Paradox for Windows

Version 1.0

ObjectPAL™ Reference

Borland International, Inc. 1800 Green Hills Road
P.O. Box 660001, Scotts Valley, CA 95067-0001, USA

Copyright ©1992 by Borland International, Inc. Portions copyright 1985 by Borland International, Inc. All rights reserved. Borland and Paradox are trademarks of Borland International. Microsoft and MS are trademarks of Microsoft Corporation. Windows, as used in this manual, refers to Microsoft's implementation of a windows system.

10 9 8 7 6 5 4 3 2 1 R1

CONTENTS

Chapter 1
Introduction 1

What's in this manual 1
ObjectPAL level 2
Prototype notation 3

Chapter 2
Built-in methods 5

About built-in methods 6
Descriptions of the built-in methods . . 6
Controlling the default behavior 16
Built-in object variables 20
Advanced topic: Sample flow of method calls 22

Chapter 3
Basic language elements 27

- = ... 27
- const 29
- disableDefault 29
- doDefault 30
- enableDefault 31
- for 31
- forEach 32
- if 33
- iif 34
- loop 35
- method 35
- passEvent 36
- proc 37
- quitLoop 38
- return 39
- scan 40
- switch 41
- try 43
- type 44
- uses 45
- var 54
- while 54

Chapter 4
Object type reference 57

Syntax notation 57
ActionEvent 60
- actionClass 60
- id 61
- setId 62

AnyType 64
- blank 65
- dataType 65
- isAssigned 66
- isBlank 67
- isFixedType 68
- unAssign 68
- view 69

Application 71
Array 72
- addLast 72
- append 74
- contains 74
- countOf 75
- empty 76
- exchange 77
- fill 77
- grow 78
- indexOf 79
- insert 80
- insertAfter 80
- insertBefore 81
- insertFirst 82
- isResizeable 82
- remove 83
- removeAllItems 84
- removeItem 85

- replaceItem . . . 85
- setSize . . . 86
- size . . . 87
- view . . . 87

Binary . . . 89
- readFromFile . . . 89
- size . . . 90
- writeToFile . . . 91

Currency . . . 92
- currency . . . 92

Database . . . 94
- close . . . 95
- delete . . . 96
- executeQBE . . . 97
- executeQBEFile . . . 98
- executeQBEString . . . 99
- isAssigned . . . 101
- isTable . . . 102
- open . . . 103
- writeQBE . . . 104

Date . . . 106
- date . . . 107
- dateVal . . . 107
- today . . . 108

DateTime . . . 109
- dateTime . . . 110
- day . . . 110
- daysInMonth . . . 111
- dow . . . 111
- dowOrd . . . 112
- doy . . . 113
- hour . . . 114
- isLeapYear . . . 114
- milliSec . . . 115
- minute . . . 116
- month . . . 116
- moy . . . 117
- second . . . 118
- year . . . 118

DDE . . . 120
- close . . . 120
- execute . . . 121
- open . . . 121
- setItem . . . 122

DynArray . . . 124
- contains . . . 124
- empty . . . 126
- getKeys . . . 127
- removeItem . . . 128
- size . . . 129
- view . . . 129

ErrorEvent . . . 131
- reason . . . 131
- setReason . . . 132

Event . . . 133
- errorCode . . . 133
- getTarget . . . 134
- isFirstTime . . . 135
- isPreFilter . . . 136
- isTargetSelf . . . 137
- reason . . . 137
- setErrorCode . . . 139
- setReason . . . 139

FileSystem . . . 141
- accessRights . . . 141
- copy . . . 142
- delete . . . 143
- deleteDir . . . 144
- drives . . . 145
- enumFileList . . . 146
- existDrive . . . 147
- findFirst . . . 148
- findNext . . . 149
- freeDiskSpace . . . 150
- fullName . . . 151
- getDir . . . 152
- getDrive . . . 153
- getFileAccessRights . . . 154
- getValidFileExtensions . . . 154
- isDir . . . 155
- isFile . . . 155
- isFixed . . . 156
- isRemote . . . 157
- isRemovable . . . 158
- makeDir . . . 158
- name . . . 159
- privDir . . . 160
- rename . . . 160
- setDir . . . 161
- setDrive . . . 161
- setFileAccessRights . . . 162
- size . . . 163
- splitFullFileName . . . 164

startUpDir 165
time 166
totalDiskSpace 166
windowsDir 167
windowsSystemDir 167
workingDir 168

