Drive not ready | 206 |
| Duplicate field name | 28 |
| End of table encountered | 4 |
| ENDCASE encountered without preceding CASE | 84 |
| ENDCLASS/PROTECT command encountered without previous CLASS command | 88 |
| ENDDO encountered without preceding WHILE | 76 |
| ENDIF encountered without preceding IF | 74 |
| ENDPRINTJOB command encountered without previous PRINTJOB command | 86 |
| ENDSCAN encountered without preceding SCAN | 80 |
| ENDTEXT command encountered without previous TEXT command | 90 |
| Error closing file | 3 |
| Error connecting to OLE server. May be bad object path if a link. | 276 |
| Error creating file | 1 |
| Error creating palette | 313 |
| Error loading DLL | 305 |
| Error opening file | 2 |
| Error positioning in file | 6 |
| Error reading file | 187 |
| Error reading from binary field | 341 |
| Error Saving VBX | 309 |
| Error unlocking file | 224 |
| Error writing file | 29 |
| Expanded macro variable does not return a valid identifier | 154 |
| Expecting #endif | 228 |
| Expecting reference to MENU object | 143 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|---|------|
| Expression expected | 56 |
| Expression too large | 57 |
| Extern | 306 |
| Extra characters ignored at end of command | 202 |
| Field must be a binary field | 51 |
| Field must be a memo field | 50 |
| Field must be an OLE field | 299 |
| Field not found | 40 |
| Fields must be in consecutive order | 346 |
| File already exists | 8 |
| File already open | 9 |
| File does not exist | 7 |
| Filename space exhausted | 114 |
| First class in .WFM file is not derived from FORM | 335 |
| First class in menu file is not derived from MENU | 144 |
| Fix or remove errors before running query | 344 |
| Form already open | 340 |
| Form cannot be MDI | 262 |
| IDAPI Error | 239 |
| IDAPI Not Initialized | 244 |
| Illegal file name | 185 |
| Illegal key | 32 |
| Illegal Opcode | 169 |
| Illegal value | 158 |
| Illegal work area number or alias | 157 |
| In use by another | 108 |
| In use by another | 372 |
| Include file not found | 230 |
| Incomplete link specification | 316 |
| Index expressions not allowed for INTEGRITY rules | 328 |
| Index is not open | 237 |
| Internal Exception Error | 260 |
| Internal stack overflow | 146 |
| Internal stack underflow | 147 |
| Invalid Catalog | 246 |
| Invalid CLASS name | 104 |
| Invalid Color | 302 |
| Invalid command verb for OLE object | 277 |
| Invalid COV file | 12 |
| Invalid directory | 210 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|--|------|
| Invalid disk drive | 211 |
| Invalid driver name or insufficient system resources | 20 |
| Invalid file Handle | 243 |
| Invalid file privileges | 339 |
| Invalid FUNCTION or PROCEDURE name | 103 |
| Invalid key label | 323 |
| Invalid label file | 14 |
| Invalid MEMBER name | 105 |
| Invalid memory variable file | 16 |
| Invalid name character | 71 |
| Invalid order number | 48 |
| Invalid password | 247 |
| Invalid preprocessor identifier | 321 |
| Invalid printer redirection | 196 |
| Invalid PRO / FMO file | 17 |
| Invalid query file | 18 |
| Invalid report file | 19 |
| Invalid subscript reference | 193 |
| Invalid table name | 320 |
| Invalid value for convert size (8-24) | 242 |
| Invalid view file | 21 |
| Invalid window file | 22 |
| Key already exists in parent | 334 |
| Keyword Repeated | 98 |
| Loop stack overflow. | 94 |
| Macros cannot expand flow of control commands | 153 |
| Maximum #ifdef nesting exceeded | 227 |
| Maximum number of DO or UDF parameters exceeded | 175 |
| Maximum number of fields reached | 36 |
| Maximum number of locked records exceeded | 222 |
| Maximum number of nested FOR NEXT loops exceeded | 172 |
| MEMO field not allowed here | 179 |
| Memo file does not exist | 337 |
| Memory variable already defined - cannot make PUBLIC | 190 |
| Memory variable space exhausted | 111 |
| Menu already active | 127 |
| Method is not available on object | 258 |
| Missing ENDCASE | 85 |
| Missing ENDCLASS | 89 |
| Missing ENDDO | 77 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|--|------|
| Missing ENDIF | 75 |
| Missing ENDPRINTJOB | 87 |
| Missing ENDSCAN | 81 |
| Missing ENDTEXT | 91 |
| Missing NEXT | 79 |
| Missing UNTIL | 83 |
| Must convert report before modifying | 342 |
| Nested views not allowed | 162 |
| NEXT encountered without preceding FOR | 78 |
| No bars defined for popup/pulldown | 134 |
| No fields were found to process | 49 |
| No matching parent record | 332 |
| No pads defined for | 254 |
| No PARAMETERS statement found | 116 |
| No previous DO WHILE/SCAN/PRINTJOB/FOR to match this command | 219 |
| No print driver selected | 253 |
| No Records Selected | 259 |
| No such bar | 119 |
| No such form | 293 |
| No such form object | 126 |
| No such listbox | 214 |
| No such menu | 120 |
| No such pad | 121 |
| No such popup | 122 |
| No such record in index | 31 |
| No such window | 124 |
| No such window | 125 |
| No table in use in area | 34 |
| Not a valid dBASE table | 30 |
| Not a valid table | 325 |
| Not an array | 168 |
| Not enough memory for DOS | 112 |
| Not enough memory for this operation | 107 |
| Not member of Class or Base class | 289 |
| Numbers are not allowed in the CURRENCY symbol | 156 |
| OLE Array is locked | 363 |
| OLE Bad call | 366 |
| OLE Bad parameter count | 364 |
| OLE Bad Variable Type | 358 |
| OLE BLOB field data is from an incompatible version | 280 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|---|------|
| OLE BLOB field is corrupted | 279 |
| OLE Class name not in registry | 371 |
| OLE Data Type Mismatch | 355 |
| OLE Dispatch Exception | 359 |
| OLE Error | 314 |
| OLE field not allowed here | 311 |
| OLE Invalid Subscript | 361 |
| OLE Member Not found | 353 |
| OLE No Named arguments | 357 |
| OLE Not a collection | 367 |
| OLE Object does not support automation | 369 |
| OLE object error | 278 |
| OLE Overflow | 360 |
| OLE Parameter Not found | 354 |
| OLE Parameter not optional | 365 |
| OLE Unable to create object | 370 |
| OLE Unknown Class | 362 |
| OLE Unknown error | 368 |
| OLE Unknown interface | 352 |
| OLE Unknown name | 356 |
| OLE: cannot create link | 295 |
| only 1 dimensional array is allowed in this operation | 315 |
| only 1 or 2 dimensional array is allowed in this operation | 283 |
| Only allowed in function or procedure scope | 324 |
| Only valid in program files | 115 |
| Operation not allowed across different databases or table types | 333 |
| Operation not allowed for calculated fields | 23 |
| Operation not allowed in transaction | 234 |
| Operation not allowed on read-only fields | 286 |
| Operation not allowed on read-only files | 24 |
| Operation not allowed on read-only tables | 287 |
| Operation not allowed on this table | 236 |
| Out of range | 164 |
| Output parameter required | 256 |
| Parameter type '...' can only be used with CDECL calling convention | 303 |
| PARAMETERS command must be at top of function or procedure | 70 |
| Parent is not a PAGETEMPLATE or BANDBODY | 308 |
| Parent is not a REPORT | 307 |
| Popup already active | 128 |
| Popup too small | 312 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|--|------|
| Port not configured for a printer | 326 |
| Position not in window | 301 |
| Preprocessor expansion too large | 229 |
| PRIMARY must start with first field | 345 |
| Printer error | 189 |
| Printer is either not connected or turned off | 131 |
| Procedure not found | 182 |
| Processing would exceed maximum allowed string length | 181 |
| Production index file does not exist | 338 |
| Program not SUSPENDED | 118 |
| Program too big to compile | 106 |
| Program was previously compiled with SET COVERAGE OFF | 203 |
| Property is not accessible | 374 |
| Property is read only | 282 |
| Property not found | 285 |
| Record is in use by another | 109 |
| Record is in use by another | 373 |
| Record out of range | 5 |
| Related records still exist in alias | 329 |
| Relation expression and active index expression must be the same | 336 |
| Relation using CONSTRAIN | 331 |
| Report Engine Error | 284 |
| Report writer has not been installed | 347 |
| Resource not found | 265 |
| Restricted command: not allowed in this context | 199 |
| Runtime buffer overflow | 149 |
| Selected tables cannot be related | 317 |
| Server Error | 240 |
| SET FIELDS space exhausted | 218 |
| SET KEY active in alias | 330 |
| Sharing buffers are full | 223 |
| Something is missing. Expecting | 67 |
| Stack overflow | 148 |
| Stored procedures not supported | 261 |
| String buffer overflow | 150 |
| Structure invalid | 11 |
| SUPER not allowed when THIS is undefined | 290 |
| Symbol table space exhausted. Increase to | 217 |
| Syntax error | 63 |
| Table already open | 231 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|--|------|
| Table is not indexed | 35 |
| Tag not found | 53 |
| The system registry does not contain an OLE server for a file with extension | 294 |
| Too few arguments. Expecting at least | 58 |
| Too many arguments. Expecting at most | 59 |
| Too many dimensions | 101 |
| Too many files open | 209 |
| Too many index files | 47 |
| Too many nested DO/UDF | 221 |
| Too many nested expressions | 161 |
| Too many nested FOR loops. | 95 |
| Too many nested SCAN loops. | 96 |
| Too many RELATIONs in this chain | 44 |
| Too many symbols in this module | 318 |
| Too many UDF/PROCEDURES defined in program | 102 |
| UDF must return a value | 100 |
| UDF or procedure already exists | 159 |
| UDF or PROCEDURE not found | 207 |
| Unable to add data while constraints active | 129 |
| Unable to create control | 292 |
| Unable to execute DOS | 160 |
| Unable to open form | 291 |
| Unable to rename file | 10 |
| Unallowed phrase/keyword in command | 97 |
| Unauthorized access level | 133 |
| Unbalanced parentheses | 62 |
| Unexpected type | 188 |
| Unknown error saving window contents | 274 |
| Unknown keyword | 68 |
| Unmatched #else | 225 |
| Unmatched #endif | 226 |
| Unmatched parameters | 117 |
| Unrecognized command verb | 54 |
| Unterminated string | 61 |
| UNTIL encountered without preceding DO | 82 |
| Value out of range | 192 |
| Variable undefined | 167 |
| VBStream file Missing or Corrupt | 349 |
| VBX dlls cannot be RELEASED | 348 |
| Window not active | 215 |

Table C.2 Error codes in alphabetical order (continued)

| Message | Code |
|---|------|
| Window out of range | 132 |
| WindowMenu must be on child of MenuBar | 33 |
| Windows print file name longer than 31 characters | 130 |
| Work area already used in relation | 42 |
| Wrong version of IDAPI01.DLL | 252 |

Index

Symbols

- ! command 11-5
 - RUN vs. 11-35
- " "symbol 4-16
- # operator 4-8
- # symbol 4-19
- \$ operator 4-8
- % operator 4-6
- & operator 4-13
- && symbol 4-17
- () operator 4-12
- * (asterisk)
 - in fields 12-78
 - pattern matching 6-16
- * operator 4-6
- ** operator 4-6
- + operator 4-4, 4-6
- ++ operator 4-6
- . (dot) operator 4-11
- / operator 4-6
- /* */ symbol 4-17
- // symbol 4-17
- :: operator 4-11
- : = operator 4-3
- ; symbol 4-17, 16-39
- < operator 4-8
- < = operator 4-8
- < > operator 4-8
- = operator 4-3, 4-8
- == operator 4-8
- > operator 4-8
- > = operator 4-8
- ? (question mark)
 - pattern matching 6-16
 - temporary files 11-21
- ? command 18-1
 - ON PAGE and 18-9
 - SET PRINTER and 18-18
 - SET SPACE and 18-20
- ?? command 18-5
 - ? command vs. 18-2
 - ON PAGE and 18-9
 - SET ALTERNATE and 18-14
 - SET PRINTER and 18-18
 - SET SPACE and 18-20
- [] operator 4-10
- ^ operator 4-6
- { } symbols 4-18
- operator 4-6
- operator 4-6

- " symbol 4-16
- __dbasewin__ 22-13
 - #if and 22-13
- __vdb__ 22-13
 - #if and 22-13

A

- abandon() method 14-31
- abandoned rowsets 14-95
- abandoning data changes 14-31, 14-33
- abandonRecord()
 - method 15-55
- abandonUpdates()
 - method 14-33
- abbreviating keywords 5-43
- ABS() 7-1
- absolute values
 - defined 7-1
 - returning 7-1
- accelerators 16-43
- access rights, assigning 11-28
- ACCESS() 21-1
- access() method 14-33
- accessDate() method 11-6
- accessing
 - alternate text editors 20-28
 - client/server
 - applications 19-27
 - OLE 15-90, 15-162
 - data 12-21, 14-1 to 14-3
 - error handling 14-11
 - multiuser
 - environments 21-10
 - file-sharing
 - modes 12-95
 - setting locks 12-48, 12-64, 12-84
 - read-only
 - restrictions 12-97, 12-102
 - specific fields 12-95
 - tables 15-56
- ACOPY() 10-7
- ACOS() 7-1
 - RTOD() and 7-14
- actions
 - See also* events
 - executing 15-38
 - recurring, setting 8-4
- activating online Help 20-19
- activating the Debugger 20-13
- active indexes, returning 12-116
- active property 14-34
- activeControl property 15-56
- ActiveX class 15-3
 - properties (table) 15-4
- ActiveX controls
 - setting use properties 15-73
- ActiveX objects (defined) 15-4
- add() method 10-8
- adding bitmaps to
 - backgrounds 15-63
- adding fields 20-22
- adding passwords 14-35
- adding records 12-5, 15-64
 - arrays and 12-8
 - event handling 15-126
 - local SQL 13-13
 - restricting 15-62
 - temporary 15-158
 - to rowsets 14-36, 14-37, 14-43
- addition 12-19, 12-114, 12-117
- addition operators 4-4, 4-5, 4-6
- addPassword() method 14-35
- advise() method 19-11
 - unadvise() and 19-28
- agAverage() method 17-13
- agCount() method 17-14
- aggregate calculations
 - highest value 17-15
 - lowest value 17-16
 - mean average 17-13
 - number of items 17-14
 - standard deviation 17-16
 - total 17-17
 - variance 17-18
- aggregate functions, local SQL 13-5
- aggregation 12-19, 12-114, 12-117
 - values, calculating 17-7
- agMax() method 17-15
- agMin() method 17-16
- agStandardDeviation()
 - method 17-16
- agSum() method 17-17
- agVariance() method 17-18
- ALen() 10-9
- alerts 20-23
- alias operator 4-12, 12-107

- alias property 15-56
- ALIAS() 12-3
- aliases
 - See also* BDE aliases
 - field names 12-78
 - tables 15-56
 - linking 12-107
 - work areas 12-2, 12-3, 12-91
- alignHorizontal property 15-57
- _alignment 18-20
 - _wrap and 18-39
- alignment
 - text 15-57, 15-58
- alignment property 15-57
- alignVertical property 15-58
- ALTER TABLE statement (SQL) 13-8
 - ADD clause and 13-8
 - DROP clause and 13-8
- alternate text editors 20-6
 - accessing 20-28
- alternate text files 18-13
- anchor property 15-61
- anchoring objects 15-61
- AND operator 4-7
 - bitwise 9-1
- angles
 - arccosecant 7-2
 - arccosine 7-1
 - arccotangent 7-3
 - arcsecant 7-2
 - arcsine 7-2
 - arctangent 7-2, 7-3
 - converting
 - degrees to radians 7-4
 - radians to degrees 7-14
 - cosecant 7-19
 - cosine 7-4
 - cotangent 7-20
 - measuring 7-5
 - secant 7-4
 - sine 7-19
 - tangent 7-20
- annotating code 4-17
- ANSI conversions 21-2
- ANSI date format 8-23
- ANSI() 21-1
 - OEM() and 21-11
- Answer tables 21-21
 - _app 16-1
- _app.frameWin 16-2
- APPEND 12-4
 - KEYMATCH() and 12-59
- APPEND AUTOMEM 12-5
- APPEND BLANK 12-4
 - BLANK vs. 12-14
- APPEND FROM 12-7
- APPEND FROM ARRAY 12-8
- APPEND MEMO 12-10
 - REPLACE MEMO vs. 12-83
- Append mode
 - attempted, handling
 - event 14-52
 - entered 14-96
- append property 15-62
- append() method 14-36
- appending data 14-36, 14-37, 14-43
 - key violations,
 - handling 14-78
 - to BLOB fields 14-110
- appendUpdate() method 14-37
- application classes
 - Menu 16-3
 - MenuBar 16-6
 - Popup 16-8
 - ToolBar 16-9
 - ToggleButton 16-11
- applications
 - closing 5-34
 - DDE *See* DDE server
 - applications
 - external 15-90, 19-16, 19-23
 - MDI 15-117
 - sound 15-91
 - stand-alone 15-23
 - standalone (using SHELL) 16-42
- applyFilter() method 14-37
- applyLocate() method 14-38
- applyUpdates() method 14-39
- arccosecant, returning 7-2
- arccosine, returning 7-1
- arccotangent, returning 7-3
- arcsecant, returning 7-2
- arcsine, returning 7-2
- arctangent, returning 7-2, 7-3
- ARGCOUNT() method 5-3
- arguments
 - automem variables 12-80
 - color 16-15
 - expressions, described 3-1
- ARGVECTOR() method 5-4
- arithmetic operations 4-6
 - mean, returning 12-11, 12-18
 - remainders, returning 7-10
 - type conversions 6-32
- arithmetic operators
 - assignment 4-3
 - local SQL 13-3
- Array class 10-2
 - methods (table) 10-3
 - properties (table) 10-3
- array classes
 - Array 10-2
 - AssocArray 10-5
- array elements
 - adding to arrays 10-8, 10-22, 10-25
 - addressing 15-93
 - copying 10-7
 - deleting 10-10
 - all 10-30
 - specified 10-30
 - finding next 10-29
 - inserting 10-25
 - literal 4-18
 - number of
 - changing 10-36
 - determining 10-36
 - numbers 10-4
 - returning 10-18
 - referencing 4-11, 10-40
 - returning 10-9
 - searching for key 10-28
 - sorting 10-37
 - storing specified
 - values 10-20
 - subscripts 10-4
 - finding 10-40
- array functions
 - ACOPY() 10-7
- array index operator 4-10
- array methods
 - count() 10-9
 - delete() 10-10
 - dir() 10-14
 - dirExt() 10-16
 - element() 10-18
 - fields() 10-19
 - fill() 10-20
 - grow() 10-22
 - insert() 10-25
 - isKey() 10-28
 - nextKey() 10-29
 - removeAll() 10-30
 - removeKey() 10-30
 - resize() 10-31
 - scan() 10-35
 - sort() 10-37
 - subscript() 10-40
- Array objects
 - defined 10-4
 - dimensions
 - finding number of 10-13

- overview 10-1 to 10-7
- arrays 12-8
 - adding rows and columns 10-25
- associative
 - looping through 10-22
 - overview 10-6
 - returning number of elements 10-9
- columns
 - adding 10-22
 - finding how many 10-9
- copying data 12-30, 12-81
- defined 2-8
- dimensions
 - changing 10-32
 - finding how many 10-32
- elements *See* array elements
- expressions
 - finding 10-35
 - storing 5-45
- Field objects, in rowset 14-69
- file information 10-14, 10-16
- initializing values 10-20
- nested
 - accessing values 10-5
 - defined 10-4
- object references 15-93
- one-dimensional 10-8
- ragged, defined 10-4
- rows
 - adding 10-22
 - finding how many 10-9
 - sorting 10-37
- size, changing 10-22, 10-31
- storing values 12-11, 12-19, 12-114
- table structures 10-19
- two-dimensional
 - creating 10-23
 - number of columns 10-9
 - number of rows 10-9
- arrow keys
 - command execution 16-32
 - input focus 15-104
- asc() method 6-4
- ASC() 6-4
 - CHR() vs. 6-7
- ascending sort order 12-53, 12-112
- ASCII values, returning 6-4, 6-7, 6-8
- ASIN() 7-2
 - RTOD() and 7-14
- assigning keystrokes

- command execution 16-29, 16-31, 16-38
- interrupts 16-37
- assigning values 4-11
 - to properties 5-10
- assignment operators 4-3
- assignment-only operator 4-3
- AssocArray class 10-5
 - methods (table) 10-6
 - properties (table) 10-6
- asterisk (*)
 - in fields 12-78
 - pattern matching 6-16
- AT() 6-5
 - RAT() and 6-18
- ATAN() 7-2
 - ATN2() vs. 7-3
 - RTOD() and 7-14
- atFirst() method 14-40
- atLast() method 14-41
- ATN2() 7-3
 - ATAN() vs. 7-3
 - RTOD() and 7-14
- attach method 16-13
- autoCenter property 15-62
- autoDrop property 15-62
- autoEdit property 14-42
- automatic compiling 20-27
- automatic file locks 12-48, 12-61, 12-85, 14-15
 - disabling 12-102
- automatic record locks 14-15
- automatically saving data 12-92
- automatically updating
 - indexes 12-56
- automem variables
 - arguments 12-80
 - clearing 12-76
 - creating 12-21, 12-113
 - defined 12-6
 - storing data 12-113
- autoSize property 15-63
- autosizing forms 15-63
- autoSort property 17-19
- AVERAGE 12-11
- average, returning mean 17-13
- averages, returning 12-11, 12-18, 13-5
- AVG() function (SQL) 13-5

B

- background property 15-63
- backgrounds
 - adding bitmaps 15-63
 - blended (hatched) 15-146