Form 169
action 170
attach 171
bringToTop 172
close 173
create 173
delayScreenUpdates 174
deliver 176
design 176
disableBreakMessage 177
dmAddTable 177
dmGet 178
dmHasTable 179
dmPut 180
dmRemoveTable 180
enumSource 181
enumSourceToFile 182
enumTableLinks 183
enumUIObjectNames 184
enumUIObjectProperties 185
formCaller 186
formReturn 186
getPosition 188
getTitle 188
hide 189
hideSpeedBar 190
isMaximized 190
isMinimized 191
isSpeedBarShowing 191
isVisible 192
keyChar 192
keyPhysical 193
load 195
maximize 196
menuAction 197
methodDelete 197
methodGet 198
methodSet 199
minimize 199
mouseDouble 200
mouseDown 201
mouseEnter 202
mouseExit 203
mouseMove 204
mouseRightDouble 205
mouseRightDown 206
mouseRightUp 207
mouseUp 208
moveTo 209
moveToPage 210
open 210
openAsDialog 212
postAction 213
run 214
save 215
setPosition 215
setTitle 216
show 216
showSpeedBar 217
wait 217
windowClientHandle 218
windowHandle 218

Graphic 220
readFromClipboard 220
readFromFile 221
writeToClipboard 222
writeToFile 222

KeyEvent 224
char 224
charAnsiCode 225
isAltKeyDown 225
isControlKeyDown 226
isFromUI 227
isShiftKeyDown 227
setAltKeyDown 227
setChar 228
setControlKeyDown 229
setShiftKeyDown 230
setVChar 231
setVCharCode 231
vChar 232
vCharCode 233

Library 235
close 235
enumSource 236
enumSourceToFile 237
execMethod 238
open 239

Logical 241
logical 241
LongInt 243
bitAND 243
bitIsSet 244
bitOR 246
bitXOR 246
longInt 247
Memo 249
memo 249
readFromFile 250
writeToFile 251
Menu 252
addArray 253
addBreak 253
addPopUp 254
addStaticText 255
addText 256
contains 259
count 260
empty 261
getMenuChoiceAttribute 262
getMenuChoiceAttributeById 263
hasMenuChoiceAttribute 265
remove 266
removeMenu 267
setMenuChoiceAttribute 268
setMenuChoiceAttributeById 270
show 271
MenuEvent 273
data 273
id 273
isFromUI 275
menuChoice 276
reason 277
setData 278
setId 279
setReason 279
MouseEvent 281
getMousePosition 282
getObjectHit 282
isControlKeyDown 283
isFromUI 284
isInside 284
isLeftDown 285
isMiddleDown 286
isRightDown 286
isShiftKeyDown 287
setControlKeyDown 288
setInside 288
setLeftDown 289
setMiddleDown 290
setMousePosition 290
setRightDown 291
setShiftKeyDown 292
setX 293
setY 294
x 294
y 295
MoveEvent 296
getDestination 296
reason 297
setReason 298
Number 300
abs 301
acos 302
asin 302
atan 303
atan2 303
ceil 304
cos 304
cosh 305
exp 306
floor 306
fraction 307
fv 307
ln 308
log 309
max 309
min 310
mod 311
number 311
numVal 312
pmt 313
pow 314
pow10 314
pv 315
rand 316
round 317
sin 318
sinh 318
sqrt 319
tan 319

- tanh . . . 320
- truncate . . . 320

OLE . . . 322
- canReadFromClipboard . . . 323
- edit . . . 323
- enumVerbs . . . 325
- getServerName . . . 326
- readFromClipboard . . . 327
- writeToClipboard . . . 328

Point . . . 329
- distance . . . 329
- isAbove . . . 330
- isBelow . . . 331
- isLeft . . . 331
- isRight . . . 332
- point . . . 332
- setX . . . 333
- setXY . . . 333
- setY . . . 334
- x . . . 335
- y . . . 335

PopUpMenu . . . 336
- addArray . . . 336
- addBar . . . 337
- addBreak . . . 338
- addPopUp . . . 339
- addSeparator . . . 341
- addStaticText . . . 342
- addText . . . 342
- show . . . 345
- switchMenu . . . 346

Query . . . 348
- executeQBE . . . 348
- isAssigned . . . 350
- query . . . 350
- writeQBE . . . 353

Record . . . 355
- view . . . 355

Report . . . 357
- attach . . . 357
- close . . . 358
- currentPage . . . 359
- design . . . 360
- enumUIObjectNames . . . 360
- enumUIObjectProperties . . . 361
- load . . . 362
- moveToPage . . . 363
- open . . . 363
- print . . . 364
- run . . . 365

Session . . . 367
- addAlias . . . 367
- addPassword . . . 368
- advancedWildcardsInLocate . . . 369
- blankAsZero . . . 370
- close . . . 371
- enumAliasNames . . . 371
- enumDatabaseTables . . . 372
- enumDriverCapabilities . . . 373
- enumDriverInfo . . . 375
- enumDriverNames . . . 376
- enumDriverTopics . . . 377
- enumEngineInfo . . . 377
- enumFolder . . . 378
- enumOpenDatabases . . . 379
- enumUsers . . . 380
- getAliasPath . . . 380
- getNetUserName . . . 381
- ignoreCaseInLocate . . . 381
- isAdvancedWildcardsInLocate . . . 382
- isAssigned . . . 383
- isBlankZero . . . 383
- isIgnoreCaseInLocate . . . 384
- lock . . . 384
- open . . . 385
- removeAlias . . . 386
- removeAllPasswords . . . 387
- removePassword . . . 387
- retryPeriod . . . 388
- saveCFG . . . 388
- setAliasPath . . . 389
- setRetryPeriod . . . 389
- unlock . . . 390

SmallInt . . . 391
- bitAND . . . 392
- bitIsSet . . . 393
- bitOR . . . 394
- bitXOR . . . 395
- int . . . 396
- smallInt . . . 396

StatusEvent . . . 398
- reason . . . 398
- setReason . . . 399

setStatusValue 401
statusValue 401

String 403
advMatch 404
ansiCode 406
breakApart 407
chr 409
chrOEM 409
chrToKeyName 410
fill 410
format 411
ignoreCaseInStringCompares 418
isIgnoreCaseInStringCompares 419
isSpace 419
keyNameToChr 420
keyNameToVKCode 420
lower 421
lTrim 422
match 422
oemCode 423
rTrim 424
search 424
size 425
space 426
string 426
strVal 427
substr 427
toANSI 428
toOEM 428
upper 429
vkCodeToKeyName 430

System 431
beep 431
close 432
constantNameToValue 433
constantValueToName 434
cpuClockTime 434
debug 435
dlgAdd 436
dlgCopy 436
dlgCreate 437
dlgDelete 438
dlgEmpty 438
dlgNetDrivers 439
dlgNetLocks 439
dlgNetRefresh 440
dlgNetRetry 441
dlgNetSetLocks 441
dlgNetSystem 442
dlgNetUserName 442
dlgNetWho 443
dlgRename 444
dlgRestructure 444
dlgSort 445
dlgSubtract 445
dlgTableInfo 446
enumDesktopWindowNames 447
enumFonts 448
enumFormNames 449
enumReportNames 449
enumRTLClassNames 450
enumRTLConstants 450
enumRTLMethods 451
enumWindowNames 452
errorClear 453
errorCode 454
errorLog 455
errorMessage 456
errorPop 456
errorShow 456
errorTrapOnWarnings 457
execute 458
exit 458
fail 459
fileBrowser 460
formatAdd 463
formatDelete 463
formatExist 464
formatSetCurrencyDefault 465
formatSetDateDefault 465
formatSetDateTimeDefault 466
formatSetLogicalDefault 466
formatSetLongIntDefault 467
formatSetNumberDefault 467
formatSetSmallIntDefault 468
formatSetTimeDefault 469
getMouseScreenPosition 469
helpOnHelp 470
helpQuit 470
helpSetIndex 471
helpShowContext 471
helpShowIndex 471
helpShowTopic 472
helpShowTopicInKeywordTable 472
message 473

msgAbortRetryIgnore 473
msgInfo 474
msgQuestion 474
msgRetryCancel 475
msgStop 476
msgYesNoCancel 476
pixelsToTwips 477
play 478
readEnvironmentString 478
readProfileString 479
setMouseScreenPosition 480
setMouseShape 480
sleep 481
sound 482
sysInfo 483
tracerClear 484
tracerHide 485
tracerOff 485
tracerOn 486
tracerSave 486
tracerShow 486
tracerToTop 487
tracerWrite 487
twipsToPixels 488
version 488
winGetMessageID 488
winPostMessage 489
winSendMessage 489
writeEnvironmentString 490
writeProfileString 490