- colors 15-75
 - image, setting 15-63
- backup files 20-22
- Band class 17-5
- Band objects
 - automatic creation 17-6
 - constant size 17-23
 - detail 17-6
 - expanding 17-23
 - footer 17-6
 - header 17-6
 - rendering data in 17-20
- BDE (defined) 14-1
- BDE aliases
 - assigning 14-63
 - not required 14-1
 - required 14-1
 - restrictions 14-5
 - Standard tables and portability 14-5
- BDE errors 14-10, 21-4, 21-5
- BDE functions, calling 14-76
- beeps 20-23
- before property 15-63
- beforeGetValue event 14-42
- beginAppend() method 14-43, 15-64
- beginEdit() method 14-45
- beginFilter() method 14-46
- beginLocate() method 14-46
- beginNewFrame
 - property 17-20
- beginning-of-file
 - indicator 12-15
- begins with operator 4-8
- beginTrans() method 14-47
- BEGINTRANS() 12-12
- binary data types
 - combining 12-26
 - returning 12-13
 - user-defined 12-26
- binary data, defined 2-5
- binary fields
 - changing 12-80
 - copying 12-26
 - determining type 12-13
 - sound effects 19-22
- binary files
 - coverage analysis 20-25
 - creating 12-26
 - reading from 12-80
- binary operators
 - numeric 4-6
 - unary 2-5
- BINTYPE() 12-13

- BITAND() 9-1
- BITLSHIFT() 9-2
- bitmaps 12-26, 15-27
 - See also* images
 - adding to
 - backgrounds 15-63
 - icons, on forms 15-110
 - on pushbuttons 15-90, 15-91, 15-100, 15-176
 - supported formats 15-28
- BITOR() 9-4
- BITRSHIFT() 9-4
- BITSET() 9-5
- bitwise functions 9-1
- bitwise operators 9-5
 - AND 9-1
 - OR 9-4
 - shift bits 9-2
 - XOR 9-5
- BITXOR() 9-5
- BLANK 12-14
- blank fields 12-14
- blank lines, suppressing 17-41
- blank records 12-4
 - averaging numeric fields 12-11, 12-114
- blank values 12-20
- blended (hatched)
 - backgrounds 15-146
- BLOB fields
 - appending to 14-110
 - copying into 14-110
 - copying to new files 14-62
- block comment symbol 4-17
- BOF() 12-15
 - SKIP and 12-111
- boldface attributes 15-100
- bookmark data types 12-16
- bookmark() method 14-49, 14-50
- BOOKMARK() 12-16
- bookmarks
 - adding 14-49
 - moving to 14-75
 - returning 12-16
- boolean values, defined 2-3
- border property 15-66
- borders
 - adding to objects 15-66
 - form areas 15-40
 - setting 15-66
 - shape objects 15-147, 15-148
- borderStyle property 15-66
- Borland online 1-1
- bottom property 15-67

- braces ({}) 4-18
- breakpoints
 - defined 21-22
- British date format 8-23
- BROWSE 12-16
 - SET REFRESH and 12-105
- Browse class 15-4
 - events (table) 15-5
 - methods (table) 15-5
 - properties (table) 15-5
- browse objects
 - adding records 15-62
 - changing data 15-120
 - controlling cursors 15-82
 - designating tables 15-56
 - displaying data 15-96
 - restricting data entry 15-62
 - scrolling 15-109, 15-159, 15-182
- browsing 12-16, 14-24
- buffers
 - data
 - updating 12-74
 - writing to disk 12-49
 - file, flushing 11-20
 - typeahead
 - clearing 16-15
 - information, getting 16-21, 16-28
 - inserting keystrokes 16-24
 - size, setting 16-40
- BUILD 20-1
- buttons *See* pushbuttons; radio buttons

C

- cacheUpdates property 14-50
- caching updates 14-5, 14-50
 - attempt to apply 14-39
 - logging transactions vs. 14-47
- CALCULATE 12-18
- calculated fields
 - accessing 12-97
 - displaying 12-18, 15-97
 - indexing 12-54
 - read-only 15-97
 - setting values 14-14, 14-42
- calculations in reports 17-13 to 17-19
- call chain 5-12
 - preprocessor 22-10
- call operator 4-12
- canAbandon event 14-51

- canAppend event 14-52
- CANCEL 21-3
 - SUSPEND vs. 21-22
- canceled
 - changes to data 14-31 to 14-33
 - program execution 21-3
 - transactions 14-113
- canChange event 14-53
- canClose event 14-54, 15-68, 15-71
- canDelete event 14-54
- canEdit event 14-55
- canGetRow event 14-56
- canNavigate event 14-56
- canOpen event 14-57
- canRender event 17-21
- canSave event 14-57
 - canChange and 14-53
- capital gains 7-6
 - present value 7-11
- capitalization, specifying 6-18
- carriage returns
 - character, counting 6-14, 6-15, 6-21
 - substrings 6-5, 6-28
 - files 11-24, 11-31
- CASE 5-5
- case
 - converting 6-17, 6-29, 6-31, 12-55
 - first letter 6-17, 6-29
 - testing for lowercase 6-11, 6-12
 - testing for uppercase 6-12, 6-13
- case sensitivity
 - pattern matching 6-16
 - program code 3-2
 - searches using scan() 10-35
 - sorting data 12-112
- case statements 5-13
- CATCH 5-5
- CD 11-6
- CDOW() 8-6
- CEILING() 7-3
 - compared (table) 7-7
- CENTER() 6-6
- centering forms 15-62
- centering graphics 15-58
- centering text 6-6
- century 8-22
- CERROR() 21-3
- CHANGE() 12-20
 - CONVERT and 20-5

- changedTableName
 - property 14-58
- changes
 - undo() property 15-175
- changes to data
 - canceled
 - current row 14-31
 - multiple rows 14-32, 14-33
 - transactions 14-113
 - committing 14-60
 - undoing 14-58, 16-19
- changing
 - binary fields 12-80
 - data 5-28, 12-75, 12-77, 12-120, 15-84
 - attempts 14-53
 - browse objects 15-120
 - DDE applications 19-17, 19-18, 19-28
 - DDE servers 19-11
 - event handling 14-97
 - multiuser
 - environments 12-20, 12-48, 12-64, 12-84
 - data types 20-22
 - drive and directory 11-6
 - current working 11-37
 - field names 20-22
 - field widths 20-22
 - file names 11-33, 11-34
 - forms 20-8
 - key fields 12-78
 - memo fields 12-77
 - mouse pointers 15-121, 15-137
 - property settings 20-19
 - records 15-113
 - SET command values 20-23
 - statements at runtime 4-13
 - table structures (local SQL) 13-8
 - tables 20-5
 - names 12-76
 - structures 20-6, 20-21
 - text, spin boxes 15-168
- character codes (data types) 10-19
- character data
 - case, testing 6-11, 6-12, 6-13
 - converting dates 8-10, 8-11
 - converting numbers 6-25, 20-22
 - deleting specific
 - characters 6-26, 6-27
 - finding in DLLs 19-25
 - key expressions 12-55
 - returning logical 20-22
 - returning numbers 20-22
 - writing to file 11-41, 11-42
- character expressions
 - deleting spaces 6-30
 - phonetic values 6-8, 6-23
 - picture templates 18-1
 - repeating 6-19
 - replacing strings 6-26
- character set conversions 21-1, 21-11
- character sets 21-19
 - current, returning 21-4
- characters
 - adjusting spacing 17-42
 - ASCII values, returning 6-4, 6-7, 6-8
 - carriage return 11-24, 11-31
 - converting to dates 20-22
 - currency symbol 7-15
 - date separators 8-25, 16-44
 - changing 8-23
 - decimal separator 7-16
 - function templates 15-103, 18-2
 - linefeeds 11-24, 11-31
 - literal 6-16
 - position, specifying 6-7
 - returning number
 - written 11-42
 - returning specified
 - number 11-32
 - spaces, returning 6-24
 - testing for alphabetic 6-11
 - thousands separator 7-17
 - time separators 16-44
 - wildcard
 - pattern matching 6-16
 - temporary files 11-21
- charAt() method 6-7
- CHARSET() 21-4
- check boxes 15-6
- CheckBox class 15-6
 - events (table) 15-7
 - properties (table) 15-7
- checked 15-72
- Checked property 16-13, 16-14, 16-45
- checkmarks, adding to
 - menus 16-13
- child tables 12-106
 - moving through 12-109
- CHOOSEPRINTER() 18-6
- _pdriver and 18-28
- _porientation and 18-34
- _ppitch and 18-35
- chr() method 6-8
- CHR() 6-7
 - ASC() vs. 6-4
 - SET BELL and 20-24
- class keyword 4-11
- CLASS...ENDCLASS 5-5
- classes
 - ActiveX 15-3
 - Array 10-2
 - AssocArray 10-5
 - Band 17-5
 - Browse 15-4
 - CheckBox 15-6
 - ColumnCheckBox 15-8
 - ColumnComboBox 15-9
 - ColumnEntryField 15-10
 - ColumnHeadingControl 15-11
 - ColumnSpinBox 15-12
 - ComboBox 15-13
 - Container 15-15
 - creating new instances 4-10
 - Database 14-3
 - DataModRef 14-9
 - DataModule 14-6
 - Date 8-2
 - DbError 14-10
 - DbException 14-11
 - DbfField 14-12
 - DDELink 19-1
 - DDETopic 19-6
 - declaring 5-5
 - defined 2-14
 - dynamic subclassing 2-14, 5-3
 - Editor 15-16
 - Entryfield 15-18
 - Exception 5-1
 - Field 14-3
 - File 11-3
 - finding 5-19
 - Form 15-20
 - Grid 15-24
 - GridColumn 15-26
 - Group 17-6
 - identifying 5-7
 - Image 15-27
 - Line 15-28
 - ListBox 15-30
 - LockField 14-15
 - NoteBook 15-31
 - Object 5-3

- OLE 15-32
- OleAutoClient 19-9
- PageTemplate 17-8
- PaintBox 15-34
- Parameter 14-15
- PdxField 14-17
- Popup 16-8
- Progress 15-36
- PushButton 15-37
- Query 14-18
- RadioButton 15-38
- Rectangle 15-39
- Report 17-9
- ReportViewer 15-41
- Rowset 14-20
- ScrollBar 15-42
- Session 14-25
- Shape 15-43
- Slider 15-44
- SpinBox 15-46
- SqlField 14-27
- StoredProc 14-27
- StreamFrame 17-10
- StreamSource 17-12
- String 6-1
- TabBox 15-48
- Text 15-49
- Timer 8-4
- Toolbar 16-9
- ToolButton 16-11
- TreeItem 15-51
- TreeView 15-53
- UpdateSet 14-30
- classId property 15-73
- className property 5-7
- CLEAR 18-6
- CLEAR ALL 20-2
- CLEAR AUTOMEM 12-21
 - RELEASE AUTOMEM and 12-76
 - STORE AUTOMEM vs. 12-113
- CLEAR FIELDS 12-21
- CLEAR MEMORY 5-7
- CLEAR PROGRAM 5-7
- CLEAR TYPEAHEAD 16-15
- clearFilter() method 14-59
- clearing memory variables 5-7, 5-35, 5-37
- clearing typeahead buffers 16-15
- client/server applications
 - MDI forms 15-117
- client-based SQL *See* SQL (local)
- clock 8-8
 - setting 8-24, 8-25
 - time elapsed 8-11
- CLOSE ALL 20-2
- CLOSE ALTERNATE
 - SET ALTERNATE and 18-14
- CLOSE DATABASES 12-22
- CLOSE INDEXES 12-22
- CLOSE PROCEDURE 5-8
- CLOSE TABLES 12-22
- close() method 15-74
 - connections 14-59
 - files 11-7
- closing
 - applications 5-34
 - databases 14-34
 - files 11-7, 20-2
 - forms 15-68, 15-71, 15-74, 15-95
 - indexes 12-22
 - procedures 5-8
 - program files 5-40
 - queries 14-97
 - reports 15-74
 - sessions 14-26
 - stored procedures 14-97
 - tables 20-2
 - work areas 12-22
- closing events 15-129
- CMONTH() 8-7
- code
 - case sensitivity 3-2
 - commenting 2-17, 4-17
 - coverage analysis 22-12
 - editing 20-4, 20-28
 - optimizing 22-4
 - reserved symbols 4-16
 - testing 20-26, 21-3
- code execution
 - after try block 5-19
 - conditionally 5-12, 5-23, 5-24
 - managing with exceptions 5-2
 - redirecting calls 5-38
 - skipping statements 5-26
 - stopping 5-34
- code modules 5-22
- codeblocks 2-12, 5-5, 5-26
 - assigning to variables 4-18
 - creating 4-18
 - defined 2-12
 - executing 4-12
- color arguments 16-15
- color palettes 16-19
- colorHighlight property 15-75
- colorNormal property 15-75
- colorHighlight vs. 15-75
- colors
 - defining 16-15
 - custom 16-19
 - objects 15-75
 - selecting 16-19
- colors commands
 - DEFINE COLOR 16-15
 - GETCOLOR() 16-19
- columns *See* fields
- combo boxes
 - displaying lists 15-92
 - selecting options 15-85
 - setting styles 15-169
 - sorting 15-167
- ComboBox class 15-13
 - events (table) 15-14
 - methods (table) 15-14
 - properties (table) 15-14
- Command window
 - clearing results pane 18-6
 - displaying files 11-15, 11-26, 11-40
 - displaying messages 20-15, 20-32
 - current environment 20-16
 - pausing program execution 16-44
 - restoring memory variables 5-38
 - resuming program execution 21-16
 - returning table structures 20-18
 - saving output 18-14
 - shelling to DOS 16-42
 - suspending program execution 21-22
 - writing to 12-42, 12-60, 18-19
- commands 18-14
 - executing
 - page formatting 18-8
 - shortcuts 16-29
 - scope 12-3
- comments 4-17
- commit() method 14-60
- COMMIT() 12-23
- committing changes 14-60
- committing transactions 12-23
 - DDE applications 19-27
- common logarithms 7-8
- comparing
 - dates 4-8
 - expressions 7-8, 7-9

- mismatched data types 4-8
- record counters 12-20
- comparison operators 4-7
 - local SQL 13-3
- COMPILE 20-3
- compiler errors 21-3
- compiling 4-19, 22-11
 - automatically 5-11, 20-27
 - canceled 20-3
 - conditional (#else) 22-6
 - conditional (#if) 22-6
 - conditional (#ifdef) 22-8
 - conditional (#ifndef) 22-9
 - controlling (#define) 22-2
 - format files 20-27
 - multiple programs 22-11
 - options, setting 22-11
- components
 - controlled update 14-95
 - current value 15-179
 - custom 15-34
 - displaying conditionally (in reports) 17-41
 - generic 15-35
 - height 15-107
 - varying 17-42
 - hiding 15-181
 - hints 15-166, 15-168
 - left edge position 15-115
 - loading 15-130
 - locating 15-101
 - losing focus 15-127, 15-135
 - multi-line text 15-18
 - opened events 15-139
 - properties, altering
 - conditionally 17-21
 - referencing 15-123
 - refreshing data 14-108, 14-109
 - rendered in reports
 - handling event 17-33
 - resetting default (in reports) 17-33
 - right edge position 15-157
 - setting focus 15-163
 - title, setting 17-42
 - top edge position 15-174
 - visual
 - common events
 - (table) 15-2
 - common methods
 - (table) 15-3
 - common properties
 - (table) 15-1
 - width 15-183
- compound expressions 4-2
- concatenation operators 4-4, 4-5
- conditional execution 5-12, 5-24
 - OS() 11-29
- conditional statements 5-26
- conditions
 - See also* control structures
 - evaluating 5-23
 - exceptions 5-13
 - search operations 12-62
 - testing 5-20, 5-24
 - alternate 5-17
 - multiple 5-24
- cones, measuring volume 7-11
- confirmation messages 12-108
- connections *See* database servers
- constants
 - changing 22-4
 - defining 22-2
 - identifiers 22-3
 - pi 7-11
- constrained property 14-60
- constraining updates 14-60
- constructor code 5-6
- contained in operator 4-8
- Container class 15-15
 - properties (table) 15-16
- container objects 5-10
 - testing for parent 5-30
- containers
 - creating 14-6, 15-15
 - getting object
 - position 15-115, 15-157
 - multi-page 15-31
- context-sensitive help 15-108
- CONTINUE 12-23
 - FOUND() and 12-50
 - LOCATE and 12-63
- continuing search
 - operations 12-23
- control codes (printer) 18-17
- Control menu 15-170
- control statements 5-46, 5-52
- control structures
 - linear 5-12, 5-23, 5-24
 - loops 5-14, 5-15, 5-20, 12-86
- controlling table access 21-15
- controls *See* components
- conventions, typographical 1-2
- CONVERT 20-4
 - COPY and 12-25
 - COPY STRUCTURE and 12-28
- converting
 - ASCII to characters 6-7, 6-8
- case 6-17, 6-29, 6-31, 12-55
 - first letter 6-17, 6-29
- characters to ASCII 6-4
- characters to dates 8-7
- characters to numbers 20-22
- dates to characters 8-10, 8-11
- dates to strings 8-9, 8-19
- decimal to hexadecimal 9-7
- degrees to radians 7-5
- external functions 19-15
- hexadecimal to decimal 9-7
- incompatible data
 - types 20-23
- logical fields to
 - characters 20-22
- numbers to characters 6-25, 20-22
- numbers to logical
 - values 20-22
- radians to degrees 7-14
- COPY 12-24
 - PACK and 12-71
- COPY BINARY 12-26
- COPY FILE 11-9
- COPY MEMO 12-26
- COPY STRUCTURE 12-27
- COPY TABLE 12-29
- COPY TO ARRAY 12-30
- COPY TO...STRUCTURE
 - EXTENDED 12-28
 - CREATE...FROM and 12-35
 - CREATE...STRUCTURE EXTENDED vs. 12-37
- copy() method 11-8, 14-61, 15-81
- copying
 - array elements 10-7
 - binary fields 12-26
 - BLOB fields 14-62
 - data 12-7, 14-30, 14-61
 - arrays and 12-30, 12-81
 - automatically 12-24
 - multiple fields 12-25, 12-28
 - files 11-8, 11-9, 11-41
 - to BLOB fields 14-110
 - index files 12-25
 - memo fields 12-9, 12-25, 12-26
 - to text files 12-25
 - memory variables 5-37
 - parameters 5-29
 - tables 12-29, 14-61
 - structures 12-27, 12-28
 - text 15-81, 16-17

- text files 12-10, 12-83
- copyTable() method 14-61
- copyToFile() method 14-62
- core language elements 5-1
- COS() 7-4
 - ACOS() and 7-1
 - DTOR() and 7-5
- cosecant 7-19
 - inverse 7-2
- cosine 7-4
 - inverse, returning 7-1
 - reciprocal 7-4
- cotangent 7-20
 - inverse 7-3
- COUNT 12-32
 - RECCOUNT() vs. 12-73
- COUNT() function (SQL) 13-6
- count() method 10-9, 14-62, 15-82
- counting fields 12-46
- counting items in group 17-14
- counting records 11-14, 12-18, 12-73, 14-62, 14-113
- counting rows in rowset 14-62
- counting specified values 13-6
- coverage files 20-13
 - creating 20-25
 - updating 20-25
- CREATE 20-5
 - CREATE...FROM vs. 12-36
- CREATE... commands
 - SET DESIGN and 20-26
- CREATE COMMAND 20-6
- CREATE DATAMODULE 20-7
- CREATE FILE 20-7
- CREATE FORM 20-8
- CREATE INDEX statement (SQL) 13-9
- CREATE LABEL 20-9
- CREATE MENU 20-10
- CREATE POPUP 20-10
- CREATE PROJECT 20-11
- CREATE QUERY 20-11
- CREATE REPORT 20-12
- CREATE SESSION 12-33
- CREATE TABLE statement (SQL) 13-9
 - data type mappings 13-10
- CREATE...FROM 12-35
 - COPY TO...STRUCTURE EXTENDED and 12-29
 - CREATE...STRUCTURE EXTENDED and 12-37
- CREATE...STRUCTURE EXTENDED 12-36

- CREATE...FROM and 12-35
- createDate() method 11-11
- createTime() method 11-11
- creating
 - containers 15-15
 - custom components 15-34
 - files 11-9
 - forms 15-23
 - member properties 5-5
 - methods 5-5
 - objects 4-10, 5-3, 5-9
 - pop-up menus 15-150
 - tables (local SQL) 13-9
- CTOD() 8-7
- Ctrl keys, command execution 16-38
- CUAENTER(SET) 16-36
- cuaTab property 15-82
- currency symbols 7-15
- current database 12-92
 - name, returning 12-37
- current date
 - returning 8-8
 - setting 8-24
- current object
 - colors, setting 15-75
 - finding 15-56
- current record 14-2
 - number, returning 12-73
 - updating 12-80
- current settings 5-42, 5-43
 - character set 21-4
 - environment 20-16
 - language driver 21-7
- current work areas 12-124
- curSel property 15-83
- cursor, row
 - determining position 14-40, 14-41, 14-67
 - moving forward or backward 14-94
 - moving to first row 14-73
 - moving to last row 14-78
 - moving to specified row 14-75
- cursors
 - controlling 15-82, 16-35, 16-36
- custom classes 5-5
- custom components 15-34
- cut() method 15-84
- cutting text 15-84
- cylinders, measuring volume 7-11

D

- data
 - accessing 12-21, 14-1 to 14-3
 - error handling 14-11
 - multiuser
 - environments 21-10
 - file-sharing
 - modes 12-95
 - setting locks 12-48, 12-64, 12-84
 - read-only
 - restrictions 12-97, 12-102
 - specific fields 12-95
 - appending 14-24, 14-36, 14-43
 - and updating 14-37
 - handling key violations 14-78
 - to BLOB fields 14-110
 - attempting to change 14-53
 - bookmarking 14-49
 - browsing 14-24
 - caching updates
 - locally 14-50
 - canceling changes
 - current row 14-31
 - multiple rows 14-32, 14-33
 - transactions 14-113
 - change indicator 14-93
 - changed 14-97
 - changing 5-28, 12-75, 12-77, 12-120, 14-58, 15-84
 - attempts 14-53
 - browse objects 15-120
 - DDE applications 19-11, 19-17, 19-18, 19-28
 - multiuser
 - environments 12-20, 12-48, 12-64, 12-84
 - committing changes 14-60
 - constraining updates 14-60
 - converting formats 14-30
 - copying 12-7, 14-30, 14-61
 - arrays and 12-30, 12-81
 - automatically 12-24
 - BLOB fields 14-62
 - multiple fields 12-25, 12-28
 - determining if editable 14-79
 - displaying
 - browse objects and 15-96
 - memo fields 12-103