Table 491
add 491
attach 493
cAverage 494
cCount 495
cMax 496
cMin 497
cNpv 498
compact 499
copy 500
CREATE 501
cSamStd 505
cSamVar 506
cStd 507
cSum 508
cVar 509
delete 510
dropIndex 511
empty 512
enumFieldNames 513
enumFieldNamesInIndex 514
enumFieldStruct 515
enumIndexStruct 516
enumRefIntStruct 518
enumSecStruct 519
familyRights 520
fieldName 521
fieldNo 522
fieldType 523
index 524
isAssigned 527
isEmpty 528
isEncrypted 529
isShared 529
isTable 530
lock 531
nFields 532
nKeyFields 532
nRecords 533
protect 534
reIndex 535
reIndexAll 536
rename 536
setExclusive 537
setFilter 538
setIndex 540
setReadOnly 541
showDeleted 541
sort 542
subtract 544
tableRights 545
type 545
unAttach 546
unlock 547
unProtect 548
usesIndexes 549

TableView 550
action 551
close 551
open 552
wait 553

TCursor 554
add 555
atFirst 556

atLast . . . 557
attach . . . 557
attachToKeyViol . . . 559
bot . . . 560
cancelEdit . . . 561
cAverage . . . 562
cCount . . . 562
close . . . 563
cMax . . . 564
cMin . . . 565
cNpv . . . 566
compact . . . 567
copy . . . 568
copyFromArray . . . 569
copyRecord . . . 570
copyToArray . . . 571
cSamStd . . . 572
cSamVar . . . 573
cStd . . . 574
cSum . . . 575
currRecord . . . 576
cVar . . . 576
deleteRecord . . . 577
didFlyAway . . . 578
dropIndex . . . 579
edit . . . 580
empty . . . 581
end . . . 582
endEdit . . . 582
enumFieldNames . . . 583
enumFieldNamesInIndex . . . 584
enumFieldStruct . . . 585
enumIndexStruct . . . 586
enumLocks . . . 588
enumRefIntStruct . . . 589
enumSecStruct . . . 590
enumTableProperties . . . 592
eot . . . 593
familyRights . . . 594
fieldName . . . 594
fieldNo . . . 595
fieldRights . . . 595
fieldSize . . . 596
fieldType . . . 597
fieldUnits2 . . . 598
fieldValue . . . 600
getLanguageDriver . . . 601
getLanguageDriverDesc . . . 601
home . . . 602
initRecord . . . 603
insertAfterRecord . . . 603
insertBeforeRecord . . . 604
insertRecord . . . 605
isAssigned . . . 606
isEdit . . . 607
isEmpty . . . 608
isEncrypted . . . 609
isRecordDeleted . . . 610
isShared . . . 610
isShowDeletedOn . . . 611
isValid . . . 612
locate . . . 613
locateNext . . . 614
locateNextPattern . . . 615
locatePattern . . . 617
locatePrior . . . 619
locatePriorPattern . . . 620
lock . . . 622
lockRecord . . . 622
lockStatus . . . 623
moveToRecNo . . . 624
moveToRecord . . . 625
nextRecord . . . 625
nFields . . . 626
nKeyFields . . . 627
nRecords . . . 628
open . . . 628
postRecord . . . 630
priorRecord . . . 631
qLocate . . . 632
recNo . . . 633
recordStatus . . . 633
reIndex . . . 634
reIndexAll . . . 635
seqNo . . . 636
setFieldValue . . . 637
setFilter . . . 638
setFlyAwayControl . . . 639
showDeleted . . . 640
skip . . . 641
sortTo . . . 641
subtract . . . 643
switchIndex . . . 644
tableName . . . 645
tableRights . . . 646
type . . . 647

unDeleteRecord 647
unlock 648
unlockRecord 649
updateRecord 650
TextStream 651
advMatch 651
close 653
commit 654
create 654
end 656
eof 657
home 657
open 658
position 659
readChars 660
readLine 661
setPosition 662
size 663
writeLine 663
writeString 664
Time 665
time 665
TimerEvent 667
UIObject 669
action 670
atFirst 671
atLast 672
attach 672
broadCastAction 674
cancelEdit 674
convertPointWithRespectTo 675
copyFromArray 676
copyToArray 677
create 678
currRecord 679
delete 679
deleteRecord 680
edit 681
empty 682
end 684
endEdit 684
enumFieldNames 685
enumLocks 685
enumObjectNames 686
enumSource 687
enumSourceToFile 688
enumUIClasses 689
enumUIObjectNames 690
enumUIObjectProperties 691
execMethod 692
getBoundingBox 693
getPosition 694
getProperty 695
getPropertyAsString 696
getRGB 696
hasMouse 697
home 698
insertAfterRecord 698
insertBeforeRecord 700
insertRecord 701
isContainerValid 701
isEdit 702
isEmpty 702
isLastMouseClickedValid 703
isLastMouseRightClickedValid 703
isRecordDeleted 704
keyChar 705
keyPhysical 705
killTimer 706
locate 707
locateNext 708
locateNextPattern 709
locatePattern 711
locatePrior 712
locatePriorPattern 713
lockRecord 715
lockStatus 716
menuAction 717
methodDelete 718
methodGet 719
methodSet 720
mouseClick 720
mouseDouble 721
mouseDown 721
mouseEnter 722
mouseExit 723
mouseMove 723
mouseRightDouble 724
mouseRightDown 724
mouseRightUp 725
mouseUp 726
moveTo 727
moveToRecNo 728
moveToRecord 729

nextRecord 729
nFields 730
nKeyFields 731
nRecords 731
pixelsToTwips 732
postAction 732
postRecord 733
priorRecord 734
pushButton 735
recordStatus 735
resync 736
rgb 737
setFilter 738
setPosition 739
setProperty 740
setTimer 741
skip 742
switchIndex 743
twipsToPixels 744
unDeleteRecord 745
unlockRecord 745
view 745
wasLastClicked 747
wasLastRightClicked 747

ValueEvent 749
newValue 749
setNewValue 750

Part II
Appendixes 753

Appendix A
ObjectPAL methods by type 755

Appendix B
ANSI character set 781

Appendix C
Windows keycodes 783

Appendix D
Keywords 789

Appendix E
Compatibility procedures 791
add 791
cAverage 791
cCount 792
cMax 792
cMin 792
cNpv 792
copy 793
cSamStd 793
cSamVar 793
cStd 793
cSum 794
cVar 794
delete 794
empty 794
familyRights 794
fieldName 795
fieldNo 795
fieldType 795
isEmpty 795
isEncrypted 795
isShared 796
isTable 796
nFields 796
nKeyFields 796
nRecords 796
protect 796
rename 797
subtract 797
tableRights 797

Appendix F
Properties 799

Appendix G
Object PAL Constants 823

Index 849

CHAPTER 1

Introduction

This manual is a reference to the ObjectPAL™ language. It's set up to make it easy for you to perform a series of actions on an object, rather than issue isolated commands. For example, when you write code to work with a table, you may want to open the table, read from it, write to it, and close it. All the methods and procedures you need are in one place in the "Table" section.

What's in this manual

The first chapter, "Built-in methods," presents ObjectPAL's built-in methods. Built-in methods specify how objects respond to events.

The next chapter, "Basic language elements," describes the basic language elements. Basic language elements include keywords for creating control structures, for declaring methods, procedures, and variables, and for other tasks not bound to specific object types.

Methods and procedures are grouped by object type.

Chapter 4, "Object type reference" presents the methods and procedures for each ObjectPAL type. For example, the procedures and methods for working with tables are grouped in one section ("Table"), those for working with disk files are in another section ("FileSystem"), and those for working with numeric data are in another section ("Number").

Note This chapter also describes ObjectPAL language structures associated with specific types. For example, the CREATE, INDEX, and SORT structures are all associated with tables, so they're described in the Table section.