- multiple lines 15-16
- one line 15-18
- specific records 12-42, 12-60
 - with BROWSE 12-16
- Edit mode, setting 14-42
- editing 12-43, 14-24, 14-45, 14-79
 - with BROWSE 12-16
- entering *See* data entry
- filtering 14-24, 14-37, 14-46, 14-56
- formatting 15-148
- grouping 17-7
 - calculating aggregate values 17-7
 - report-level 17-10
- linking fields to report 17-12
- locating 14-24, 14-38, 14-46
 - controlling criteria 14-80
 - next match 14-79
 - similar spellings 6-23
- losing 12-71, 12-78, 20-22
 - minimizing loss 12-92
- manipulating 12-111
- multi-line input fields 15-16
- organizing 12-54, 12-112
- overwriting 12-77
 - binary fields 12-26
 - confirmation
 - messages 12-108
 - memo fields 12-10, 12-27, 12-83
- processing
 - optimizing 12-92
 - specific records 12-101
- protecting 20-27, 21-15
- refreshing 14-108, 14-109, 15-85
- rendering in reports 17-10
- sample 12-51
- saving 12-49, 14-93
 - automatically 12-92
- searching 14-38, 14-46
 - locate options 14-80
 - next match 14-79
- shared resources 14-117
- similar spellings, finding 6-23
- single-line input fields 15-18
- source, identifying 17-40
- time stamps 14-96
- updating 12-65, 12-120, 14-37, 15-84
 - automatic 14-95
 - cached 14-39
 - constraints 14-60
 - from another table 14-30
 - multiuser
 - environments 12-48, 12-85
 - problems with 14-106
 - SQL (local) 13-17
 - validating 15-178
- data access methods
 - abandon() 14-31
 - abandonUpdates() 14-33
 - access() 14-33
 - addPassword() 14-35
 - append() 14-36
 - appendUpdate() 14-37
 - applyFilter() 14-37
 - applyLocate() 14-38
 - applyUpdates() 14-39
 - atFirst() 14-40
 - atLast() 14-41
 - beginAppend() 14-43
 - beginEdit() 14-45
 - beginFilter() 14-46
 - beginLocate() 14-46
 - beginTrans() 14-47
 - bookmark() 14-49, 14-50
 - clearFilter() 14-59
 - commit() 14-60
 - copy() 14-61
 - copyTable() 14-61
 - copyToFile() 14-62
 - count() 14-62
 - delete() (Rowset) 14-65
 - delete() (UpdateSet object) 14-65
 - dropTable() 14-66
 - emptyTable() 14-67
 - executeSQL() 14-68
 - first() 14-73
 - goto() 14-75
 - last() 14-78
 - locateNext() 14-79
 - lockRow() 14-82
 - lockSet() 14-84
 - login() 14-85
 - next() 14-94
 - packTable() 14-103
 - prepare() 14-105
 - refresh() 14-108
 - refreshControls() 14-109
 - refreshRow() 14-109
 - reindex() 14-109
 - renameTable() 14-110
 - replaceFromFile() 14-110
 - requery() 14-111
 - rollback() 14-113
 - save() 14-115
 - tableExists() 14-120
 - unlock() 14-122
 - unprepare() 14-122
 - update() 14-123
 - user() 14-125
- data buffers 12-49
 - flushing to disk 14-73
 - updating 12-74
- data definition statements 13-4
- data entry
 - controlling 15-177
 - cursors, moving 16-35
 - DDE applications 19-23
 - invalid 15-178, 20-24
 - multiple lines 15-16
 - restricting 15-62, 20-27
 - single line 15-18
 - templates 15-103, 15-149
- data integrity 12-49
- data manipulation statements 13-5
 - parameter substitutions 13-5
- data modules 14-7
 - getting class name 14-64
 - referencing 14-9, 14-108
- data source, identifying 17-40
- data types
 - binary fields 12-13, 12-26
 - bookmark 12-16
 - changing 20-22
 - character codes 10-19
 - comparing mismatched 4-8
 - DLLs 19-12
 - external functions 19-15
 - incompatible 20-23
 - kinds of 2-2 to 2-5
 - returning 5-50, 14-121
 - SQL mappings 13-10
 - user-defined 12-26
- Database class 14-3
 - methods (table) 14-4
 - properties (table) 14-4
- database driver, identifying 14-66
- Database objects
 - See also* databases
 - creating 14-3
 - default, defined 14-2
 - setting up 14-5
 - testing for active 14-34
- database property 14-63
- database servers

- connecting to 12-69, 15-90, 19-16, 19-23
- disconnecting 14-59, 19-27
- DATABASE() 12-37
- databaseName property 14-63
- databases
 - See also* data; Database objects; tables
 - cached updates 14-5
 - closing 14-34
 - connecting to 14-102
 - current, specifying 12-92
 - default 14-5
 - information, getting 14-74
 - logging into 14-85, 14-86, 14-125
 - names, returning 12-37
 - opening 12-70, 14-34
 - parent, determining 5-30
 - referencing tables 4-17
 - sessions, assigning to 14-116
 - transaction logging 14-5
 - transaction processing 14-6
- dataLink property 15-84
- dataLinked, defined 14-3
- dataModClass property 14-64
- DataModRef class 14-9
 - properties (table) 14-9
- DataModule class 14-6
 - properties (table) 14-7
- dataSource property 15-85, 15-87
- date and time commands
 - SET CENTURY 8-22
 - SET DATE 8-23
 - SET DATE TO 8-24
 - SET EPOCH 8-24
 - SET MARK 8-25
 - SET TIME 8-25
- date and time functions
 - CDOW() 8-6
 - CMONTH() 8-7
 - CTOD() 8-7
 - DATE() 8-8
 - DAY() 8-8
 - DMY() 8-9
 - DOW() 8-9
 - DTOD() 8-10
 - DTOS() 8-11
 - ELAPSED() 8-11
 - MDY() 8-19
 - MONTH() 8-19
 - UTC() 8-30
 - YEAR() 8-31
- date and time methods
 - getDate() 8-13
 - getDay() 8-14
 - getHours() 8-14
 - getMinutes() 8-15
 - getMonth() 8-15
 - getSeconds() 8-16
 - getTime() 8-16
 - getTimezoneOffset() 8-17
 - getYear() 8-18
 - parse() 8-21
 - setDate() 8-25
 - setHours() 8-26
 - setMinutes() 8-26
 - setMonth() 8-26
 - setSeconds() 8-27
 - setTime() 8-27
 - setYear() 8-28
 - toGMTString() 8-29
 - toLocaleString() 8-29
 - toString() 8-30
- date and time stamps, returning 11-12
- Date class 8-2
 - methods (table) 8-3
 - properties (table) 8-3
- date fields
 - converting characters 20-22
- date formats 8-22
 - returning 8-9, 8-11, 8-19
 - specifying 8-23
- Date objects 8-1 to 8-4
- DATE parameter 8-23
- DATE() 8-8
- date() method 11-12
- date/time classes
 - Date 8-2
 - Timer 8-4
- dates 8-23, 15-169
 - See also* Date objects
 - comparing 4-8
 - converting
 - using GMT conventions 8-29
 - using locale conventions 8-29
 - converting to characters 8-10, 8-11
 - converting to strings 8-9, 8-19
 - default settings 8-23
 - key expressions 12-55
 - literal 4-18
 - manipulating 8-7
 - resetting 8-24
 - returning 8-7, 8-8, 12-65
- character expressions
 - as 8-7
 - day of month 8-25
 - system 8-8
 - weekdays 8-6, 8-9
 - year 8-22
- separators 8-25
 - changing 8-23
- setting base year 8-24
- sorting 8-11
- valid range 8-24
- DAY() 8-8
- dBASE, exiting 5-34
- dBASE III PLUS files 12-24
- dBASE IV commands
 - (backward compatible)
 - CLEAR TYPEAHEAD 16-15
 - SET ODOMETER 12-104
- dBASE IV printing commands
 - _alignment 18-20
 - _indent 18-21
 - _lmargin 18-22
 - _padvance 18-23
 - _pageno 18-24
 - _pbpage 18-25
 - _pcolno 18-26
 - _pcopies 18-27
 - _pdriver 18-27
 - _peject 18-28
 - _pepage 18-29
 - _pform 18-30
 - _plength 18-31
 - _plineno 18-32
 - _ploffset 18-33
 - _porientation 18-34
 - _ppitch 18-35
 - _pquality 18-36
 - _pspacing 18-36
 - _rmargin 18-37
 - _tabs 18-38
 - _wrap 18-39
- _dbaselock fields 12-20, 12-25, 20-4
 - accessing 12-61
 - copying 12-28, 12-29
- _DBASELOCK field 14-15
- __dbasewin__ 22-13
 - #if and 22-13
- DbError class 14-10
 - properties (table) 14-11
- DBERROR() 21-4
- DbException class 14-11
 - properties (table) 14-11
- DBF() 12-38
- DbfField class 14-12

- properties (table) 14-12
- DBMESSAGE() 21-5
- _dbwinhome 11-43
- DDE 19-1
- DDE links
 - disabling 19-27
 - information, getting 19-27, 19-28
 - setting 19-16
- DDE server applications 19-1
 - accessing 19-27
 - changing data 19-17, 19-18, 19-28
 - events, trapping 19-19, 19-20, 19-21
 - name, returning 19-26, 19-28
 - reading from 19-21
 - transactions,
 - committing 19-27
 - writing to 19-11, 19-23
- DDELink class 19-1
- DDETopic class 19-6
- DEBUG 20-13
- Debugger, opening 20-13
- debugging
 - conditional compilation 22-7
 - coverage analysis 20-25
 - procedures 21-14
 - program flow, tracking 21-8
 - programs, stepping through 20-13, 20-14, 20-15
 - records, stepping through 12-86
 - suspending program
 - execution 21-22
 - UDFs 21-14
- debugging commands
 - DEBUG 20-13
 - DISPLAY COVERAGE 20-13
 - GENERATE 12-51
 - RESUME 21-16
 - SET COVERAGE 20-25
 - SUSPEND 21-22
- decimal digits 6-25
 - decimal separator 7-16
 - deleting 7-7
 - equality 7-4, 7-6
- decimal places 7-15
 - returning 7-13, 12-44, 14-64
- decimal values
 - converting to
 - hexadecimal 9-7
 - keystrokes, returning 16-21, 16-28
 - returning 9-7
- decimalLength property 14-64
- declarations
 - classes 5-5
 - functions 5-21
 - procedures 5-32
 - variables 4-12, 5-31, 5-33
 - local 5-25
 - private 5-31
 - public 5-33
 - static 5-44
- decreasing spin box
 - values 15-169
- decrement operator 4-6
- default property 14-64, 15-88
- defaults
 - date and time 8-23
 - separators 16-44
 - decimal places 7-16
 - file names 12-24
 - function keys 16-39
 - sort order 12-112
 - system bell 20-24
 - tables 12-93
- DEFINE 5-9
- DEFINE COLOR 16-15
- #define directive 22-2
- defining fields lists 12-95
- degrees
 - converting from radians 7-14
 - converting to radians 7-5
 - returning 7-14
- delaying program
 - execution 16-44
- DELETE 12-38
 - PACK vs. 12-71
 - RECALL and 12-72
 - ZAP vs. 12-124
- DELETE FILE 11-14
- DELETE statement (SQL) 13-11
- DELETE TABLE 12-39
- DELETE TAG 12-40
- delete() method
 - arrays 10-10
 - files 11-13
- delete() method (rowSet
 - object) 14-65
- delete() method (UpdateSet
 - object) 14-65
- DELETED() 12-40
- deleting
 - array elements 10-10, 10-30
 - all 10-30
 - current row 14-65
 - decimal digits 7-7
 - fields 13-8, 20-22
- files 11-13, 11-14, 11-17
- index files 12-39, 13-12
- indexes (local SQL) 13-12
- leading spaces 6-15, 6-17
- memo files (local SQL) 13-12
- memory variables 5-35
- records 12-40, 12-71, 12-124, 13-11, 15-55
 - all in table 14-67
 - attempted 14-54
 - confirming 12-108
 - controlling 12-93
 - event handling 14-98
 - from destination
 - tables 14-65
 - in rowsets 14-65
 - specified characters 6-26, 6-27
 - tables 12-39, 13-12, 14-66
 - text 15-84, 16-18
 - trailing spaces 6-21, 6-30
 - trailing zeros 7-13
- delimiters
 - command execution 16-39
 - date 8-25
 - changing 8-23
 - decimal digits 7-16
 - directory paths 11-38
 - thousands 7-17
 - time 16-44
- derived classes 5-5
- descending sort order 12-53, 12-112
- DESCENDING() 12-41
- designing forms 15-89, 20-27
- designing table structures 12-36
- designView property 15-89
- destination property 14-66
- detach method 16-16
- detail band 17-6
- detailBand property 17-22
- development tools 20-25
- dialog boxes 15-154
 - message 16-26
- DIFFERENCE() 6-8
 - LIKE() vs. 6-16
 - SOUNDEX() and 6-23
- dimensions property 10-13
- dir() method 10-14
- DIR/DIRECTORY 11-14
 - SET SEPARATOR and 7-18
- directives 4-19
- directories
 - changing 11-6
 - current working 11-37

- creating 11-26, 11-27
- searching 11-38
- directory lists 11-14
- directory paths
 - returning 11-37, 21-14
 - VDB.EXE 11-25, 11-26
 - separators 11-38
- dirExt() method 10-16
- disabledBitmap property 15-89
- disk drives
 - changing 11-6
 - current working 11-37
 - disk space, returning 11-14, 11-15
 - valid, returning 11-41
- DISKSPACE() 11-15
- DISPLAY 12-42
 - SET HEADINGS and 12-99
- DISPLAY COVERAGE 20-13
- DISPLAY FILES 11-15
 - SET SEPARATOR and 7-18
- DISPLAY MEMORY 20-15
- display options 12-17
 - browse objects 15-96
- DISPLAY STATUS 20-16
- DISPLAY STRUCTURE 20-17
 - RECSIZE() and 12-74
- displaying
 - data
 - browse objects and 15-96
 - field names 12-99
 - in reports 17-22
 - in StreamFrames 17-20
 - memo fields 12-103
 - specific records 12-42, 12-60
 - with BROWSE 12-16
 - editing windows 20-6
 - field definitions 20-17
 - forms
 - display states 15-184
 - specifying 15-174
 - topmost 15-174
 - graphics 19-25
 - messages 16-44, 20-32
 - current
 - environment 20-16
 - in status bars 15-169, 16-40
 - memory variables 20-15
 - multi-line text 15-18
 - property settings 20-19
 - summary information (reports) 17-25
 - table structures 20-21

- text files 11-40
- windows 16-44
- division 4-6, 7-10
- division operator 4-6
- DLLs 19-1, 19-12
 - calling external 19-12
 - character strings, getting 19-25
 - defined 19-17
 - initializing 19-16
 - releasing 19-24
 - search paths 19-14
- DMY() 8-9
- DO 5-10
 - SET DEVELOPMENT and 20-27
 - SUSPEND and 21-23
- DO CASE 5-12
 - IF vs. 5-24
- DO WHILE 5-14
 - DO...UNTIL vs. 5-15
 - SCAN vs. 12-87
 - SLEEP vs. 16-45
- DO...UNTIL 5-15
- documentation
 - organization 1-1
 - typographical conventions 1-2
- doesn't begin with operator 4-8
- DOS
 - commands, executing 11-16
 - environment variables 11-22
 - file attributes 10-15
 - return codes 5-34
- DOS command 11-16
 - RUN vs. 11-35
- dot operator 4-11
- double-quotation mark
 - symbol 4-16
- doVerb() method 15-90
- DOW() 8-9
- downBitmap property 15-91
- drawing lines 15-28, 15-147
- drawing shapes 15-147, 15-148, 15-164
- driverName property 14-66
- DROP INDEX statement (SQL) 13-12
- DROP TABLE statement (SQL) 13-12
- drop-down lists 15-170
- dropDownHeight
 - property 15-92
- dropTable() method 14-66
- DTOC() 8-10

- DTOR() 7-4
 - COS() and 7-4
 - SIN() and 7-19
- DTOS() 8-11
- duplicate data, suppressing (in reports) 17-41
- duplicate values 12-59, 13-14
 - keys 12-109
- duplicating
 - character strings 6-19, 6-20
 - tables 14-61
- duration (bell) 20-24
- dynamic subclassing 2-14, 5-3

E

- EDIT 12-43
 - SET REFRESH and 12-105
- Edit mode
 - attempted 14-55
 - automatic 14-42
 - controlling 14-45
 - entered event 14-98
 - setting 14-42
- editCopyMenu property 16-17
- editCutMenu property 16-18
- editing 12-17
 - code 20-4, 20-28
 - data 12-43, 14-24, 14-45, 14-79
 - with BROWSE 12-16
 - multi-line text 15-18
 - programs 20-6
 - restricting 12-18
 - text files 20-7
- editing windows 20-6
 - displaying 20-7
- Editor class 15-16
 - events (table) 15-17
 - methods (table) 15-17
 - properties (table) 15-17
- Editor objects
 - controlling cursors 15-82
 - current line 15-115
 - memo fields 12-103
 - scrolling 15-109, 15-159, 15-182
 - wordwrapping text 15-184
- editors, text, alternate 20-6, 20-28
- editPasteMenu property 16-18
- editUndoMenu property 16-19
- EJECT 18-7
 - _peject and 18-29
- EJECT PAGE 18-8
 - EJECT vs. 18-7

- ON PAGE and 18-9
- ELAPSED() 8-11
- element() method 10-18
- elements, array *See* array elements
- elements property 15-93
- ELSE 5-17
- #else directive 22-6
- ELSEIF 5-17
- empty expressions 5-17
- empty memo fields 6-15
- EMPTY() 5-17
 - BLANK and 12-14
- emptying tables 14-67
- emptyTable() method 14-67
- enabled property 8-12, 15-94
- encrypting tables 21-17
- #endif directive 22-6
- end-of-file indicator 12-43
- end-of-line characters 11-24, 11-31, 11-42
 - writing 11-31, 11-41
- end-of-line comments 4-17
- endOfSet property 14-67
- endPage property 17-23
- Enter key, simulating Tab 16-36
- entry fields
 - backgrounds 15-146
 - keystrokes, evaluating 15-113
 - scrolling width 15-117
- Entryfield class 15-18
 - events (table) 15-10, 15-13, 15-19
 - methods (table) 15-19
 - properties (table) 15-8 to 15-12, 15-19
- environment commands
 - CHARSET() 21-4
 - CLEAR 18-6
 - CREATE SESSION 12-33
 - DISPLAY MEMORY 20-15
 - DISPLAY STATUS 20-16
 - LDRIVER() 21-7
 - MEMORY() 21-9
 - SET 20-23
 - SET BELL 20-23
 - SET DESIGN 20-26
 - SET EDITOR 20-28
 - SET FULLPATH 11-37
 - SET LDCHECK 21-18
 - SET LDCONVERT 21-19
 - SET MESSAGE 16-40
 - SET ODOMETER 12-104
 - SET SAFETY 12-108
 - SET TALK 20-32
 - SET() 5-42
 - SETTO() 5-43
 - SHELL() 16-41
- environment variables (DOS) 11-22
- environments, getting information 20-16
- eof() method 11-16
- EOF() 12-43
 - RECNO() and 12-73
- SEEK and 12-89
- SET RELATION and 12-107
- SKIP and 12-111
- epoch, setting 8-24
- equal to operator 4-8
- equality
 - comparing character strings 12-94
 - finding 7-3, 7-5
- ERASE 11-17
- erasing memo fields 12-10, 12-27, 12-83
- Error dialog box 20-3
- error handling
 - CERROR() 21-3
 - DBERROR() 21-4
 - DBMESSAGE() 21-5
 - ERROR() 21-6
 - LINENO() 21-8
 - MESSAGE() 21-10
 - ON ERROR 21-12
 - ON NETERROR 21-13
 - PROGRAM() 21-14
 - RETRY 21-16
 - SET ERROR 21-18
 - SQLERROR() 21-19
 - SQLMESSAGE() 21-22
- error messages
 - customizing 21-18
 - returning 21-5, 21-10
- error objects 14-10
- error() method 11-18
- ERROR() 21-6
 - resetting 21-16
- errors
 - BDE 14-10
 - compiler, returning 21-3
 - data entry 15-178
 - DDE applications 19-27
 - error number, returning 11-18
 - exceptions and 5-2
 - file (causes and numbers) 11-18
 - finding information 5-2
 - fixing 20-4
 - multiuser environments 21-13
 - resolving 21-16
 - run-time 21-13, 21-18
 - IDAPI 21-4, 21-5
 - line numbers, returning 21-8
 - server 21-19, 21-22
 - server 14-10
 - syntax 20-3
- Esc key 16-37
 - disabling 15-95, 16-30
- escape sequences 18-17
- escExit property 15-95
- event handling
 - Append mode
 - attempted 14-52
 - button clicked 15-128
 - component ready to render 17-21
 - component rendered (in report) 17-33
 - Edit mode attempted 14-55
 - Edit mode entered 14-98
 - field value change attempted 14-53
 - field's value property read 14-42
 - navigation attempted 14-56
 - query deactivation attempted 14-54
 - query open attempted 14-57
 - report page rendered 17-32, 17-35
 - row buffer addition attempted 14-56
 - row buffer save attempted 14-57
 - row buffer saved 14-102
 - row deleted 14-98
 - row deletion attempted 14-54
 - rowset abandon attempted 14-51
 - rowset abandoned 14-95
 - rowset closed 14-97
 - rowset navigated 14-99
 - rowset opened 14-100
 - value property changed (field) 14-96, 14-97
 - value property read (field) 14-99
- events

- adding records 15-126
- defined 2-14
- moving forms 15-122, 15-138
- moving record
 - pointers 15-138
- null values 5-18
- resizing forms 15-142
- selecting objects 15-94
- trapping
 - DDE applications 19-19, 19-20, 19-21
- visual components, common (table) 15-2
- exact matches (searches) 12-88
- failing 12-103
- exactly equal to operator 4-8
- Exception class 5-1
 - properties (table) 5-1
- Exception objects 5-1, 14-11
- exceptions
 - conditional 5-13
 - describing 5-1
 - generating 5-46
 - handling 5-2, 5-46
 - catch block and 5-5
 - managing execution with 5-2
- exclusive mode 12-94
- execute() method
 - DDE commands 19-11
 - queries or stored procedures 14-68
- executeSQL() method 14-68
- executing
 - actions 15-38
 - code
 - after try block 5-19
 - conditionally 5-12, 5-23, 5-24
 - managing with exceptions 5-2
 - redirecting calls 5-38
 - skipping statements 5-26
 - stopping execution 5-34
 - codeblocks 4-12
 - dBASE commands
 - page formatting 18-8
 - shortcuts 16-29
 - DOS commands 11-16
 - functions 5-10
 - macros, DDE applications 19-11
 - programs *See* program execution
 - queries 14-68
 - SQL statements 14-20, 14-68, 21-20
 - stored procedures 14-68
 - exists() method 11-19
 - create() and 11-10
 - EXIT 5-18
 - exiting dBASE 5-34
 - exiting loops 5-14, 5-18
 - EXP() 7-5
 - expandable property 17-23
 - exponentiation
 - base e 7-5
 - exponents, returning 7-8
 - square roots and 7-20
 - exporting tables with COPY 12-26
 - expression commands
 - EMPTY 5-17
 - TYPE() 5-50
 - expressions
 - arithmetic operators and 4-3
 - changing 22-2
 - comparing 7-8, 7-9
 - complex, how evaluated 2-9
 - defined 2-6
 - empty 12-57
 - testing for 5-17, 12-57
 - evaluating 5-24, 18-1
 - fields list 12-46
 - finding 12-64
 - in arrays 10-35
 - grouping 4-12
 - identifiers 22-3
 - operator precedence 4-2
 - replacing data 12-77
 - results, viewing 18-1, 18-2, 18-5, 18-20
 - storing 5-45
 - EXTERN 19-12
 - LOAD DLL and 19-17
 - external applications 15-90, 19-16, 19-23
 - external functions 19-12
 - external programming elements 19-1
 - EXTRACT() function (SQL) 13-7
- F**

 - FCREATE() 11-9
 - FDECIMAL() 12-44
 - Field class 14-13
 - events (table) 14-13
 - methods (table) 14-13
 - properties (table) 14-13
 - field commands
 - APPEND MEMO 12-10
 - BINTYPE() 12-13
 - CLEAR FIELDS 12-21
 - COPY BINARY 12-26
 - COPY MEMO 12-26
 - FDECIMAL() 12-44
 - FIELD() 12-45
 - FLDCOUNT() 12-46
 - FLDLIST() 12-46
 - FLENGTH() 12-47
 - ISBLANK() 12-57
 - MEMBLINES() 12-67
 - MLINE() 12-68
 - REPLACE BINARY 12-80
 - REPLACE
 - MEMO...FROM 12-83
 - REPLACE OLE 12-84
 - SET BLOCKSIZE 20-24
 - SET FIELDS 12-95
 - SET MBLOCK 20-31
 - SET MEMOWIDTH 12-103
 - field morphing 14-42, 14-53, 15-149
 - field names
 - aliases 12-78
 - automem variables and 12-76
 - changing 20-22
 - designating 4-12
 - display, suppressing 12-42
 - displaying 12-99
 - getting 12-45
 - SQL naming conventions 13-2
 - Field objects
 - See also* fields
 - creating 14-12, 14-13
 - SQL tables 14-27
 - data type, determining 14-121
 - getting 14-69
 - refreshing 14-109
 - field templates (Paradox) 14-104
 - FIELD() 12-45
 - fieldName property 14-69
 - fields 14-12
 - See also* Field objects; specific field types
 - adding 13-8, 20-22
 - assigning values 14-14
 - asterisks in 12-78
 - automatic lookup values 14-86, 14-87

- browse objects 15-96
- changing data 5-28, 12-77
- copying multiple 12-25, 12-28
- counting 12-46
- default values 14-64, 14-96
- definitions, displaying 20-17
- deleting 13-8, 20-22
- determining values 14-125
- empty 12-57
- filling with blanks 12-14
- freezing 12-18
- identifying 14-69
- length 12-47
 - maximum 14-78
- locking 14-15
- maximum value 14-92
- minimum value 14-92
- multi-line input 15-16
- passing as parameters 5-28
- read-only 14-107, 15-97
- required 14-112
- single-line input 15-18
- sorting on multiple 12-112
- structure-extended
 - tables 12-37
 - updating 12-79
 - widths, changing 20-22
- fields array 4-16
- fields list 12-42
 - browse objects 15-96
 - clearing 12-21
 - defining 12-95
 - new tables 12-118
 - returning 12-46
- fields property 14-69, 15-96
- fields() method 10-19
- File class 11-3
- file extensions
 - DB 14-1
 - DBF 14-1
 - script vs. byte code 2-16
- file indicators
 - beginning 12-15
 - end 12-43
- file information methods
 - accessDate() 11-6
 - createDate() 11-11
 - createTime() 11-11
 - date() 11-12
 - eof() 11-16
 - error() 11-18
 - exists() 11-19
 - shortName() 11-38
 - size() 11-38
 - time() 11-39
- file locks
 - automatic 12-48, 12-61, 12-85, 14-15
 - disabling 12-102
- file names 12-1
 - changing 11-33, 11-34
 - creating 11-21
 - default 12-24
 - getting 14-69
 - returning 11-23, 11-30
 - DOS 11-38
 - full paths 11-37
- file pointers 11-24
 - determining position 11-29
 - moving 11-24, 11-36
 - returning location 11-16
- file utilities and information
 - COPY FILE 11-9
 - CREATE FILE 20-7
 - DELETE FILE 11-14
 - DISPLAY FILES 11-15, 11-26
 - ERASE 11-17
 - FILE() 11-19
 - GETDIRECTORY() 11-22
 - RENAME 11-34
- file utility methods
 - close() 11-7
 - copy() 11-8
 - delete() 11-13
 - flush() 11-20
 - gets() 11-23
 - open() 11-27
 - puts() 11-31
 - read() 11-32
 - readln() 11-33
 - rename() 11-34
 - seek() 11-36
 - write() 11-41
 - writeln() 11-42
- FILE() 11-19
- filename property 14-69
- filename variable 12-1
- files
 - attributes
 - DOS 10-15
 - extended 10-16
 - backing up 20-22
 - binary 12-26
 - coverage analysis 20-25
 - reading from 12-80
 - buffers, flushing 11-20
 - closing 11-7, 20-2, 21-9
 - copying 11-8, 11-9, 11-41
 - coverage 20-13, 20-25
 - creating and opening 11-9
 - date and time stamps,
 - returning 11-12
 - date created, returning 11-11
 - deleting 11-13, 11-14, 11-17
 - directory listings 11-14
 - finding 11-37
 - checking existence 11-19
 - handles, operating
 - system 11-25
 - header 22-10
 - include 22-10
 - information, getting 10-14, 10-16, 11-14
 - date created 11-11
 - date last modified 11-12
 - existence of file 11-19
 - file pointer location 11-16
 - last error number 11-18
 - locks 12-61, 20-4
 - name (DOS) of file 11-38
 - size 11-38
 - time created 11-11
 - time last modified 11-39
 - linking 20-1
 - locking automatically 12-48, 12-61, 12-85, 14-15
 - disabling locks 12-102
 - low-level 20-2
 - memory 5-37, 5-38
 - menu definition 15-119
 - object 20-3
 - opening 11-27
 - overwriting 12-108
 - path, determining 11-29
 - position, returning 12-15, 12-43
 - protecting 21-15
 - query 12-110
 - renaming 11-33, 11-34
 - saving 11-20
 - selecting 11-22
 - streaming output 18-18
 - temporary 11-21, 12-123
 - SORT and 12-112
 - text, reading from 11-33
 - types, supported 12-7
 - unlocking 12-105
 - writing characters to 11-41, 11-42
- file-sharing modes 12-94
- fill() method 10-20
- Filter mode
 - canceled 14-46
 - entering 14-46

- filter property 14-70
- filtering 14-37, 14-46, 14-56
 - clearing 14-59, 14-71
 - criteria
 - controlling 14-71
 - determining if specified 14-93
 - defined 14-70
- filterOptions property 14-71
- filters, queries, and views
 - CREATE QUERY 20-11
 - DISPLAY 12-42
 - MODIFY QUERY 20-21
 - MODIFY VIEW 20-21
 - SET FILTER 12-98
 - SET VIEW 12-110
- FINALLY 5-19
- financial transactions 12-18
 - future value 7-6
 - payments 7-10
 - present value 7-11
- finding *See* locating
- FINDINSTANCE()
 - method 5-19
- findKey() method 14-71
- findKeyNearest()
 - method 14-72
- first property 15-98
- first() method 14-73
- firstKey property 10-21
- firstPageTemplate
 - property 17-24
- fixed property 17-24
- FLDCOUNT() 12-46
- FLDLIST() 12-46
- FLENGTH() 12-47
- float values
 - adding 12-114
 - returning
 - future value 7-6
 - present value 7-12
- FLOCK() 12-48
 - LOCK() vs. 12-85
 - SET REPROCESS and 12-108
- FLOOR() 7-5
 - compared (table) 7-7
- FLUSH 12-49
- flush() method
 - data buffers 14-73
 - files 11-20
- FNAMEMAX() 11-20
- focus 15-98
 - color options 15-75
 - gained 15-131
 - getting 15-56
 - losing 15-127, 15-135
 - moving 15-170, 16-36
 - arrow keys and 15-104
 - restricting 15-177
 - setting 15-163
- FocusBitmap property 15-99
- fontBold property 15-100
- fontName property 15-100
- fonts 18-2
 - scaling 15-159
 - selecting 15-159, 16-20
 - setting font name 15-100
 - setting point size 15-101
 - text attributes 15-100, 15-101
- fonts commands
 - DEFINE COLOR 16-15
 - GETFONT() 16-20
- Fonts dialog box, calling 16-20
- fontSize property 15-101
- fontStrikeout property 15-101
- fontUnderline property 15-101
- footer band 17-6
- footerBand property 17-25
- footers, printing 18-9
- for loops
 - SLEEP vs. 16-45
- FOR() 12-49
- FOR...ENDFOR 5-20
- FOR...NEXT 5-20
- Form class 15-20
 - events (table) 15-22
 - methods (table) 15-22
 - properties (table) 15-20
- form commands
 - CREATE FORM 20-8
 - CREATE MENU 20-10
 - CREATE POPUP 20-10
 - MODIFY
 - APPLICATION 20-21
 - MODIFY FORM 20-21
 - MODIFY MENU 20-21
 - MODIFY SCREEN 20-21
 - MSGBOX() 16-26
 - SET CUAENTER 16-36
- Form designer 20-8
- form files 20-8
- Form objects
 - See also* forms
 - creating 15-23
 - grouping 15-104
 - tabbing order 15-63
- form property 15-101
- Form wizard 20-8
- formats
 - date 8-22
 - returning 8-9, 8-11, 8-19
 - specifying 8-23
- function templates 15-103, 18-2
- international 8-23
- numeric data
 - currency symbols 7-15
 - decimal separator 7-16
 - thousands separator 7-17
- picture templates 7-18, 15-149, 18-1
- text 15-103
 - size 15-159
 - time 8-23, 8-25
- formatting text 15-148
- formfeeds 18-7
- forms
 - See also* Form objects
 - active page, setting 15-144
 - ActiveX controls, including 15-4
 - adding Edit and Windows menus 16-6
 - adding menus 16-3, 16-8
 - top-level 16-6
 - adding pop-up menus 15-150
 - anchoring objects 15-61
 - area borders 15-40
 - associated views 15-180
 - background image, setting 15-63
 - centering 15-62
 - changing 20-8
 - closing 15-68, 15-71, 15-74, 15-95
 - event handling 15-129
 - color options 15-75
 - Control menu 15-170
 - control tips 15-166, 15-168
 - creating 20-8
 - designing 15-89, 20-27
 - display states 15-184
 - displaying text in
 - HTML 15-51
 - elements array 15-93
 - listing components 15-93
 - loading 15-130
 - MDI 15-23, 15-117
 - menu definition files 15-119
 - menus 16-3
 - modal 15-153
 - moving 15-122, 15-138
 - moving through 15-109, 15-159, 15-182

- multi-page 15-23, 15-49
 - counting pages 15-143
 - getting page number 15-144
- object references array 15-93
- opening 15-143, 15-153
 - event handling 15-139
- overview 15-1
- printing 15-151
- referencing 15-102
- refreshing 15-155
- scrolling 15-109, 15-159, 15-182
- SDI 15-23
- sizing 15-116, 15-120, 15-184
 - automatically 15-63
 - event 15-142
 - preventing 15-166
- specifying topmost 15-174
- submitting 15-88, 15-141
- toolbars 16-10
- toolbuttons 16-11
- forward slash symbol 4-17
- FOUND() 12-50
 - CONTINUE and 12-23, 12-63
 - SEEK and 12-88
 - SELECT and 12-91
 - SET NEAR and 12-104
- freeing memory 5-7
 - indexes and 12-40
- French date format 8-23
- frequency (bell) 20-24
- FUNCTION 5-21
 - CLASS and 5-6
 - PROCEDURE vs. 5-32
- function calls 4-12
 - BDE 14-76
 - external 19-12
 - internal to program file 5-41
 - program to program 5-41
 - retrying 21-16
- function keys
 - command execution 16-38
 - current setting 5-42, 5-43
 - default assignments 16-39
- function pointers 2-12, 4-12
- function property 15-103
- function symbols 6-30
- function templates 15-103, 18-2
- functions 5-6
 - accessing values 5-27
 - declaring 5-21
 - defined 2-11
 - external

- accessing 19-12
- inline 22-4
- key expressions and 12-54
- maximum per program 5-22
- naming 5-22
- prototypes for external 19-12
- return values 5-38
- running 5-10
- FUNIQUE() 11-21
- future value, returning 7-6
- FV() method 7-6

G

- GENERATE 12-51
- generating random numbers 7-13
- generic controls 15-35
- German date format 8-23
- GETCOLOR() 16-19
 - DEFINE COLOR and 16-16
- getDate() method 8-13
- getDay() method 8-14
- GETDIRECTORY() 11-22
- GETENV() 11-22
- GETFONT() 16-20
- getHours() method 8-14
- getMinutes() method 8-15
- getMonth() method 8-15
- gets() method 11-23
- getSchema() property 14-74
- getSeconds() method 8-16
- getTextExtent() method 15-103
- getTime() method 8-16
- getTimezoneOffset() method 8-17
- getYear() method 8-18
- global values 5-33
- GO 12-51
 - BOOKMARK() and 12-16
- goto() method 14-75
- graphics 12-26
 - See also* bitmaps; images
 - centering 15-58
 - displaying 19-25
 - printing 19-26
 - pushbuttons 15-89, 15-91, 15-99
 - nonselected 15-176
 - storing
 - binary fields 12-26
 - memo fields 12-27
- greater-than operator 4-8
- greater-than or equal to operator 4-8
- Grid class 15-24

- events (table) 15-25
- methods (table) 15-25
- properties (table) 15-24
- GridColumn class 15-26
 - properties (table) 15-26
- gridLineWidth property 15-104
- grids 15-24
- Group class 17-6
 - methods (table) 17-7
 - properties (table) 17-6
- group property 15-104
- groupBy property 17-25
- grouping data 17-7, 17-25
- grouping objects 15-104
- grouping operator 4-12
- groups
 - footers in 17-25
 - headers in 17-27
 - identifying 17-27
 - rendering in 17-20
 - report-level 17-10, 17-38
 - returning calculations 17-13 to 17-19
- grow() method 10-22

H

- handle property 11-25, 14-76, 15-105
- handles, file 11-25
- handling exceptions 5-2, 5-46
 - catch block and 5-5
- hasButtons 15-105
- hatched (blended) backgrounds 15-146
- header band 17-6
- header files 22-10
 - changing 22-11
- headerBand property 17-27
- headerEveryFrame property 17-27
- headers
 - printing 18-9
 - repeating over StreamFrames 17-27
- height property 15-107
- HELP 20-19
- Help system
 - activating 20-19
 - keywords, specifying 15-108
 - topics, specifying 15-108
- Help, online 1-1
- helpFile property 15-108
 - helpID and 15-108
 - onHelp and 15-132
- helpID property 15-108

- helpFile and 15-108
- onHelp and 15-132
- heterogeneous joins 13-16
- hexadecimal numbers
 - decimal equivalents 9-7
 - returning 9-7
- hiding components 15-181
 - in reports 17-21, 17-41
- hints 15-166, 15-168
- HOME() 11-25
- horizontal scroll bars 15-180
- hours
 - returning 8-14
 - setting 8-26
- HTML text 15-49
- HTOI() 9-7
- hWnd property 15-109

I

I/O

- display widths, memo fields 12-103
- environment messages 20-32
- printing 18-7
 - page formatting 18-9

I/O commands

- ? command 18-1, 18-2
- ?? command 18-5
- CREATE LABEL 20-9
- CREATE REPORT 20-12
- MODIFY LABEL 20-21
- MODIFY REPORT 20-21
- SET ALTERNATE 18-13
- SET HEADINGS 12-99
- SET SPACE 18-20

Icon property 15-110

icons 15-110

ID checking, language drivers 21-18, 21-19

ID property 15-110

- NETWORK() and 21-11

ID() 21-6

IDAPI errors 21-4, 21-5

identifiers

- defining 22-2
 - without replacement text 22-3, 22-4
- multiple programs 22-11
- objects 15-110
- replacing with specified values 22-3
- undefining 22-4, 22-12
- Visual dBASE version number 22-13

IF 5-23

- DO CASE vs. 5-13
- ELSE and 5-17
- ELSEIF and 5-17
- IIF() vs. 5-25
 - multiple ELSEIFs 5-24
- #if directive 22-6
- #ifdef directive 22-8
- #ifndef directive 22-9
- IIF() method 5-24
- Image class 15-27
 - properties (table) 15-27
- Image objects 15-28
- images
 - See also* bitmaps; graphics
 - displaying 15-28
 - enlarging 15-57
 - linking bitmap to form component 15-87
 - setting background 15-63
 - shrinking 15-57
 - sizing and positioning 15-57, 15-121
- #include directive 22-10
- INCLUDE directory 22-10
- include files 22-10
 - changing 22-11
- incompatible data types 20-23
- increment operators 4-6
- incrementing spin box values 15-169
- _indent 18-21
- _rmargin and 18-38
- _wrap and 18-39
- INDEX 12-52
 - SEEK and 12-89
 - SET EXCLUSIVE and 12-95
 - SET INDEX and 12-100
 - SORT vs. 12-112
- index files
 - allocating memory 20-24, 20-29
 - closing 12-22
 - copying 12-25
 - creating 12-28, 12-36
 - deleting 12-39, 13-12
 - information, getting 20-16
 - multiple 12-116
 - names, returning 12-66, 12-69
 - opening 12-99, 12-122
 - temporary 12-112
- index operator 4-10
- indexes
 - copying tables 12-25
 - creating 12-52, 12-58, 13-9

- deleting 13-12
- master 12-57
 - returning names 12-70
 - specifying 12-100, 12-104, 12-122
- number of active 12-116
- numeric fields 6-25
- order, reversing 12-55
- processing speed 20-29
- rebuilding 12-77, 12-82
 - Standard tables 14-109
- replacing data 12-78
- SCAN and 12-87
- sort order, setting 12-53
- tags 12-57
 - number, returning 12-116
- updating 20-23
 - automatically 12-56

indexing and sorting

CLOSE INDEXES 12-22

DELETE TAG 12-40

FOR() 12-49

INDEX 12-52

KEY() 12-58

MDX() 12-65

NDX() 12-69

ORDER() 12-70

REINDEX 12-75

SET IBLOCK 20-29

SET INDEX 12-99

SET KEY TO 12-101

SET ORDER 12-104

SET UNIQUE 12-110

SORT 12-111

TAG() 12-115

TAGCOUNT() 12-116

TAGNO() 12-116

UNIQUE() 12-119

indexName property (Rowset) 14-76

indexName property (UpdateSet) 14-77

indirection operator 4-12

infinity, returning 7-20

information resources 1-1

initializing

- DLLs 19-16
- memory variables 5-26, 5-32, 5-33, 12-21
 - during program suspension 21-23
- values in arrays 10-20

initiate() method 19-16

initiation handlers 16-33

INKEY() 16-21
 NEXTKEY() and 16-28
 inline functions 22-4
 input *See* I/O
 input fields
 multi-line 15-16
 single-line 15-18
 INSERT statement (SQL) 13-13
 insert() method 10-25
 INSPECT() 20-19
 Inspector
 opening 20-19
 instantiating classes 5-6
 instantiation 4-10
 INT() 7-7
 compared (table) 7-7
 integers
 See also numbers
 decimal separator 7-16
 returning 7-7, 7-18
 equality 7-3, 7-5
 interest rates
 future value 7-6
 payments 7-11
 present value 7-12
 international date/time
 formats 8-23
 interrupting SLEEP 16-44, 16-45
 interrupts, Esc key 16-37
 interval property 8-18
 invalid data entry 15-178, 20-24
 involution, returning
 exponents 7-8
 isAlpha() method 6-11
 ISALPHA() 6-11
 ISBLANK() 12-57
 AVERAGE and 12-11, 12-114
 BLANK and 12-14
 EMPTY and 5-18
 isKey() method 10-28
 isLastPage() method 17-28
 isLower() method 6-12
 ISLOWER() 6-11
 isolation levels 14-77
 See also transactions
 isolationLevel property 14-77
 isRecordChanged()
 method 15-113
 ISTABLE() 12-58
 isUpper() method 6-13
 ISUPPER() 6-12
 Italian date format 8-23
 italic attributes 15-100
 ITOH() 9-7