Types are presented in alphabetical order, from ActionEvent to ValueEvent, and within each type, methods, procedures, and structures are listed alphabetically. When you search for a particular method or procedure, ask, "What object am I working with?" and turn to that section of the chapter. Then ask, "What do I want that

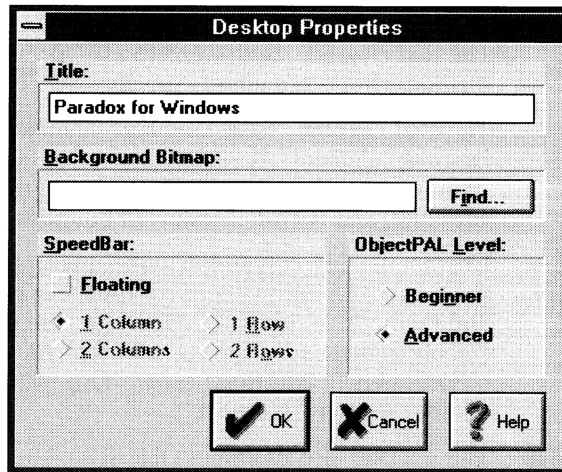
object to do?" (If you need to search for a method by name, turn to the index.)

At the end of the book are appendixes, a glossary, and an index.

ObjectPAL level

By setting a Desktop property, you can configure the ObjectPAL IDE to show you everything in the language, or to select a subset of essential elements (the default setting). To set the ObjectPAL level, choose Properties | Desktop to open the Desktop Properties dialog box, shown in Figure 1-1. Then, in the ObjectPAL Level panel, choose Beginner if all you want to see is the subset, or choose Advanced to see everything. The ObjectPAL level affects the IDE only: code executes identically at either level, and you can use advanced elements in code even when the level is set to Beginner. In this manual, Beginner-level methods and procedures are marked with the word *Beginner*.

Figure 1-1 Setting the ObjectPAL Level

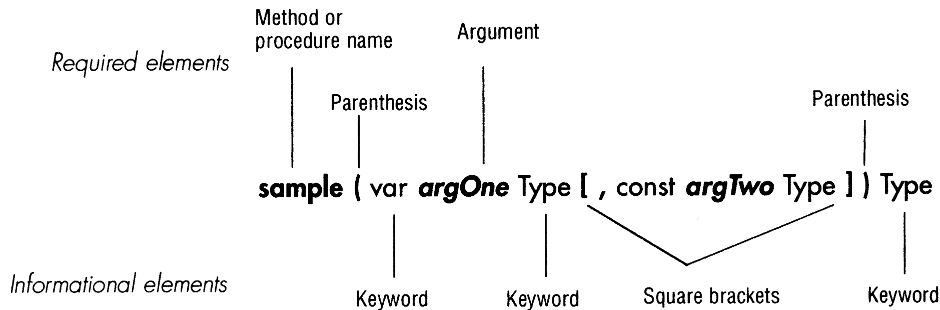


This manual lists every method, procedure, and keyword in the ObjectPAL language. Beginner-level methods and procedures are marked with the word *Beginner*.

Prototype notation

Prototypes are presented for each method and procedure. This section describes the syntax notation used in prototypes throughout this reference. The prototype for each ObjectPAL method and procedure includes elements required as part of the syntax and elements presented for your information, as shown in Figure 1-2.

Figure 1-2 A sample prototype



Required elements

The following elements are required when shown as part of the prototype:

- Name: the name of the method or procedure.
- Parentheses: parentheses are required, even if the method or procedure takes no arguments.
- Argument: if an argument is shown as part of the syntax, it must be included. An argument can be a variable, an expression, or a literal (hard-coded) value. Lists of arguments are separated by commas.

Informational elements

Informational elements are not part of the syntax you type in for the method or procedure; they just tell you how the method or procedure works.

- Keywords: keywords (shown in lightface type) provide information about the arguments for a method or procedure.

  An argument preceded by the keyword **var** is passed by reference. An argument preceded by the keyword **const** is passed as a constant. An argument by itself, without either keyword, is passed by value.

The keyword that follows each argument specifies its data type (for example, String, Number, Table, or Logical).

If a method or procedure returns a value, the keyword at the end of the prototype specifies its data type. Most, but not all, ObjectPAL methods and procedures return values.

Square brackets

Square brackets indicate an optional argument. If a prototype shows an argument enclosed in square brackets, you can include that argument or not, depending on what you want to do.