J

Japanese date format 8-23
 joins
 heterogeneous 13-16
 live, restrictions 13-7

K

kerning, adjusting 17-42
 key codes 16-32
 Key event 15-113
 key expressions
 linking tables 12-106
 matching 12-88, 12-103
 returning 12-58
 key fields
 changing 12-78
 key values
 duplicate 12-109
 restricting 12-101, 12-110
 searching on 12-90
 key violations, handling 14-78
 KEY() 12-58
 KEYBOARD 16-24
 keyboard
 accelerators 16-43
 default assignments 16-39
 keyboard event commands
 INKEY() 16-21
 NEXTKEY() 16-28
 ON ESCAPE 16-29
 SET ESCAPE 16-37
 SET FUNCTION 16-38
 keyboard() method 15-114
 KEYMATCH() 12-59
 keys (associative arrays)
 finding next 10-29
 strings, searching for 10-28
 keystrokes
 assigning
 command
 execution 16-29, 16-31,
 16-38
 interrupts 16-37
 clearing buffer 16-15
 evaluating 15-113, 15-127,
 15-132, 15-133
 getting 16-21
 simulating 15-114, 16-36
 values, returning 16-21,
 16-28
 keyViolationTableName
 property 14-78
 keywords
 abbreviating 5-43

Help systems 15-108
 local SQL 13-3
 naming restrictions and 5-22

L

labels
 creating 17-4, 17-20
 designing 20-27
 language drivers 21-4, 21-7
 current, returning 21-7
 ID checking 21-18, 21-19
 ISALPHA() and 6-11
 ISLOWER() and 6-12
 ISUPPER() and 6-12
 LOWER() and 6-17
 primary/secondary
 weights 12-94
 PROPER() and 6-18
 SOUNDEX() and 6-23
 UPPER() and 6-31
 language elements 5-1
 last() method 14-78
 lastIndexOf() method 6-13
 layouts, report page 17-8
 LDRIVER() 21-7
 leading property 17-29
 leading spaces
 deleting 6-15, 6-17
 left property 15-115
 bottom and 15-67
 left() method 6-14
 LEFT() 6-14
 leftTrim() method 6-15
 LEN() 6-15
 LEFT() and 6-14
 SUBSTR() and 6-28
 length property 14-78
 String objects 6-15
 less-than operator 4-8
 less-than or equal to
 operator 4-8
 libraries 5-39
 LIKE() 6-16
 DIFFERENCE() vs. 6-8
 SOUNDEX() and 6-23
 Line class 15-28
 properties (table) 15-29,
 15-36
 line comment symbol 4-17
 line lengths, memo fields 12-68
 line spacing, setting 17-29
 Line objects *See* lines
 linear control structures 5-12,
 5-23, 5-24
 linefeeds 18-7

- automatic 18-8
- character, counting 6-14, 6-15, 6-21
 - substrings 6-5, 6-28
- files 11-24, 11-31
- lineNo property 15-115
- LINENO() 21-8
 - PROGRAM() and 21-14
- lines
 - creating 15-28
 - setting row position 15-67
 - specifying patterns 15-147
- link expressions 12-75
- linkFileName property 15-116
- linking
 - form component to
 - bitmap 15-87
 - form component to Field object (bitmap) 15-87
- linking and relating
 - RELATION() 12-75
 - SET RELATION 12-105
 - SET SKIP 12-109, 12-120
 - TARGET() 12-117
 - UPDATE() 12-120
- linking files 20-1
- linking form components 15-84
- linking report
 - components 17-12
- linking tables 12-106
- links 14-3
 - DDE 19-16, 19-27, 19-28
 - disabling 19-27
 - OLE 12-84, 15-116
- LIST 12-60
 - DISPLAY vs. 12-42
 - EOF() and 12-44
 - SET HEADINGS and 12-99
 - SET PRINTER and 18-18
- list boxes 15-49
 - counting prompts 15-82
 - current prompt 15-83
 - multiple selection,
 - setting 15-122
 - selecting options 15-85
 - selecting prompts 15-141
 - sorting 15-167
- LIST FILES 11-26
 - DISPLAY FILES vs. 11-16
 - SET SEPARATOR and 7-18
- LIST/DISPLAY
 - commands 20-20
- ListBox class 15-30
 - events (table) 15-30
 - methods (table) 15-30
 - properties (table) 15-30
- ListBox objects 15-31
 - See also* list boxes
- lists
 - combo box styles 15-170
 - programming
 - multiple selection 15-122
 - single selection 15-13
 - scrollable 15-31
- literal array elements 4-18
- literal characters 6-16
- literal dates 4-18
- literal strings 4-16
- live joins, restrictions (local SQL) 13-7
- live property 14-79
- live queries
 - restrictions (local SQL) 13-7
 - specifying 14-112
- LKSYS() 12-61
 - CONVERT and 20-5
- _lmargin 18-22
 - _alignment and 18-20
 - _ploffset and 18-34
 - _rmargin and 18-37
 - _wrap and 18-39
- LOAD DLL 19-16
 - RELEASE DLL and 19-24
- loading components 15-130
- loading forms 15-130
- LOCAL 5-25
- local variables
 - declaring 5-25
 - as static 5-44
 - initializing 5-26, 5-32
- LOCATE 12-62
 - CONTINUE and 12-23
 - FOUND() and 12-50
 - SEEK vs. 12-88, 12-89
- Locate mode
 - canceling 14-47
 - entering 14-46
- locateNext() method 14-79
- locateOptions property 14-80
- locating data 14-24, 14-38, 14-46
 - See also* searching
 - controlling criteria 14-80
 - next match 14-79
 - similar spellings 6-23
- lock property 14-80
- LOCK() 12-64
 - RLOCK() vs. 12-85
- LockField class 14-15
 - properties (table) 14-15
- lockRetryCount property 14-81
- lockRetryInterval
 - property 14-82
- lockRow() method 14-82
- locks
 - field 14-15
 - file 12-48, 12-61, 12-85, 14-15
 - disabling 12-102
 - getting user
 - information 14-124
 - record 12-64, 12-84, 14-80, 14-82
 - automatic 14-15
 - information,
 - getting 12-61, 20-4
 - releasing 12-105
 - retry attempts 14-81
 - retry intervals,
 - setting 14-82
 - releasing 14-122
 - retry messages 12-108
 - rowsets 14-25, 14-84
 - table 12-102
 - information,
 - getting 12-61, 20-4
 - releasing 12-105
- lockSet() method 14-84
- LOG() 7-7
 - EXP() vs 7-5, 7-8
- LOG10() 7-8
- logarithms 7-8
 - base e 7-5
- logical expressions 5-24
- logical fields
 - blank values 12-14
 - converting to character 20-22
 - converting to numeric 20-22
- logical operators 4-7
- logical values, defined 2-3
- login dialog box, preventing
 - appearance 14-86
- login name, returning 14-125
- login() method 14-85
- loginString property 14-86
- LOGOUT 21-8
- lookup tables 14-88
- lookup values 14-86, 14-87
- LOOKUP() 12-64
 - FOUND() and 12-50
 - INDEX and 12-56
- lookupRowset property 14-86
 - lookupSQL vs. 14-87
- lookupSQL property 14-87
 - lookupRowset vs. 14-86
- lookupTable property 14-88
- lookupType property 14-88

- LOOP 5-26
- loop counter 5-20
- looping, in associative
 - arrays 10-22, 10-29
- loops 5-14, 5-15, 5-20, 12-86
 - conditional statements and 5-26
 - exiting 5-14, 5-18
- losing data 12-71, 12-78, 20-22
 - minimizing loss 12-92
- LOWER() function (SQL) 13-6
- LOWER() 6-17
 - AT() and 6-5, 6-18
 - LIKE() and 6-16
 - scan() and 10-35
- lowercase letters
 - converting to
 - uppercase 6-29, 6-31, 12-55
 - first letter 6-17, 6-29
 - sorting data 12-112
 - testing for 6-11, 6-12
- low-level files 20-2
- LTRIM() 6-17
 - TRIM() vs. 6-31
- LUPDATE() 12-65

M

- macro operator 4-13
- macro substitution 4-12, 4-13
- macros (DDE) 19-11
- magnitude (defined) 7-1
- magnitudes 15-44
- main menus 16-6
- manipulating data 12-111
- manipulating dates 8-7
- marginBottom property 17-29
- marginHorizontal
 - property 17-30
- marginLeft property 17-30
- marginRight property 17-30
- margins, setting 18-16
 - both sides 17-30
 - both top and bottom 17-31
 - bottom (in reports) 17-29
 - changing from page to
 - page 17-32, 17-35
 - left (in reports) 17-30
 - right (in reports) 17-30
 - top (in reports) 17-31
- marginTop property 17-31
- marginVertical property 17-31
- marking records for
 - deletion 12-40
 - removing marks 12-72
- master index 12-57

- returning names 12-70
- specifying 12-100, 12-104, 12-122
- masterFields property 14-89
- masterRowset property 14-91
- masterSource property 14-91
- MAX() 7-8
- MAX() function (SQL) 13-6
- Maximize buttons,
 - enabling 15-116
- maximize property 15-116
- maximum property 14-92
- maximum values,
 - returning 12-19
- maxLength property 15-117
- MD 11-26
- MDI applications *See* MDI forms
- MDI forms 15-23, 16-42
 - testing 15-117
- MDI property 15-117
 - Maximize and 15-117
 - Minimize and 15-120
 - moveable and 15-122
- .MDX files
 - allocating memory 20-29
 - copying 12-25
 - creating 12-28, 12-36
 - name, returning 12-65
- MDX() 12-65
- MDY() 8-19
- mean average, returning in
 - reports 17-13
- MEMLINES() 12-67
- memo data, defined 2-4
- memo fields 20-22
 - allocating memory 20-24, 20-31
 - case, converting 6-17, 6-31
 - changing 12-77
 - copying 12-9, 12-25, 12-26
 - text files to 12-10, 12-83
 - to text files 12-25
 - creating 20-25, 20-31
 - display width, setting 12-103
 - empty 6-15
 - line lengths 12-68
 - number of lines 12-67
 - overwriting 12-10, 12-27, 12-83
 - text, returning 12-68
- memo files, deleting (local SQL) 13-12
- memory
 - allocating
 - indexes 20-24, 20-29

- memo fields 20-24, 20-31
 - checking available 21-9
 - freeing 5-7
 - indexes and 12-40
 - managing 5-8, 5-36
 - releasing objects from 5-36, 5-37
 - variables, overriding 22-3
- memory blocks 20-31
 - size, changing 20-29
- memory files, creating 5-37, 5-38
- memory variables 12-79
 - clearing 5-7, 5-35, 5-37
 - copying 5-37
 - current settings,
 - program 5-42, 5-43
 - decrementing/
 - incrementing 5-20
 - deleting 5-35
 - information, getting 20-15
 - initializing 5-26, 5-32, 5-33, 12-21
 - during program
 - suspension 21-23
 - objects 5-9
 - passing as parameters 5-27
 - preserving 5-37
 - public 5-33, 5-37
 - saving 5-37, 5-38
 - scope 5-25, 5-33
 - static 5-44
 - storing data 12-113
 - storing expressions 5-45
 - storing values 12-11, 12-19, 12-114
 - testing 5-20
- memory variables commands
 - CLEAR MEMORY 5-7
 - LOCAL 5-25
 - PRIVATE 5-31
 - PUBLIC 5-33
 - RELEASE 5-35
 - RESTORE 5-37
 - SAVE 5-38
 - STATIC 5-43
 - STORE 5-45
- memory() method 21-9
- MEMORY() 21-9
- Menu class 16-3
- menu commands
 - choosing 16-43
- menu definition files 15-119
- Menu designer 15-119, 20-10
- menu objects 16-4

- MenuBar class 16-6
 - events (table) 16-6
 - methods (table) 16-7
 - properties (table) 16-6
- menuFile property 15-119
- menus
 - adding checkmarks 16-13
 - assigning actions 16-5
 - checkmarks 16-13
 - creating 15-150, 16-3
 - Edit menu, standard 16-7
 - generating 16-3, 16-6
 - initializing 16-34
 - popup 16-8
 - separators 16-35
 - Window 16-47
- MESSAGE() 21-10
- messages
 - confirmation 12-108
 - displaying 16-44, 20-32
 - current environment 20-16
 - memory variables 20-15
 - environment
 - information 20-16, 20-18, 20-32
 - file locking 12-108
 - status bar 15-169, 16-40
- methods 5-6
 - accessing
 - by name 4-11
 - by value 4-10
 - calling 4-12
 - creating 5-5
 - defined 2-14
 - maximum per program 5-22
 - referencing 4-11
 - visual components, common (table) 15-3
- MIN() 7-9
- MIN() function (SQL) 13-6
- Minimize buttons, enabling 15-120
- minimize property 15-120
- minimum property 14-92
- minimum values, returning 12-19
- minutes
 - returning 8-15
 - setting 8-26
- mismatched data types 4-8
- MKDIR 11-27
 - MD vs. 11-26
- MLINE() 12-68
- MOD() method 7-10
- modal forms 15-153, 15-174
- mode, determining
 - current 14-119
- modified property 14-93
- MODIFY... commands
 - SET DESIGN and 20-26
- MODIFY APPLICATION 20-21
- MODIFY COMMAND
 - SET DEVELOPMENT and 20-27
- MODIFY FORM 20-21
- MODIFY LABEL 20-21
- MODIFY MENU 20-21
- modify property 15-120
- MODIFY QUERY 20-21
- MODIFY REPORT 20-21
- MODIFY SCREEN 20-21
- MODIFY STRUCTURE 20-21
 - CREATE...FROM vs. 12-36
- MODIFY VIEW 20-21
- MODIFY/CREATE
 - commands 20-20
- modifying *See* changing
- modules 5-22
- modulus, returning 7-10
- modulus operator 4-6
- monetary values 7-15
- MONTH() 8-19
- months
 - returning 8-15, 8-19
 - setting 8-26
- morphing 14-42, 14-53, 15-149
- mouse buttons
 - clicking 15-135
 - middle button 15-136, 15-140
 - twice 15-134, 15-136, 15-140
 - releasing 15-135, 15-136, 15-141
- mouse event commands
 - INKEY() 16-21
- mouse events
 - assigning 15-135, 15-136, 15-140, 15-141
 - double-clicks 15-134, 15-136, 15-140
 - moving forms 15-122
- mouse pointer
 - changing 15-121, 15-137
 - moving 15-121, 15-137
- mousePointer property 15-121
- move() method 15-121
- moveable property 15-122
- moving
 - file pointers 11-24, 11-36
 - forms 15-122, 15-138
 - record pointers 12-51, 12-87, 12-92, 12-111
 - linked tables 12-109
 - moving through forms 15-109, 15-159, 15-182
 - moving through tables 12-51, 12-109
- MSGBOX() 16-26
- multi-line comment blocks 4-17
- multi-line input fields 15-16
- multi-line statements 4-17
- multi-page containers 15-31
- multi-page forms 15-23, 15-49
 - counting pages 15-143
 - getting page number 15-144
- multiple conditions, testing 5-24
- Multiple Document Interface *See* MDI forms
- multiple documents, opening 15-23, 15-117
- multiple ELSEIF statements 5-24
- multiple fields
 - copying 12-25, 12-28
 - sorting data 12-112
- multiple index files, returning 12-116
- multiple programs
 - compiling 22-11
 - identifiers 22-11
- multiple property 15-122
- multiple selection, allowing 15-31
- multiplication operator 4-6
- multiuser environments
 - changing data 12-20, 12-48, 12-64, 12-84
 - deleting records 12-124
 - errors 21-13
 - file-sharing modes 12-94
 - releasing locks 12-105
 - screens, refreshing 12-105
 - setting locks 12-48, 12-64, 12-84, 12-102
 - retry messages 12-108
 - testing for 21-10
 - transactions 12-12
 - committing 12-23
 - rolling back 12-86
 - updating data 12-48, 12-85
 - user names, returning 21-6

N

name property 15-123
names
 columns, in local SQL 13-2
 databases 12-37
 DDE applications,
 returning 19-26, 19-28
 file, returning DOS 11-38
 functions 5-22
 index files, returning 12-66,
 12-69
 resolving conflicts 4-13
 table
 changing 12-76
 in local SQL 13-2
 returning 12-38, 12-117
 work areas 4-16
nativeObject property 15-124
natural logarithms 7-8
 base e 7-5
.NDX files
 name, returning 12-69
 specifying as master 12-122
NDX() 12-69
negating an operand 4-5
negative values
 absolute 7-1
 finding 7-18
net present values 12-19
network drives 11-41
NETWORK() 21-10
NEW operator 4-10, 5-6
 DEFINE and 5-9
new tables, creating 12-37
next() method 14-94
NEXTKEY() 16-28
 INKEY() and 16-23
nextKey() method 10-29
nextObj property 15-124
nextPageTemplate
 property 17-32
non-editable text 15-171
non-operational symbols 4-16
not equal to operator 4-8
NOT operator 4-7
NoteBook class 15-31
 events (table) 15-32
 properties (table) 15-31
notify() method 19-17
notifyControls property 14-95
null values 5-18
 defined 2-4
number sign symbol 4-19
numbers

See also float values; integers
adding 12-19, 12-114, 12-117
averaging 12-11
constants 22-3
dividing 7-10
hexadecimal
 decimal equivalents 9-7
 returning 9-7
incrementing 15-169
negative 7-18
 absolute values 7-1
 random 7-13
 rounding 6-25, 7-13
 sign, determining 7-18
numeric constants
 pi 7-11
numeric data
 bitwise operations
 return values, getting 9-5
 shift bits 9-2
 comparing 7-8, 7-9
 converting characters 20-22
 converting to character 6-25
 converting to strings 6-25
 formatting
 currency symbols 7-15
 decimal separator 7-16
 thousands separator 7-17
 key expressions 12-55
 precision, setting 7-17
 replacing 12-78
 returning
 characters as 20-22
 logical 20-22
 truncating 7-7
numeric data commands
 SET CURRENCY 7-15
 SET DECIMALS 7-15
 SET POINT 7-16
 SET PRECISION 7-16
 SET SEPARATOR 7-17
numeric data functions
 ABS() 7-1
 ACOS() 7-1
 ASIN() 7-2
 ATAN() 7-2
 ATN2() 7-3
 CEILING() 7-3
 COS() 7-4
 DTOR() 7-4
 EXP() 7-5
 FLOOR() 7-5
 INT() 7-7
 LOG() 7-7
 LOG10() 7-8

MAX() 7-8
MIN() 7-9
PI() 7-11
RANDOM() 7-12
ROUND() 7-13
RTOD() 7-14
SIN() 7-19
SQRT() 7-19
TAN() 7-20
numeric data methods
 FV() 7-6
 MOD() 7-10
 PAYMENT() 7-10
 PV() 7-11
 SIGN() 7-18
numeric fields
 blank values 12-20
 indexing 6-25
 precision, setting 14-104
numeric operators 4-6

O

Object class 5-3
object classes
 creating 5-6
 identifying members 5-7
object files 20-3
object operators 4-10 to 4-11
object pointers 15-98
object references 5-28
objects
 adding borders 15-66
 anchoring 15-61
 clearing from memory 5-7,
 5-36
 closing 14-34
 colors, setting 15-75
 creating 4-10, 5-3, 5-9, 15-34
 default 15-88
 defined 2-14
 displaying data 15-96
 finding current 15-56
 focus 15-98
 getting 15-56
 moving 15-170, 16-36
 arrow keys and 15-104
 restricting 15-177
 grouping 15-104
 height 15-107
 varying 17-24, 17-42
 left edge position 15-115
 passing as parameters 5-28
 redefining properties 5-34
 referencing
 data modules 14-9, 14-108