From prototype to ObjectPAL

Given the prototype presented in Figure 1-2, the following statements are valid:

```
x = sample(custName)      ; one argument, variable x stores the return value

y = sample(custName, custAddress) ; two arguments, variable y stores the return
                                  ; value
```

CHAPTER 2

Built-in methods

This chapter describes the built-in methods for Paradox objects.

This chapter presents the following topics:

- About built-in methods: discusses attaching code to a built-in method and controlling the default behavior of an object.
- Descriptions of the built-in methods: describes the built-in methods for internal events and external events.
- Special built-in methods: describes built-in methods unique to specific objects.
- Built-in object variables: describes ObjectPAL variables for referring to objects in a form.

Important Before you read this chapter, you should be familiar with the ObjectPAL event model, described in Chapter 6 of the *ObjectPAL Developer's Guide.*

About built-in methods

Every object in a form (and the form itself) has built-in methods for handling events. Built-in methods have the same names as the events that trigger them. For example, changing a value triggers an object's built-in **changeValue** method, pressing the mouse button triggers the built-in **mouseDown** method, and releasing the mouse button triggers the built-in **mouseUp** method. The behavior of an object is simply the combined effects of its built-in methods.

There are three kinds of built-in methods in ObjectPAL:

- Built-in methods for internal events—events generated internally by ObjectPAL or Paradox.
- Built-in methods for external events—events typically generated by user actions (although they can also be generated by ObjectPAL statements).
- Special built-in methods—methods built into a few specific objects.

This section discusses built-in methods and built-in variables, then presents some simple examples showing how these methods are used in a form while editing data.

Attaching code to built-in methods

You can attach code to any built-in method by opening an ObjectPAL Editor window and typing some code. For example, every design object has a built-in **mouseClick** method that performs some default behavior (described later) when you click that object. To change that behavior, inspect the object, choose Methods from the Properties menu, then choose **mouseClick** from the list of methods to open an ObjectPAL Editor window. Type your code and save it. Now your code executes whenever this object's **mouseClick** method is called.

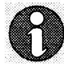
The built-in code executes too, *after* your code (just before the **endmethod** keyword).

The built-in code is implicit and executes automatically. But, if you want to change the default behavior—for example, call the built-in code *before* your code, or block it from executing—you can (as described later in this chapter). First, though, you should understand the default behavior for each built-in method.

Descriptions of the built-in methods

Table 2-1 lists the built-in methods for internal and external events, and the special built-in methods. Following the table are descriptions

that include when each method is called, the default behavior, and the effects (if any) of an error.



This section builds on material presented in Chapters 6 and 7 in the *ObjectPAL Developer's Guide.*

Table 2-1 Built-in methods

Internal	External	Special
open	mouseClick	pushButton
close	mouseDown	changeValue
canArrive	mouseUp	newValue
arrive	mouseDouble	
setFocus	mouseRightDown	
canDepart	mouseRightUp	
removeFocus	mouseRightDouble	
depart	mouseMove	
mouseEnter	keyPhysical	
mouseExit	keyChar	
timer	error	
	status	
	action	
	menuAction	

Built-in methods for internal events

The following methods are triggered by internal events—events generated internally by ObjectPAL. Like all events, internal events go to the form first, which dispatches them to the target object. Internal events do not bubble up through the containership hierarchy.

open

open is called once for every object on the form, starting from the form and working down each container in turn. By default, a form's **open** method opens all tables attached to the form.

An error from any object prevents a form from opening.

For every object, the default code for the **open** method calls the **open** method for each of its child objects (that is, the objects one level below it in the containership hierarchy). In other words, by default, the form's **open** calls the **open** for each page in the form, and each page's **open** calls **open** for each object on the page, and so on.

close

close is called once for every object on a form being closed. By default, a form's **close** method closes all tables attached to the form.

canArrive

canArrive is called when moving to an object. It asks whether the object (usually a field object) can be made active. Paradox calls this method by working through the object's containers, triggering

canArrive for each object until it reaches the object itself. At any level of the containership, an error denies permission, blocking the move.

By default, Paradox blocks **canArrive** for records beyond the ends of a table (except for blank records in Edit mode). The same is true for field objects, which also check the Tab Stop property, blocking moves to field objects that are not tab stops. A crosstab object blocks **canArrive** if the target is not a cell.

arrive **arrive** is called only after the target and its containers have allowed a **canArrive**. As with **canArrive**, Paradox calls **arrive** on the target object's containers, finishing at the target. Pages, table frames, and multi-record objects all move to the first tab stop object they contain.