- rowsets 14-114
 - with numeric values 15-110
- releasing from
 - memory 5-36, 5-37
- returning 5-19
- right edge position 15-157
- selecting 15-94, 15-182
- setting size 5-10
- tabbing order 15-63
- testing for active 14-34
- objects commands
 - DEFINE 5-9
 - INSPECT() 20-19
 - PLAY SOUND 19-22
 - RESTORE IMAGE 19-25
- OEM conversions 21-2
- OEM() 21-11
 - ANSI() and 21-2
- OLE 19-1
- OLE automation example 19-9
- OLE class 15-32
 - events (table) 15-33
 - methods (table) 15-33
 - properties (table) 15-33
- OLE data, defined 2-5
- OLE documents, adding 12-84
- OLE fields
 - adding OLE documents 12-84
 - information, getting 15-126
 - writing to 12-84
- OLE files 15-116
- OLE links 12-84, 15-116
- OLE objects 15-34
- OLE server applications 15-32
 - accessing 15-90, 15-162
- OleAutoClient class 19-9
- OleType property 15-126
- ON ERROR 21-12
 - LINENO() and 21-8
 - ON NETERROR vs. 21-13
 - PROGRAM() and 21-14
 - RETRY and 21-16
 - SET ERROR vs. 21-18
- ON ESCAPE 16-29
- ON KEY 16-31
- ON NETERROR 21-13
- ON PAGE 18-8
 - EJECT PAGE and 18-8
- onAbandon event 14-95
- onAdvise event 19-18
- onAppend event 14-96, 15-126
- onChange event 14-97, 15-127
- onChar event 15-127
- OnClick property
 - ShortCut and 16-43
- onClose event 14-97, 15-129
- onDelete event 14-98
- onDesignOpen event 15-130
- onEdit event 14-98
- one-to-many
 - relationships 12-109
- onExecute event 19-19
- onFormSize event 15-131
- onGotFocus event 15-131
- onGotValue event 14-99
- onHelp event 15-132
 - HelpID and 15-108
- onInitiate event 16-33
- onInitMenu event 16-34
- onKeyDown event 15-132
- OnKeyUp event 15-133
- onLeftDbClick event 15-134
- onLeftMouseDown
 - event 15-135
- onLeftMouseUp event 15-135
- online Help 1-3
- onLostFocus event 15-135
- onMiddleDbClick event 15-136
- onMiddleMouseDown
 - event 15-136
- onMiddleMouseUp
 - event 15-136
- onMouseMove event 15-137
- onMove event 15-138
- onNavigate event 14-99, 15-138
- onNewValue event 19-19
- onOpen event 14-100, 15-139
- onPage event 17-32
- onPaint event 15-140
- onPeek event 19-20
- onPoke event 19-20
- onProgress event 14-101
- onRender event 17-33
- OnRightDbClick event 15-140
- onRightMouseDown
 - event 15-140
- onRightMouseUp event 15-141
- onSave event 14-102
- onSelChange event 15-141
- onSelection event 15-141
- OnSelection property
 - ID property and 15-110
- onSize event 15-142
- onTimer event 8-20
- onUnadvise event 19-21
- onUpdate event 16-34
- OODML sections 12-1
- OPEN DATABASE 12-69
- SET DATABASE and 12-92
- open() method 11-27, 14-102, 15-143
- opening
 - databases 12-70, 14-34
 - Debugger 20-13
 - files 11-27
 - Form designer 20-8
 - forms 15-143, 15-153
 - index files 12-99, 12-122
 - Inspector 20-19
 - Menu designer 20-10
 - query files 12-110
 - Report designer 20-9, 20-12
 - Table designer 20-6, 20-22
 - tables 12-91, 12-121
 - default, setting 12-93
 - file-sharing modes 12-94
- operands 4-1
 - defined 2-5
 - negating 4-5
- operator symbols (table) 2-5, 4-1
- operators 4-1
 - addition 4-4, 4-5
 - alias 4-12, 12-107
 - assignment 4-3
 - binary 2-5
 - bitwise 9-5
 - AND 9-1
 - OR 9-4
 - shift bits 9-2
 - XOR 9-5
- call 4-12
- comparison 4-7
- concatenation 4-4, 4-5
- defined 2-5
- grouping 4-12
- indirection 4-12
- logical 4-7
- macro 4-13
- non-operational
 - symbols 4-16
- numeric 4-6
- object 4-10, 4-11
- precedence 4-2
 - overriding 4-12
- relational 4-7
- SQL (table) 13-3
- string 4-4, 4-5
- table of 2-5, 4-1
- unary 2-5
- optimizing
 - data processing 12-92
 - memory allocation 20-31
 - program execution 20-28

- search operations 12-65, 12-89
- source code 22-4
- options, compiler 22-11
- OR operator 4-7
 - bitwise 9-4
- ORDER() 12-70
- organization of documentation 1-1
- organizing data 12-54, 12-112
- OS() 11-28
- OTHERWISE 5-26
- output (reports) 17-10
- output devices 18-12
- output property 17-34
- outputFilename property 17-35
- overstriking text 18-4
- overwriting data 12-77
 - binary fields 12-26
 - confirmation messages 12-108
 - memo fields 12-10, 12-27, 12-83

P

- PACK 12-71
 - DELETE vs. 12-38
 - RECALL and 12-72
- packing tables 14-103
- packTable() method 14-103
- padding strings 6-25
- _padvance 18-23
- pageCount() method 15-143
- page-handling routines 18-8
- _pageno 18-24
- pageno property 15-144
- pages
 - alternating left and right 17-32
 - cover page, specifying 17-32
 - current page (in reports) 17-38
 - rendered in report, handling event 17-32, 17-35
 - specifying first (in reports) 17-39
 - specifying first template 17-24
 - specifying last (in reports) 17-23
 - specifying next (in reports) 17-32
- PageTemplate class 17-8
 - methods (table) 17-8, 17-36
 - properties (table) 17-8
- PageTemplate objects, creating multiple 17-9
- paging through text 15-109, 15-159, 15-182
- PaintBox class 15-34
- PaintBox objects 15-35
 - redrawing 15-131, 15-140
- Paradox tables
 - adding fields 14-17
 - constraining updates 14-60
 - creating 20-5
 - field templates 14-104
 - indexing
 - primary indexes 12-40
 - secondary indexes 12-40
 - linking 12-106
 - lookup values 14-88
 - querying 21-21
- Parameter class 14-15
 - properties (table) 14-16
- Parameter objects 14-16
 - types described 14-16
- PARAMETERS 5-27
- parameters 4-12
 - copying 5-29
 - finding number of 5-3, 5-31
 - finding specified 5-4
 - passing 5-22, 5-27, 5-28, 15-145, 20-13
 - by reference 5-27
 - by value 5-27
 - returning information on 5-3, 5-4, 5-31
 - SQL statements 14-103
 - stored procedures 14-103
 - creating 14-15
 - substitutions, local SQL 13-5
 - type, getting 14-121
- params array
 - Query object and 14-2
 - StoredProc object and 14-3
- params property 14-103
- parent, determining 5-30
- parent property 5-30
 - form property and 15-102
- parent tables 12-106
 - moving through 12-109
- parenthesis in code 4-12
- parse() method 8-21
- passing fields as parameters 5-28
- passing memory variables as parameters 5-27
- passing properties as parameters 5-27
- passwords 14-86
 - adding 14-35
- paste() method 15-146
- pasting text 15-146, 16-18
- path property 11-29
- pattern matching 6-16
- patternStyle property 15-146
- PAYMENT() 7-10
- payments
 - future value 7-6
 - present value 7-11
 - principal balance 7-10
- _pbpage 18-25
 - _pepage and 18-29
- PCOL() 18-10
 - _pcolno and 18-26
 - SET PCOL and 18-17
- _pcolno 18-26
- _pcopies 18-27
- PCOUNT() method 5-31
- _pdriver 18-27
- PdxField class 14-17
 - properties (table) 14-17
- peek() method 19-21
- _peject 18-28
- pen property 15-147
- penStyle property 15-147
- penWidth property 15-148
- _pepage 18-29
 - _pbpage and 18-25
- performance, increasing 14-122
- _pform 18-30
- phonetic matches 6-8, 6-23
- PI() 7-11
 - ACOS() and 7-2
- picture property 14-104, 15-148
- picture templates 7-18, 15-149, 18-1
- PLAY SOUND 19-22
- _plength 18-31
 - _padvance and 18-24
 - _porientation and 18-34
- _plineno 18-32
 - _plength and 18-24
- _ploffset 18-33
- point size, setting 15-101
- pointers
 - automatic load 5-40
 - file 11-24
 - determining position 11-29
 - moving 11-24, 11-36
 - returning location 11-16
 - function 2-12, 4-12
 - object 15-98

- record
 - linked tables 12-109
 - moving 12-51, 12-87, 12-92, 12-111
 - events and 15-138
 - position, returning 12-15, 12-43
 - work areas 12-91
- poke() method 19-23
- polymorphism 5-6
- population statistics 12-19
- Popup class 16-8
 - events (table) 16-9
 - methods (table) 16-9
 - properties (table) 16-9
- pop-up menus 16-8
 - creating 15-150
 - initializing 16-34
 - right mouse click 16-45
- popupMenu property 15-150
- _porientation 18-34
- position property 11-29
- positive values, finding 7-18
- posting transactions 14-60
- _ppitch 18-35
 - _pcolno and 18-26
 - _rmargin and 18-38
 - _tabs and 18-39
- _pquality 18-36
- #pragma 22-11
- precedence 4-2
 - overriding 4-12
- precision property 14-104
- prepare() method 14-105
- preprocessor
 - call chain 22-10
 - directives 22-1 to 22-13
 - symbol 4-19
 - search-and-replace operations 22-12
- preRender event 17-35
- present value, returning 7-11
- preserving memory
 - variables 5-37
- principal 7-10
 - future value 7-6
 - present value 7-11
- print() method 15-151
- printer control codes 18-17
- printer property 17-36
- Printer Setup dialog box 18-6
- printers
 - escape sequences 18-17
 - horizontal printing
 - position 18-10, 18-17
 - specifying 18-18
 - vertical printing
 - position 18-13, 18-19
- printing
 - data 12-42, 18-11
 - advancing paper 18-7, 18-8
 - page formatting 18-8
 - setting margins 18-16
 - environment
 - information 20-16, 20-18, 20-32
 - files list 11-16, 11-26
 - forms 15-151
 - graphics 19-26
 - headers and footers 18-9
 - reports 18-7
 - specifying options 17-36
 - text files 11-40
- printing commands
 - CHOOSEPRINTER() 18-6
 - EJECT 18-7
 - EJECT PAGE 18-8
 - ON PAGE 18-8
 - PCOL() 18-10
 - PRINTJOB 18-11
 - PROW() 18-13
 - SET MARGIN 18-16
 - SET PCOL 18-17
 - SET PRINTER 18-18
 - SET PROW 18-19
- PRINTJOB...ENDPRINTJOB 18-11
 - _pcopies and 18-27
 - _peject and 18-28
- PRINTSTATUS() 18-12
- PRIVATE 5-31
 - LOCAL vs. 5-25
- private variables 4-12, 5-27
 - clearing 5-35, 5-37
 - declaring 5-31
 - macro substitution and 4-13
- problem tables 14-106
- problemTableName
 - property 14-106
- PROCEDURE 5-32
- procedure calls 5-23, 5-32
 - call chain 5-12
 - retrying 21-16
 - stored procedures 14-27, 14-106
- procedureName
 - property 14-106
- procedures 5-6
 - See also* stored procedures
- closing 5-8
- compiling
 - automatically 20-27
- debugging 20-13 to 20-17, 21-14
- declaring 5-32
- processing data
 - optimizing 12-92
 - specific records 12-101
- processing speed 20-28, 20-32
 - FLUSH and 12-49
 - indexes 20-29
- program calls 5-12
 - recursive 21-12, 21-13
- program commands
 - ARGCOUNT() 5-3
 - ARGVECTOR() 5-4
 - BUILD 20-1
 - CLEAR PROGRAM 5-7
 - COMPILE 20-3
 - CREATE COMMAND 20-6
 - DO...ENDDO 5-10
 - DO CASE 5-12
 - DO WHILE 5-14
 - DO...UNTIL 5-15
 - FOR...ENDFOR 5-20
 - IF...ENDIF 5-23
 - IIF() 5-24
 - PARAMETERS 5-27
 - PCOUNT() 5-31
 - PROCEDURE 5-32
 - RETURN 5-38
 - SCAN...ENDSCAN 12-86
 - SET DEVELOPMENT 20-27
 - SET LIBRARY 5-39, 5-40
 - SLEEP 16-43
- program execution 5-10
 - canceling 21-3
 - conditional 5-12, 5-23, 5-24
 - OS() 11-29
 - coverage analysis 22-12
 - delaying 16-44
 - interrupting 16-37
 - optimizing 20-28
 - problems with 20-25
 - repeating 5-14
 - resuming 21-16, 21-23
 - retrying 21-16
 - stopping 5-34, 5-38, 21-3, 21-22
 - suspending 21-3, 21-22
 - for specified
 - duration 16-43
 - until key pressed 16-46
 - viewing 20-13

- program files 20-3
 - closing 5-40
 - creating 20-6
 - memory management 5-41
 - recompiling 20-27
 - removing from memory 5-41
 - search paths 5-12
- PROGRAM() 21-14
 - LINENO() and 21-8
- programming
 - Windows 19-1
- programs
 - accessing values 5-27
 - changing suspended 21-23
 - clearing from memory 5-7
 - compiling *See* compiling
 - coverage analysis 20-25
 - creating 20-28
 - current settings 5-42, 5-43
 - developing 20-25, 20-28
 - editing 20-6
 - flow, tracking 21-8
 - interrupting SLEEP 16-44, 16-45
 - multiple, identifiers 22-11
 - names, returning 21-14
 - recompiling 22-11
 - testing 20-26, 21-3
 - version control 22-7, 22-11
- Progress class 15-36
- progress indicators 15-36
- progress information 14-101
- prompts
 - currently selected 15-83
 - list boxes 15-82
 - selecting 15-141
- PROPER() 6-17
 - scan() and 10-35
- properties
 - accessing
 - by name 4-11
 - by value 4-10
 - assigning values 5-10
 - changing 20-19
 - creating member 5-5
 - defined 2-14
 - passing as parameters 5-27
 - preventing creation 4-3
 - redefining 5-34
 - specifying default
 - object 5-52
 - viewing 20-19
 - visual components, common
 - (table) 15-1
- property names 3-2

- PROTECT 5-5, 21-17
- protecting
 - data 20-27, 21-15
 - files 21-15
 - properties and methods 5-5
- prototypes (DLL functions) 19-12
- PROW() 18-13
 - _plineno and 18-32
 - SET PROW and 18-19
- _pspacing 18-36
- PUBLIC 5-33
- public variables 4-12
 - clearing 5-37
 - declaring 5-33
 - macro substitution and 4-13
- PushButton class 15-37
 - events (table) 15-37
 - properties (table) 15-37
- pushbuttons 15-167
 - See also* buttons
 - adding graphics 15-89, 15-91, 15-99
 - to nonselected 15-176
 - creating 15-37
 - default 15-88
 - disabling 15-89
- PUTFILE() 11-29
- puts() method 11-31
- PV() method 7-11

Q

- .QBE files
 - names, setting 15-180
 - opening 12-110
- queries
 - See also* SQL
 - assigning to database 14-63, 14-116
 - closing 14-34, 14-97
 - creating 14-19
 - deactivation
 - attempted 14-54
 - designing 20-27
 - eliminating duplicate
 - values 13-14
 - live, specifying 14-112
 - opening 14-34
 - attempted 14-57
 - parent, determining 5-30
 - referencing rowsets 14-114
 - rerunning 14-111
 - results 14-23
 - accessing 14-19
 - appending data 14-24

- browsing 14-24
- editable,
 - specifying 14-112
- editing 14-24
- filtering data 14-24
- locating data 14-24
- locking rows and
 - sets 14-25
- running 14-20, 14-68
- tables, accessing 15-56
- updating restrictions (local SQL) 13-7
- Query class 14-18
 - events (table) 14-19
 - methods (table) 14-19
 - properties (table) 14-18
- Query objects 14-2
 - creating 14-18
 - StoredProc object vs. 14-3
- question mark (?)
 - temporary files 11-21
 - wildcard character 6-16
- QUIT 5-34
 - CANCEL vs. 5-34
- quitting dBASE 5-34
- quitting loops 5-14, 5-18
- quotation mark symbol 4-16

R

- radians
 - arccosine 7-1
 - arcsine 7-2
 - arctangent 7-2, 7-3
 - converting from degrees 7-5
 - converting to degrees 7-14
 - cosine 7-4
 - returning 7-5
 - sine 7-19
 - tangent 7-20
- radio buttons
 - creating 15-38
- RadioButton class 15-38
 - events (table) 15-39
 - properties (table) 15-38
- random access memory (RAM) 21-9
- random numbers 7-13
- random records 12-51
- RANDOM() 7-12
- rangeMax property 15-152
 - rangeMin and 15-153
 - rangeRequired and 15-153
- rangeMin property 15-152
 - rangeMax and 15-152
 - rangeRequired and 15-153

- rangeRequired property 15-153
- ranges
 - key fields 12-101
 - spin boxes 15-152, 15-153
- RAT() 6-18
 - AT() and 6-5
- read() method 11-32
- reading from text files 11-33
- readln() method 11-33
- readModal() method 15-153
 - MDI property and 15-118
- read-only access 12-97, 12-102
- read-only fields 14-107, 15-97
- readOnly property 14-107
- RECALL 12-72
- RECCOUNT() 12-73
 - COUNT vs. 12-33
- RECNO() 12-73
- recompiling programs 20-27, 22-11
- reconnect() method 19-23
- record commands
 - APPEND 12-4
 - APPEND AUTOMEM 12-5
 - APPEND BLANK 12-4
 - APPEND FROM
 - ARRAY 12-8
 - BLANK 12-14
 - BOF() 12-15
 - BOOKMARK() 12-16
 - BROWSE 12-16
 - CLEAR AUTOMEM 12-21
 - COPY TO ARRAY 12-30
 - COUNT 12-32
 - DELETE 12-38
 - DELETED() 12-40
 - EDIT 12-43
 - EOF() 12-43
 - FLUSH 12-49
 - GO 12-51
 - LUPDATE() 12-65
 - RECCOUNT() 12-73
 - RECNO() 12-73
 - RECSIZE() 12-74
 - RELEASE AUTOMEM 12-76
 - REPLACE 12-77
 - REPLACE
 - AUTOMEM 12-79
 - REPLACE FROM
 - ARRAY 12-81
 - SET AUTOSAVE 12-92
 - SET DELETED 12-93
 - SKIP 12-111
 - STORE AUTOMEM 12-113
 - ZAP 12-124

- record counters 12-104
 - comparing 12-20
- record numbers
 - display, suppressing 12-42
 - returning 12-73
- record pointers
 - moving 12-51, 12-87, 12-92, 12-111
 - events and 15-138
 - linked tables 12-109
 - position, returning 12-15, 12-43
 - work areas 12-91
- records
 - adding 12-5, 13-13, 15-64
 - arrays and 12-8
 - event handling 15-126
 - restrictions 15-62
 - temporary 15-158
 - to rowsets 14-36, 14-37, 14-43
 - blank 12-4
 - return values 12-11, 12-114
 - change indicator 14-93
 - changing 15-113
 - browse objects 15-120
 - copying 12-7, 12-30
 - automatically 12-24
 - counting 11-14, 12-18, 12-73, 14-62, 14-113
 - deleting 12-40, 12-71, 12-124, 13-11, 15-55
 - all in table 14-67
 - attempted 14-54
 - confirming 12-108
 - controlling 12-93
 - event handling 14-98
 - from destination
 - tables 14-65
 - in rowsets 14-65
 - displaying 12-42, 12-60
 - editing 12-16, 12-17
 - filling with blanks 12-14
 - locking 12-64, 12-84, 14-80, 14-82
 - automatically 14-15
 - information,
 - getting 12-61, 20-4
 - retry attempts 14-81
 - retry intervals,
 - setting 14-82
 - retry messages 12-108
 - manipulating 12-111
 - moving through 12-52

- processing 12-101
- random 12-51
- saving temporary 15-158
- size, returning 12-74
- stepping through 12-86
- unlocking 12-105, 14-122
- updating 12-80
- RECSIZE() 12-74
 - LIST STRUCTURE and 12-74
 - RECCOUNT() and 12-73
- Rectangle class 15-39
 - properties (table) 15-40
- Rectangle objects 15-40
- recurring actions, setting 8-4
- recursive calls
 - ON ERROR and 21-12
 - ON NETERROR and 21-13
- REDEFINE 5-34
- redefining object
 - properties 5-34
- ref property 14-108
- reference, passing by 5-27
- referencing array elements 4-11, 10-40
- referencing components 15-123
- referencing forms 15-102
- referencing methods 4-11
- referencing objects
 - data modules 14-9, 14-108
 - in elements array 15-93
 - rowsets 14-114
 - with numeric values 15-110
- referencing reports 15-102
- referencing tables 4-17
- REFRESH 12-74
- refresh() method 14-108, 15-155
- refreshControls()
 - method 14-109
- refreshing data 14-108, 14-109
- refreshing screens 12-105, 15-85, 15-155
- refreshing work areas 12-74
- refreshRow() method 14-109
- REINDEX 12-75
 - SET UNIQUE and 12-110
- reindex() method 14-109
- RELATION() 12-75
- relational operators 4-7
- relationships (tables) 12-109
 - defining 12-106
 - restoring 12-76, 12-117
- RELEASE 5-35
 - CLEAR MEMORY vs. 5-7
- RELEASE AUTOMEM 12-76
- RELEASE DLL 19-24

- RELEASE OBJECT 5-37
- release() method 5-36
 - reconnect() and 19-23, 19-27
- releasing
 - locks 14-122
 - memory variables 5-7, 5-35, 5-37
 - objects from memory 5-36, 5-37
- remainders (division) 7-10
- remarks, adding to code 4-17
- removeAll() method 10-30
- removeKey() method 10-30
- RENAME 11-33
- RENAME TABLE 12-76
- rename() method 11-34
- renameTable() method 14-110
- renaming
 - files 11-33, 11-34
 - tables 12-77, 14-110
- render() method 17-37
- rendering reports 17-37
- repeating character strings 6-19, 6-20
- repeating program
 - execution 5-14
- REPLACE 12-77
 - REPLACE AUTOMEM vs. 12-79
- REPLACE AUTOMEM 12-79
- REPLACE BINARY 12-80
- REPLACE FROM ARRAY 12-81
- REPLACE MEMO 12-83
- REPLACE OLE 12-84
- replaceFromFile()
 - method 14-110
- replacing character strings 6-26
- replicate() method 6-20
- REPLICATE() 6-19
 - SPACE() vs. 6-24
- Report class 17-9
 - events (table) 17-10
 - methods (table) 17-10
 - properties (table) 17-9
- Report designer 20-9, 20-12
- report files 20-12
- report methods
 - agAverage() 17-13
 - agCount() 17-14
 - agMax() 17-15
 - agMin() 17-16
 - agStandardDeviation() 17-16
 - agSum() 17-17
 - agVariance() 17-18
 - isLastPage() 17-28
 - render() 17-37
- Report objects 17-10
- Report wizard 20-12
- reportGroup property 17-38
- reportPage property 17-38
- reports
 - closing 15-74, 15-129
 - designing 20-27
 - displaying data 17-10
 - displaying text in HTML 15-51
 - example report 17-2
 - groups, returning calculations 17-13 to 17-19
 - last page, determining 17-28
 - linking to data source 17-12
 - opening 15-139
 - output
 - objects 17-5
 - specifying file name 17-35
 - specifying medium 17-34
 - specifying printer options 17-36
 - overview 17-1 to 17-5
 - printing 18-7
 - referencing 15-102
 - rendering 17-4, 17-37
 - rendering data 17-10
 - in StreamFrames 17-20
 - sorting and groups 17-19
 - specifying page layout 17-8
 - summary-only, detail band and 17-6
 - titles, setting 17-42
- ReportViewer class 15-41
 - methods (table) 15-41
 - properties (table) 15-41
- requery() method 14-111
- requestLive property 14-112
- required fields 14-112
- required property 14-112
- reserved symbols 4-16
 - comments 4-17
- reserved words
 - naming restrictions and 5-22
 - SQL (list) 13-3
- resize() method 10-31
- RESOURCE() 19-25
- resources, information 1-1
- RESTORE 5-37
 - SAVE and 5-39
- RESTORE IMAGE 19-25
- restoring memory
 - variables 5-38
- restoring table
 - relationships 12-76, 12-117
- restricting data entry 15-62, 20-27
- RESUME 21-16
 - SUSPEND vs. 21-22
- resuming program
 - execution 21-16, 21-23
- RETRY 21-16
- RETURN 5-38
 - QUIT vs. 5-34
- return codes (DOS) 5-34
- return values 5-24, 5-38
 - absolute 7-1
 - angles 7-1, 7-2, 7-4, 7-19
 - tangents 7-2, 7-3, 7-20
 - averages 12-11, 12-18
 - blank records 12-11, 12-114
 - characters as dates 8-7
 - data types 5-50
 - decimal places 7-13
 - hexadecimal 9-7
 - infinity 7-20
 - integers 7-7, 7-18
 - expressing equality 7-3, 7-5
 - maximum 12-19
 - minimum 12-19
 - modulus 7-10
 - square roots 7-19
 - standard deviation 12-19
 - variance 12-19
- RGB color values 15-77
- right property 15-157
 - bottom and 15-67
- right() method 6-21
- RIGHT() 6-20
- rightTrim() method 6-21
- RLOCK() 12-84
 - FLOCK() vs. 12-48
 - LOCK() vs. 12-64
 - SET REPROCESS and 12-108
- _rmargin 18-37
 - _alignment and 18-20
 - _wrap and 18-39
- rollback() method 14-113
- ROLLBACK() 12-86
- rolling back transactions 12-86
- rotate property 17-39
- rotating text 17-39
- ROUND() function 7-13
 - compared (table) 7-7
- rounding 6-25, 7-13

- row buffer
 - addition attempted 14-56
 - saved event 14-102
 - saving 14-115
 - attempted 14-57
 - validation code 14-58
- row cursor
 - determining position 14-40, 14-41, 14-67
 - moving forward or backward 14-94
 - moving to first row 14-73
 - moving to last row 14-78
 - moving to specified row 14-75
- rowCount() property 14-113
- rowNo() property 14-114
- rows *See* records
- Rowset class 14-20
 - events (table) 14-21
 - methods (table) 14-22
 - properties (table) 14-21
- Rowset objects 14-20
- rowset property 14-114
- rowsets
 - abandoning 14-95
 - abandoning attempted 14-51
 - adding rows 14-36, 14-37, 14-43
 - appending data 14-24
 - browsing 14-24
 - controlling multiple
 - detail 14-89
 - copying 14-61
 - counting rows 14-62, 14-113
 - creating 14-20
 - current mode,
 - determining 14-119
 - current position
 - bookmarking 14-49
 - returning 14-49
 - defined 14-2
 - deleting rows 14-65
 - detail
 - constraining 14-91
 - controlling 14-24
 - link to master 14-89, 14-91
 - determining if editable 14-79
 - editing 14-24, 14-45
 - filtering 14-24, 14-37
 - forward navigation
 - only 14-122
 - getting 14-118
 - getting fields 14-69
 - getting row number 14-114

- index tag 14-76
- indexed searches 14-72
- locating data 14-24
- locking 14-25, 14-84
 - retry attempts 14-81
 - retry intervals,
 - setting 14-82
- master-detail link
 - calculated fields 14-90
 - canceling 14-89
- navigating 14-56, 14-99
- opened 14-100
- referencing 14-114
- refreshing data 14-108
- unlocking 14-122
- updating 14-37, 14-123
- RTOD() 7-14
 - ACOS() and 7-2
 - ASIN() and 7-2
 - ATAN() and 7-3
 - ATN2() and 7-3
- RTRIM() 6-21
 - LTRIM() vs. 6-17
- rules, creating 15-28
- RUN 11-35
- RUN() 11-35
 - DOS vs. 11-16
- run-time errors
 - IDAPI 21-4, 21-5
 - line numbers, returning 21-8
 - messages, customizing 21-18
 - multiuser
 - environments 21-13
 - server 21-19, 21-22
- running *See* executing

S

- sample data 12-51
- SAVE 5-38
 - RESTORE and 5-37
- save() method 14-115
- saveRecord() method 15-158
- saving
 - current row buffer 14-115
 - data 12-49, 14-93
 - automatically 12-92
 - files 11-20
 - memory variables 5-38
 - output 18-14
 - records 15-158
- scale property 14-115
- scaleFontName property 15-159
- ScaleFontSize property 15-159
- scaling fonts 15-159
- SCAN 12-86

- EOF() and 12-44
- scan() method 10-35
- scientific notation 7-5, 7-8
- scope 12-3
 - memory variables 5-25, 5-33
- scope resolution operator 4-11
- screens, refreshing 12-105, 15-85, 15-155
- scroll bars
 - creating 15-109, 15-159, 15-182
 - setting position 15-180
- scrollable lists 15-31
- ScrollBar class 15-42
 - events (table) 15-42
 - properties (table) 15-42
- scrollBar property 15-159
- scrolling forms 15-109, 15-159, 15-182
- SDI forms 15-23
- search operations
 - conditions 12-62
 - continuing 12-23
 - exact matches 12-88
 - failing 12-103
 - expressions, finding 12-64
 - files 11-37
 - checking existence 11-19
 - key values and 12-54, 12-90, 12-110
 - matches, finding 12-50
 - optimizing 12-65, 12-89
 - pattern matching 6-16
 - phonetic matches 6-8, 6-23
 - sequential 12-63, 12-88
 - substrings 6-5
- search order
 - (preprocessor) 22-10
- search path 5-12, 11-37
 - DLL files 19-14
 - preprocessor 22-10
- search-and-replace
 - operations 6-26, 6-27
 - preprocessor 22-12
- searching
 - See also* locating
 - case-sensitivity, turning off 6-13
 - data 14-38, 14-46
 - locate options 14-80
 - next match 14-79
 - files, existence of 11-19
 - keys in array elements 10-28
 - values in arrays 10-35
- searching and summarizing

- AVERAGE 12-11
- CALCULATE 12-18
- CONTINUE 12-23
- DESCENDING() 12-41
- FOUND() 12-50
- KEYMATCH() 12-59
- LOCATE 12-62
- LOOKUP() 12-64
- SEEK 12-88
- SEEK() 12-90
- SET NEAR 12-103
- SUM 12-114
- TOTAL 12-117
- secant 7-4
 - inverse 7-2
- seconds
 - returning 8-16
 - setting 8-27
- security 14-26, 14-35
 - access levels 14-33
 - ACCESS() 21-1
 - login attempts 14-85, 14-125
 - LOGOUT 21-8
 - PROTECT 21-15
 - SET ENCRYPTION 21-17
 - USER() 21-23
- SEEK 12-88
 - EOF() and 12-44
 - FOUND() and 12-50
 - INDEX and 12-56
 - LOCATE vs. 12-63
 - SET NEAR and 12-103
- seek() method 11-36
- SEEK() 12-90
 - EOF() and 12-44
 - FOUND() and 12-50
 - INDEX and 12-56
 - SEEK vs. 12-89
 - SET NEAR and 12-103
- SELECT 12-90
- SELECT statement (SQL) 13-13
 - to 13-17
- SELECT() 12-91
- selectAll property 15-160
- selected property 15-161
- selecting
 - colors 16-19
 - files 11-22
 - fonts 15-159, 16-20
 - magnitudes 15-44
 - menu commands 16-43
 - multiple 15-31
 - objects 15-94, 15-182
 - prompts 15-141
 - work areas 12-90, 12-91
- semicolons (;)
 - command separator 16-39
- separator property 16-35
- separators
 - command execution 16-39
 - date 8-25
 - changing 8-23
 - decimal digits 7-16
 - directory paths 11-38
 - menus 16-35
 - thousands 7-17
 - time 16-44
- sequential searches 12-63, 12-88
- server errors 21-19, 21-22
 - getting descriptions 14-10
- server property 19-26
- serverName property 15-162
- servers
 - connecting to 12-69, 15-90, 19-16, 19-23
 - disconnecting 19-27
- Session class 14-25
 - methods (table) 14-26
 - properties (table) 14-26
- session property 14-116
- sessions
 - closing 14-26
 - default 14-2
 - defined 14-2
 - getting assigned 14-116
 - number supported 14-26
 - opening 14-25
 - parent, determining 5-30
- SET 20-23
- SET... commands
 - changing interactively 20-23
 - current setting 5-42
 - information, getting 20-16
- SET ALTERNATE 18-13
- SET TALK and 20-32
- SET AUTOSAVE 12-92
- SET BELL 20-23, 20-24
- SET BLOCKSIZE 20-24
 - overriding 20-31
- SET IBLOCK and 20-29
- SET MBLOCK vs. 20-31
- SET CENTURY
 - YEAR() and 8-18, 8-31
- SET CENTURY command 8-22
- SET CONFIRM 16-35
- SET COVERAGE 20-25
 - #pragma vs. 22-12
- SET CUAENTER 16-36
- SET CURRENCY 7-15
- SET CURSOR 18-15
- SET DATABASE 12-92
 - BEGINTRANS() and 12-12
 - COMMIT() and 12-23, 12-86
 - DATABASE() and 12-37
 - DIR and 11-15
- SET DATE command 8-23
- SET DATE TO command 8-24
- SET DBTYPE 12-93
 - CREATE and 20-6
 - DIR and 11-14
- SET DECIMALS 7-15
 - ACOS() and 7-2
 - ASIN() and 7-2
 - ATAN() and 7-3
 - ATN2() and 7-3
 - AVERAGE and 12-11, 12-114
 - CALCULATE and 12-19
 - COS() and 7-4
 - DTOR() and 7-5
 - EXP() and 7-5
 - FLOOR() and 7-4, 7-6
 - FV() and 7-7
 - LOG() and 7-8
 - LOG10() and 7-8
 - PAYMENT() and 7-11
 - PI() and 7-11
 - PV() and 7-12
 - RANDOM() and 7-13
 - ROUND() and 7-13
 - RTOD() and 7-14
 - SET PRECISION vs. 7-17
 - SIGN() and 7-18
 - SIN() and 7-19
 - SQRT() and 7-20
 - TAN() and 7-20
- SET DELETED 12-93
 - CONVERT and 20-4
 - KEYMATCH() and 12-60
 - PACK vs. 12-71
 - RECALL and 12-72
- SET DESIGN 20-26
- SET DEVELOPMENT 20-27
 - DO and 5-11
- SET DEVICE
 - PCOL() and 18-11
 - PROW() and 18-13
- SET DIRECTORY 11-37
 - CD vs. 11-7
- SET EDITOR 20-28
- SET ENCRYPTION 21-17
- SET EPOCH command 8-24
- SET ERROR 21-18
- SET ESCAPE 16-37
 - WAIT and 16-46
- SET EXACT 12-94

- LOCATE and 12-63
- scan() and 10-35
- SEEK and 12-89
- SET KEY and 12-102
- SET EXCLUSIVE 12-94
- FLOCK() vs. 12-48
- SET FIELDS 12-95
- CLEAR FIELDS and 12-21
- COPY and 12-25
- COPY STRUCTURE and 12-28
- FLDLIST() and 12-46
- SET FILTER 12-98
- KEYMATCH() and 12-60
- SET KEY and 12-102
- SET FULLPATH 11-37
- DBF() and 12-38
- HOME() and 11-26, 11-43
- MDX() and 12-66, 12-69
- SET FUNCTION 16-38
- SET HEADINGS 12-99
- DISPLAY and 12-42
- SET IBLOCK 20-29
- SET BLOCKSIZE vs. 20-24
- SET INDEX 12-99
- SET EXCLUSIVE and 12-95
- SET KEY TO 12-101
- KEYMATCH() and 12-60
- SET LDCHECK 21-18
- SET LD_CONVERT 21-19
- SET LIBRARY 5-39
- CLEAR PROGRAM and 5-8
- SET LOCK 12-102
- SET MARGIN 18-16
- _ploffset and 18-34
- SET MARK 8-25
- SET DATE and 8-23
- SET MBLOCK 20-31
- SET BLOCKSIZE vs. 20-24
- SET MEMOWIDTH 12-103
- MLINE() and 12-68
- SET MESSAGE 16-40
- SET NEAR 12-103
- FOUND() and 12-51
- SEEK and 12-89
- SET RELATION and 12-107
- SET ODOMETER 12-104
- SET ORDER 12-104
- ORDER() and 12-70
- SET PATH 11-37
- CD vs. 11-7
- SET PCOL 18-17
- SET POINT 7-16
- SET PRECISION 7-16
- SET PRINTER 18-18
- CHOOSEPRINTER() and 18-6
- PCOL() and 18-11
- _pcolno and 18-26
- PRINTJOB and 18-11
- PROW() and 18-13
- SET PROCEDURE 5-40
- CLEAR PROGRAM and 5-8
- CLOSE PROCEDURE vs. 5-9
- FUNCTION and 5-22
- SET LIBRARY vs. 5-40
- SET PROW 18-19
- SET REFRESH 12-105
- SET RELATION 12-105
- CALCULATE and 12-19
- FLOCK() and 12-49
- FOUND() and 12-50
- RELATION() and 12-75
- RLOCK() and 12-85
- SET DELETED and 12-93
- SET SKIP and 12-109
- TARGET() and 12-117
- UNLOCK and 12-120
- SET REPROCESS 12-108
- FLOCK() and 12-48, 12-85
- SET SAFETY 12-108
- COPY BINARY and 12-26
- COPY MEMO and 12-27
- COPY STRUCTURE and 12-28
- copy() and 11-8
- create() and 11-10
- rename() and 11-34
- SAVE and 5-39
- SET ALTERNATE and 18-14
- TYPE and 11-41
- ZAP and 12-124
- SET SEPARATOR 7-17
- DIR/DIRECTORY and 7-18
- SET SKIP 12-109
- SET RELATION and 12-107
- SET SPACE 18-20
- ? command and 18-4
- SET TALK 20-32
- AVERAGE and 12-11, 12-114
- CALCULATE and 12-19
- SET SAFETY and 12-109
- SET TIME 8-25
- SET TYPEAHEAD 16-40
- SET UNIQUE 12-110
- INDEX and 12-56
- SET VIEW 12-110
- SET() 5-42
- SETTO() vs. 5-42
- SET...TO commands, current setting 5-43
- setDate() method 8-25
- setFocus() method 15-163
- setHours() method 8-26
- setMinutes() method 8-26
- setMonth() method 8-26
- setSeconds() method 8-27
- setTime() method 8-27
- setting DDE links 19-16, 19-23
- SETTO() 5-43
- setYear() method 8-28
- Shape class 15-43
- properties (table) 15-44
- Shape objects 15-44
- borders 15-147, 15-148
- specifying shape 15-164
- shapeStyle property 15-164
- share property 14-117
- shared data commands
- BEGINTRANS() 12-12
- CHANGE() 12-20
- COMMIT() 12-23
- CONVERT 20-4
- FLOCK() 12-48
- ID() 21-6
- LKSYS() 12-61
- LOCK() 12-64
- NETWORK() 21-10
- ON NETERROR 21-13
- RLOCK() 12-84
- ROLLBACK() 12-86
- SET EXCLUSIVE 12-94
- SET LOCK 12-102
- SET REFRESH 12-105
- SET REPROCESS 12-108
- UNLOCK 12-119
- shared mode 12-95
- shared resources 14-117
- SHELL() 16-41
- shift bits operators 9-2
- Shift-key combinations
- command execution 16-38
- shortCut property 16-43
- shortName() method 11-38
- showSpeedTip property 15-166
- SIGN() 7-18
- similar spellings, finding 6-23
- SIN() 7-19
- ASIN() and 7-2
- DTOR() and 7-5
- sine 7-19
- inverse 7-2
- reciprocal 7-19
- single-line input fields 15-18

- ul style="list-style-type: none;">
- single-quotation mark
 - symbol 4-16
- size property 10-36
- size() method 11-38
- sizeable property 15-166
- SKIP 12-111
 - EOF() and 12-44
 - SCAN and 12-87
 - SEEK and 12-88
- SLEEP 16-43
 - interrupting 16-44, 16-45
- Slider class 15-44
 - events (table) 15-45
 - methods (table) 15-45
 - properties (table) 15-45
- sliders 15-44
- SORT 12-111
 - INDEX vs. 12-57
 - sort() vs. 10-39
- sort order
 - arrays 10-38
 - default 12-112
 - indexes 12-53
- sort() method 10-37
- sorting
 - array elements 10-37
 - array rows 10-37
 - in reports 17-19
- sorting data 12-111, 21-19
 - combo boxes 15-167
 - list boxes 15-167
 - multiple fields 12-112
- sorting dates 8-11
- sound applications 15-91
- sound effects 12-26
- SOUNDEX() 6-23
 - DIFFERENCE() and 6-8
- Source editor, activating 20-6
- source property 14-118
- space characters, returning 6-24
- space() method 6-25
- SPACE() 6-24
 - REPLICATE() vs. 6-19
- spaces
 - leading, deleting 6-15, 6-17
 - trailing, deleting 6-21, 6-30
- speakers 12-26, 20-24
- SpeedBar buttons 15-167
- speedBar property 15-167
- speedTip property 15-168
- spin boxes
 - ranges, setting 15-152, 15-153
 - scroll bars vs. 15-45
 - values, changing 15-168, 15-169
- SpinBox class 15-46
 - events (table) 15-47
 - methods (table) 15-47
 - properties (table) 15-46
- spinOnly property 15-168
- SQL (local) 13-1 to 13-17
 - aggregate functions 13-5
 - column naming
 - conventions 13-2
 - data definition
 - statements 13-4
 - data manipulation
 - statements 13-5
 - date functions 13-7
 - defined 13-1
 - live joins, restrictions 13-7
 - live queries, restrictions 13-7
 - string functions 13-6
 - table naming
 - conventions 13-2
 - updatable queries,
 - constraints 13-7, 13-8
- SQL clauses
 - ADD 13-8
 - DISTINCT 13-14
 - DROP 13-8
 - FROM 13-14
 - GROUP BY 13-15
 - HAVING 13-15
 - ORDER BY 13-15
 - SELECT 13-13
 - SET 13-17
 - UNION 13-16
 - VALUES 13-13
 - WHERE 14-124
 - with DELETE 13-11
 - with SELECT 13-14
 - with UPDATE 13-17
- SQL databases
 - linking tables 12-106
 - statements, executing 21-20
- SQL designer 20-11
- SQL fields, scale 14-115
- SQL functions
 - AVG() 13-5
 - COUNT() 13-6
 - EXTRACT() 13-7
 - LOWER() 13-6
 - MAX() 13-6
 - MIN() 13-6
 - SUBSTRING() 13-6
 - SUM() 13-5
 - TRIM() 13-6
 - UPPER() 13-6
- SQL operators (table) 13-3
- sql property 14-118
- SQL reserved words (list) 13-3
- SQL servers, increasing
 - performance 14-122
- SQL statements 14-2
 - See also* queries; rowsets;
 - SQL clauses
 - ALTER TABLE 13-8
 - CREATE INDEX 13-9
 - CREATE TABLE 13-9
 - DELETE 13-11
 - DROP INDEX 13-12
 - DROP TABLE 13-12
 - executing 14-20, 14-68
 - external files 14-119
 - generated rowset 14-118
 - INSERT 13-13
 - preparing 14-105
 - SELECT 13-13
 - UPDATE 13-17
- SQLERROR() 21-19
- SQLEXEC() 21-20
- SqlField class 14-27
 - properties (table) 14-27
- SqlField objects 14-27
 - precision, setting 14-104
- SQLMESSAGE() 21-22
- SQRT() 7-19
- square root, returning 7-19
- stand-alone applications 15-23
- standalone applications
 - using SHELL 16-42
- standard deviation,
 - returning 12-19
 - in reports 17-16
- Standard tables
 - constraining updates 14-60
 - defined 14-1
 - local SQL and 13-1
 - queries and 14-3
 - rebuilding indexes 14-109
- startPage property 17-39
- startSelection 15-168
- state property 14-119
- statement, defined 2-10
- statements 4-17, 5-22
 - changing at runtime 4-13
 - comments and 4-17
 - executing conditionally 5-23
 - skipping 5-26
- STATIC 5-43
- static variables 5-44
- statistical operations 12-18
- status bars

- displaying messages 15-169, 16-40
- messages 20-32
- record counter
 - information 12-104
- statusMessage property 15-169
- step property 15-169
- stepping through records 12-86
- stopping program
 - execution 5-34, 5-38, 21-3, 21-22
- STORE 5-45
 - CLEAR AUTOMEM and 12-21
- STORE AUTOMEM 12-113
 - RELEASE AUTOMEM and 12-76
- stored procedures 14-3
 - calling 14-27
 - closing 14-97
 - creating parameters 14-15
 - getting parameter type 14-121
 - preparing 14-105
 - rerunning 14-111
 - running 14-68
 - specifying 14-106
 - values 14-126
- StoredProc class 14-27
 - events (table) 14-28
 - methods (table) 14-28
 - properties (table) 14-27
- StoredProc objects 14-3, 14-28
- storing graphics
 - binary fields 12-26
 - memo fields 12-27
- storing text 12-26
- STR() 6-25
- StreamFrame class 17-10
 - properties (table) 17-11
- StreamFrame objects
 - creating multiple 17-12
 - defined 17-12
 - rendering in (bands) 17-20
- streaming output, writing to files 18-18
- StreamSource class 17-12
 - properties (table) 17-12
- StreamSource objects, multiple assignments 17-13
- streamSource property 17-40
- strikeout attributes 15-101
- String class 6-1
- string comparisons
 - expressing equality 12-94
- pattern matching 6-16
- phonetic matching 6-8, 6-23
- string conversions
 - characters to dates 8-7, 20-22
 - dates to characters 8-9, 8-10, 8-11, 8-19, 20-22
 - lowercase to
 - uppercase 6-29, 6-31, 12-55
 - first letter 6-17, 6-29
 - numbers to strings 6-25
 - OEM characters to
 - ANSI 21-2
 - uppercase to
 - lowercase 6-17, 6-29
- string data commands
 - ANSI() 21-1, 21-11
 - AT() 6-5
 - CENTER() 6-6
 - DIFFERENCE() 6-8
 - ISALPHA() 6-11
 - ISLOWER() 6-11
 - ISUPPER() 6-12
 - LEFT() 6-14
 - LEN() 6-15
 - LIKE() 6-16
 - LOWER() 6-17
 - LTRIM() 6-17
 - PROPER() 6-17
 - RAT() 6-18
 - REPLICATE() 6-19
 - RIGHT() 6-20
 - RTRIM() 6-21
 - SOUNDEX() 6-23
 - SPACE() 6-24
 - STUFF() 6-26
 - SUBSTR() 6-27
 - TRIM() 6-21, 6-30
 - UPPER() 6-31
- string data methods
 - indexOf() 6-10
 - isAlpha() 6-11
 - isLower() 6-12
 - isUpper() 6-13
 - lastIndexOf() 6-13
 - left() 6-14
 - leftTrim() 6-15
 - replicate() 6-20
 - right() 6-21
 - rightTrim() 6-21
 - space() 6-25
 - stuff() 6-27
 - substring() 6-28
 - toLowerCase() 6-29
 - toProperCase() 6-29
 - toUpperCase() 6-29
- string delimiters 4-16
- string manipulation functions (local SQL) 13-6
- string operators 4-4, 4-5
- strings
 - alphabetic characters, testing for 6-11
 - characters
 - specified position 6-7
 - conversions *See* string conversions
 - creating substrings 13-6
 - defined 2-2
 - duplicating 6-19, 6-20
 - equality comparisons 4-8
 - finding substrings 6-5, 6-10, 6-13, 6-18
 - leading spaces,
 - deleting 6-15, 6-17
 - literal 4-16
 - padding 6-25
 - replacing specific characters 6-26
 - returning 5-50
 - case (lower) 6-11, 6-12
 - case (upper) 6-12, 6-13
 - centered 6-6
 - dates 8-6, 8-7
 - current 8-8
 - DLLs 19-25
 - number of
 - characters 6-14, 6-15, 6-20
 - repeated strings 6-19, 6-20
 - spaces 6-24
 - substrings 6-5, 6-10, 6-27, 6-28
 - searching and replacing
 - in 6-26, 6-27
 - trailing spaces, deleting 6-21, 6-30
 - trimming 13-6
- structure-extended tables 12-29
 - creating 12-37
- stuff() method 6-27
- STUFF() 6-26
- style property 15-169
- subclassing, dynamic 2-14, 5-3
- submitting forms 15-88
- subscript() method 10-40
 - element() vs. 10-41
- subscripts *See* array elements, subscripts
- SUBSTR() 6-27

- SUBSTRING() function (SQL) 13-6
- substring() method 6-28
- substrings
 - creating 13-6
 - finding 6-5, 6-10, 6-13, 6-18
 - replacing characters 6-26
 - returning 6-27, 6-28
- subtraction operator 4-6
- SUM 12-114
 - SET HEADINGS and 12-99
 - TOTAL vs. 12-118
- SUM() function (SQL) 13-5
- summaries, calculating 17-7
- summary information
 - displaying (in reports) 17-25, 17-27
- summary-only report
 - detail band and 17-6
- super keyword 4-11
- superclasses 5-5
- supported file types 12-7
- supported table types 14-1
- suppressIfBlank property 17-41
- suppressIfDuplicate
 - property 17-41
- SUSPEND
 - PROGRAM() and 21-14
- suspending program
 - execution 21-3, 21-22
 - for specified duration 16-43
- switch blocks
 - designating codeblocks 5-5, 5-26
 - executing default 5-26
- symbols
 - comments 4-17
 - non-operational 4-16
 - preprocessor directive 4-19
 - string literals 4-16
 - used in syntax (table) 3-1
- syntax
 - annotated example 3-2
 - conventions, described 3-1
 - errors 20-3
 - symbols used (table) 3-1
- sysMenu property 15-170
- system
 - bell, setting 20-23
 - clock 8-8
 - setting 8-24, 8-25
 - time elapsed 8-11
 - date
 - changing 8-23
 - returning 8-8

- system memory variables 18-11
 - _alignment 18-20
 - _dbwinhome 11-43
 - _indent 18-21
 - _lmargin 18-22
 - _padvance 18-23
 - _pageno 18-24
 - _pbpage 18-25
 - _pcolno 18-26
 - _pcopies 18-27
 - _pdriver 18-27
 - _pject 18-28
 - _pepage 18-29
 - _pform 18-30
 - _plength 18-31
 - _plineno 18-32
 - _ploffset 18-33
 - _porientation 18-34
 - _ppitch 18-35
 - _pquality 18-36
 - _pspacing 18-36
 - _rmargin 18-37
 - _tabs 18-38
 - _wrap 18-39
- System menu 15-170
- system utilities and information
 - CD 11-6
 - DIR/DIRECTORY 11-14
 - DISKSPACE() 11-15
 - GETENV() 11-22
 - MD 11-26
 - MEMORY() 21-9

T

- tab boxes
 - current prompt 15-83
 - maintaining size and location 15-61
- Tab key 16-36
- tabbing order 15-63
 - SpeedBar buttons 15-168
- TabBox class 15-48
 - events (table) 15-48
 - properties (table) 15-48
- table basics commands
 - ALIAS() 12-3
 - APPEND FROM 12-7
 - CLOSE TABLES 12-22
 - COPY 12-24
 - COPY STRUCTURE 12-27
 - COPY TABLE 12-29
 - COPY TO...STRUCTURE
 - EXTENDED 12-28
 - CREATE 20-5
 - CREATE...FROM 12-35
 - CREATE...STRUCTURE
 - EXTENDED 12-36
 - DATABASE() 12-37
 - DBF() 12-38
 - DELETE TABLE 12-39
 - DISPLAY
 - STRUCTURE 20-17
 - ISTABLE() 12-58
 - MODIFY STRUCTURE 20-21
 - OPEN DATABASE 12-69
 - REFRESH 12-74
 - RENAME TABLE 12-76
 - SELECT 12-90
 - SELECT() 12-91
 - SET DATABASE 12-92
 - SET DBTYPE 12-93
 - SQLEXEC() 21-20
 - USE 12-121
 - WORKAREA() 12-124
- Table designer 20-6, 20-22
- table maintenance
 - methods 14-6
- table names
 - changing 12-76
 - returning 12-38, 12-117
 - setting 15-180
 - SQL naming
 - conventions 13-2
- table structures
 - changing 13-8, 20-6, 20-21
 - copying 12-27, 12-28
 - designing 12-36
 - displaying 20-17
 - storing 10-19
- Table wizard 20-6
- tableExists() method 14-120
- tables
 - accessing 15-56
 - adding fields 20-22
 - adding records 12-4, 12-6, 15-64, 15-158
 - event handling 15-126
 - restricting 15-62
 - aliases 15-56
 - changing 20-5
 - checking for 14-120
 - closing 20-2, 21-9
 - work areas 12-22
 - controlling access 21-15
 - copying 12-29
 - one to another 14-61
 - creating 12-27, 12-28, 12-35, 12-112, 13-9
 - temporary 12-117
 - default type 12-93

- deleting 12-39, 13-12, 14-66
- deleting indexes (local SQL) 13-12
- designing 20-27
- determining existing 12-58
- directory listings 11-14
- duplicating 14-61
- emptying 14-67
- encrypting 21-17
- exporting with COPY 12-26
- information, getting 10-19, 20-16
- key violation 14-78
- linking 12-105
- locking 12-102
 - information, getting 12-61, 20-4
 - retry messages 12-108
- moving through 12-51
 - linked 12-109
- opening 12-91, 12-121
 - file-sharing modes 12-94
- packing 14-103
- problem, specifying 14-106
- referencing 4-17
- relations 12-109
 - defining 12-106
 - restoring 12-76, 12-117
- renaming 14-110
- required fields 14-112
- security levels 14-33, 14-35
- size, returning 12-73
- structure-extended 12-29, 12-37
- temporary 12-117
- types supported 14-1
- undoing changes 14-58
- unlocking 12-105
- updating data (local SQL) 13-17
- _tabs 18-38
- tabs 15-49
- tabStop property 15-170
- TAG() 12-115
- TAGCOUNT() 12-116
- TAGNO() 12-116
- TAN() 7-20
 - DTOR() and 7-5
- tangent 7-20
 - inverse 7-2, 7-3
 - reciprocal 7-20
- target rowset/table, specifying 14-66
- TARGET() 12-117
- TEDIT setting 20-6
- template characters 7-18, 15-103, 15-149
- temporary files 11-21, 12-123
 - SORT and 12-112
- temporary tables 12-117
- terminate() method 19-27
 - initiate() and 19-16
 - reconnect() and 19-23
- terminating program
 - execution 5-34, 5-38
- testing conditions 5-20, 5-24
 - multiple 5-24
- testing memory variables 5-20
- testing programs 20-26, 21-3
- text
 - See also* Text objects
 - adjusting character spacing 17-42
 - aligning 15-58
 - horizontally 15-57
 - vertically 15-58
 - centering 6-6
 - copying 15-81, 16-17
 - cutting 16-18
 - deleting 15-84, 16-18
 - files, reading from 11-33
 - formatting 15-103, 15-148
 - size 15-159
 - with spaces 6-24
 - fully justified, setting word spacing 17-42
 - getting length 15-103
 - multi-line input fields 15-16
 - offset margins 18-16
 - overstriking 18-4
 - paging through 15-109, 15-159, 15-182
 - pasting 15-146, 16-18
 - rotating 17-39
 - SET DESIGN and 20-27
 - setting non-editable 15-171
 - single-line input fields 15-18
 - storing 12-26
 - vertically justified, setting spacing 17-43
 - wordwrapping 15-184
- text attributes
 - colors 15-75
 - setting font name 15-100
 - setting typestyle 15-100
 - size 15-101, 15-159
 - strikeout 15-101
 - underlined 15-101
- Text class 15-49
 - methods (table) 15-51
- properties (table) 15-49
- text editors, alternate 20-6
 - specifying 20-28
- text files
 - copying to memo fields 12-10, 12-83
 - creating 20-28
 - displaying 11-40
 - editing 20-7
 - printing 11-40
 - writing to
 - alternate 18-13
 - environment information 20-32
 - files list 11-16, 11-26
 - from memo fields 12-27
 - streaming output 18-18
- Text objects 15-49
 - See also* text
- text property 15-171
 - downBitmap and 15-91
- this (using) 2-15
- thousands separator 7-17
- THROW 5-46
- time 8-24
 - default settings 8-23
 - elapsed 8-11
 - intervals, setting 8-4
 - resetting 8-25
 - returning
 - in milliseconds 8-16, 8-21
 - using GMT 8-30
 - separators 16-44
 - setting 8-27
- time formats 8-25
 - specifying 8-23
- time stamp, returning 11-39
- time stamping 14-96
- time zone, returning offset 8-17
- TIME()
 - ELAPSED() and 8-12
- time() method 11-39
- timeOut property 19-27
- Timer class 8-4
 - events (table) 8-5
 - properties (table) 8-5
- Timer object
 - activating 8-12
 - idle time, specifying 8-18
 - interval elapsed, handling event 8-20
- title property 17-42
- toggle property 15-173
- toGMTString() method 8-29
- toLocaleString() method 8-29

- toLowerCase() method 6-29
- ToolBar class 16-9
 - events (table) 16-10
 - methods (table) 16-10
 - properties (table) 16-10
- toolbars 16-9
 - creating buttons 16-11
- ToolBar class 16-11
- toolButtons 16-11
- top property 15-174
 - bottom and 15-67
- topic property 19-28
- topMost property 15-174
- toProperCase() method 6-29
- toString() method 8-30
- TOTAL 12-117
 - SUM vs. 12-115
- totals 12-117
 - returning in reports 17-17
- toUpperCase() method 6-29
- tracking property 17-42
- trackJustifyThreshold
 - property 17-42
- TrackRight property 16-45
- trailing spaces, deleting 6-21, 6-30
- trailing zeros, deleting 7-13
- transactions 12-12, 14-6
 - canceling 14-113
 - committing 12-23
 - DDE applications 19-27
 - defined 14-47
 - isolation levels 14-77
 - logging 14-5, 14-47
 - caching updates vs. 14-47
 - posting 14-60
 - rolling back 12-86
- TRANSFORM() 6-30
 - SET CURRENCY and 7-15
- transparent property 15-175
- TreelItem class 15-51
 - methods (table) 15-52
 - properties (table) 15-52
- trees 15-53
 - adding items 15-51
- TreeView class 15-53
 - events (table) 15-54
 - methods (table) 15-54
 - properties (table) 15-53
- trigonometric functions
 - ACOS() 7-1
 - ASIN() 7-2
 - ATAN() 7-2
 - ATN2() 7-3
 - COS() 7-4

- DTOR() 7-4
- RTOD() 7-14
- SIN() 7-19
- TAN() 7-20
- TRIM() 6-21, 6-30
 - LTRIM() vs. 6-17
- TRIM() function (SQL) 13-6
- true/false values 15-7
- truncating numeric data 7-7
- TRY 5-46
 - CATCH and 5-5
 - FINALLY and 5-19
 - RETURN and 5-38
- TRY...ENDTRY 5-19
- twips measurement,
 - defined 17-1
- TYPE 11-40
- type conversion, explicit 2-8
- type conversions
 - ASC() 6-4
 - CTOD() 8-7
 - DTOC() 8-10
 - DTOS() 8-11
 - HTOI() 9-7
 - ITOH() 9-7
 - STR() 6-25
- type property
 - Field 14-121
 - Parameter 14-121
- TYPE() 5-50
- typeahead buffer
 - clearing 16-15
 - information, getting 16-21, 16-28
 - inserting keystrokes 16-24
 - size, setting 16-40
- typographical conventions 1-2

U

- UDFs
 - See also* functions
 - debugging 21-14
- unadvise() method 19-28
- unary operators 2-5, 4-4, 4-6
 - negation 4-5
- #undef directive 22-12
- undefining identifiers 22-4, 22-12
- underline attribute 15-101
- undo() method 15-175
- undoing table changes 14-58
- unidirectional property 14-122
- UNIQUE() 12-119
- UNLOCK 12-119
- unlock() method 14-122
- unlocking files 12-105
- unlocking records 14-122
- upBitmap property 15-176
 - downBitmap and 15-91
- UPDATE 12-120
- UPDATE statement (SQL) 13-17
 - SET clause and 13-17
 - WHERE clause and 13-17
- update() method 14-123
- UpdateSet class 14-30
 - index tags 14-77
 - methods (table) 14-30
 - properties (table) 14-30
- UpdateSet operations
 - specifying destination 14-66
 - specifying source 14-118
- updateWhere property 14-124
- updating data 12-65, 12-120, 14-37, 15-84
 - across databases 14-30
 - automatically 14-95
 - cached 14-39
 - caching locally 14-50
 - constraints 14-60
 - local SQL and 13-17
 - multiuser
 - environments 12-48, 12-85
 - problems with 14-106
 - rowset to rowset 14-123
 - table to table 14-30
- updating data buffers 12-74
- updating indexes 20-23
 - automatically 12-56
- UPPER() 6-31
 - AT() and 6-5, 6-18
 - INDEX and 12-55
 - LIKE() and 6-16
 - scan() and 10-35
- UPPER() function (SQL) 13-6
- uppercase letters 12-55
 - converting to 6-29, 6-31
 - converting to
 - lowercase 6-17, 6-29
 - initial capitals 6-17, 6-29
 - sorting data 12-112
 - testing for 6-12, 6-13
- USA date format 8-23
- USE 12-121
 - SET INDEX and 12-100
- USE...AUTOMEM
 - APPEND AUTOMEM
 - and 12-6
 - RELEASE AUTOMEM
 - and 12-76

- STORE AUTOMEM
 - vs. 12-113
- USE...EXCLUSIVE
 - CONVERT and 20-4
 - FLOCK() vs. 12-48
 - NETWORK() and 21-10
 - SET EXCLUSIVE and 12-95
- user names 14-86
 - getting 21-6
- user property 14-124
- USER() 21-23
- user() method 14-125
- user-defined memory
 - variables 5-7
- user-defined types,
 - binary 12-26
- useTablePopup property 15-176
- UTC() 8-30

V

- VAL() 6-31
- valid property 15-177
 - onLostFocus vs. 15-136
 - validErrorMsg and 15-178
 - validRequired and 15-178
- validating data 15-178
- VALIDDRIVE() 11-41
 - CD and 11-7
- validErrorMsg property 15-178
- validRequired property 15-178
- value property 15-179
 - Field 14-125
 - Parameter 14-126
 - reading 14-42, 14-99
- values 14-14, 15-84
 - absolute 7-1
 - accessing program 5-27
 - arrays 12-11, 12-19, 12-114
 - assigning 4-11
 - to properties 5-10
 - attempting to change 14-53
 - automatic updates 14-95
 - averaging 12-11, 12-18
 - blank 12-20
 - calculated fields and 14-14
 - calculating aggregate 17-7
 - change indicator 14-93
 - changed 14-97
 - changing appearance 14-42
 - component's current 15-179
 - counting specified 13-6
 - decimal 9-7
 - converting to hexadecimal 9-7

- keystrokes,
 - returning 16-21, 16-28
- default 14-64
 - filling in 14-96
- determining 14-125
- duplicate
 - checking 12-59
 - eliminating 13-14
 - keys 12-109
- finding 12-90
- global 5-33
- group
 - count of items 17-14
 - highest value 17-15
 - lowest value 17-16
 - mean average 17-13
 - standard deviation 17-16
 - total 17-17
 - variance 17-18
- linking Field object (bitmap)
 - to form 15-87
- lookup 14-86, 14-87
- maximum (fields) 14-92
- memory variables 12-11,
 - 12-19, 12-114
- minimum (fields) 14-92
- monetary 7-15
- net present 12-19
- passing by 5-27
- returning 5-38, 5-50
 - logical expressions and 5-24
 - stored procedures 14-28
- saving original 14-58, 14-98
- specifying 22-3
- spin boxes 15-153, 15-169
 - changing 15-168
 - setting 15-152
- true/false 15-7
- updating 15-84
- variableHeight property 17-42
- variables
 - See also* memory variables
 - assigning 2-7
 - assigning codeblocks to 4-18
 - declaring 4-12, 5-31, 5-33
 - defined 2-7
 - macro substitution and 4-13
 - preventing creation 4-3
 - private 5-27, 5-31
 - restrictions 4-12, 4-16
 - specifying default object 5-52
- variance 12-19
 - returning in reports 17-18

- __vdb__ 22-13
 - #if and 22-13
- VDB.EXE
 - path, returning 11-25
- VDB.INI, changing
 - settings 20-23
- version control, programs 22-7,
 - 22-11
- version control programs 22-11
- version numbers 22-13
 - returning 11-28
 - Visual dBASE,
 - returning 22-13
- VERSION() 21-24
- vertical property 15-180
- vertical scroll bars 15-180
- verticalJustifyLimit
 - property 17-43
- view property 15-180
- viewing, program
 - execution 20-13
- views 15-89
 - returning name 15-180
- visible property 15-181
- volume, measuring 7-11

W

- WAIT 16-46
 - SLEEP vs. 16-45
- warning beeps 20-23
- when property 15-182
- whole numbers 7-7, 7-14
 - thousands separator 7-17
- width property 15-183
- wildcard characters
 - pattern matching 6-16
 - temporary files 11-21
- Window menu, adding to
 - forms 16-47
- WindowMenu property 16-47
- Windows Multiple Document
 - Interface standard 15-117
- Windows programming 19-1
- Windows programming
 - commands
 - BITLSHIFT() 9-2
 - BITRSHIFT() 9-4, 9-6
 - BITSET() 9-5
 - EXTERN 19-12
 - HELP 20-19
 - LOAD DLL 19-16
 - RELEASE DLL 19-24
 - RESOURCE() 19-25
- windows, displaying 16-44
- windowState property 15-184

WITH 5-52
Word and OLE automation 19-9
wordwrapping text 15-184
work areas
 active indexes 12-116
 aliases 12-2
 returning 12-3, 12-91
 closing files 12-22
 closing tables 12-22
 current, returning 12-124
 data buffers 12-74
 designating field names 4-12
 opening files 12-100
 opening tables 12-122
 record pointers 12-91
 referencing names 4-16
 refreshing 12-74
 returning tables 12-38
 search operations 12-50

 selecting 12-90, 12-91
WORKAREA() 12-124
_wrap 18-39
 _alignment and 18-20
 _indent and 18-21
 _tabs and 18-38
wrap property 15-184
write() method 11-41
writeln() method 11-42

X

Xbase commands and
 functions 12-1
XOR bitwise operator 9-5

Y

YEAR() 8-28, 8-31

years
 returning 8-18, 8-31
 setting 8-28

Z

ZAP 12-124
 PACK vs. 12-72
 RECALL and 12-72
 SET SAFETY and 12-108
zero values
 blank vs. 12-14
 equality comparisons 7-6
 finding 7-18
 random numbers 7-13
 trailing, deleting 7-13