A successful **arrive** on a record or a field object makes it current (and opens an editing window for the field object, if appropriate).

arrive can have further effects on a field object, depending on its display type. If it's a drop-down edit list, focus moves to the list. If it's a radio button, focus moves to the first button.

setFocus **setFocus** is called after a successful **arrive** or when focus is returned to the form after going away to another window. This method is called for each of the active object's containers, starting with the outermost container, before it is called for the active object itself.

On an edit field, the default code for **setFocus** highlights the current selection and starts blinking the insertion point. At this time, also, the object's Focus property is set to True, and the form displays a status message reporting the number of the current record and the total number of records.

When a button accepts a **setFocus**, a rectangle displays around the label.

canDepart **canDepart** is called when trying to move off any object. Any error blocks the move. Field objects try to post their contents (triggering **changeValue**), and record objects try to commit the current record if changes have been made. If the record is locked, the form calls **action(DataUnlockRecord)**.

removeFocus **removeFocus** removes the flashing insertion point and highlight (if appropriate) from a field object, and removes the rectangle from a button. The object's Focus property is set to False.

This method is called for the active object and all its containers, starting with the active object, when the user activates some other window or moves to some other object.

depart **depart** is called after all containers of the current object have granted permission to leave the field via **canDepart** and **removeFocus**.

Field objects close their edit windows, repaint, and perform general cleanup.

mouseEnter **mouseEnter** is called whenever the pointer crosses into an object. It is called only on the transition into the object, not on every move across it.

By default, field objects set the pointer to the I-beam, and form, page, and button objects set it to an arrow.

If a button was the last object to receive a click and the mouse button is still down, the button's value toggles between True and False.

mouseExit **mouseExit** is called when the pointer leaves an object. An object that sets the shape of the pointer on **mouseEnter** sets it to an arrow in **mouseExit**.

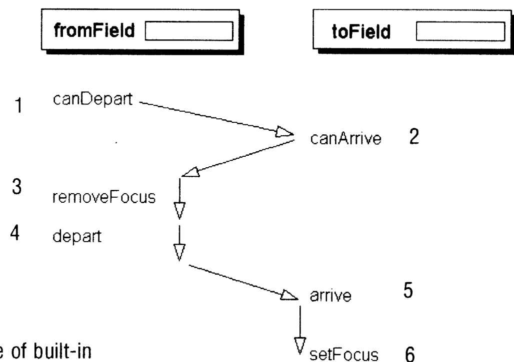
If a button was the last object to receive a click and the mouse button is still down, its value toggles between True and False.

timer **timer** is called each time a timer interval elapses. Use the UIObject method **setTimer** to set timer intervals.

Sequence of execution

Figure 2-1 shows the sequence in which built-in methods execute when you move from one field object to another (for example, by pressing *Tab*). In this example, the field object you're moving from is named *fromField*, the one you're moving to is named *toField*.

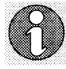
Figure 2-1 Moving from one field object to another



This figure shows the sequence of built-in methods that execute when you move from one field object to another field object.

Built-in methods for external events

The following methods are triggered by external events—events typically generated by user actions—although they can also be generated by ObjectPAL statements. Processing for all external events begins with the form, which acts as a dispatcher. An external event bubbles up the containership hierarchy if an object doesn't handle it.



For most objects and most built-in methods, the default behavior is to pass the event up the containership hierarchy. If an object does something different, is it noted in the following descriptions.

mouseDown

mouseDown is called when the logical left mouse button is pressed. The event packet for this method contains the mouse coordinates in twips, relative to the last object whose **mouseEnter** method was called.

An active field object enters Field View, positions the insertion point and begins a drag-selection.

When the form handles a **mouseDown**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDown** to the object the mouse was pointing at.

This method toggles a button's value between True and False.

mouseUp

mouseUp is called when the left mouse button is released. It's called for the last object to receive a **mouseDown**, even if the button is released outside the object, so the object always sees the **mouseDown/mouseUp** pair.

An active field object ends the selection; a field object that is not active performs a **self.moveTo()**.

When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseUp** to the object that received the last click.

This method toggles a button's value between True and False. If **mouseUp** is called and the pointer is inside a button, it triggers the button's **pushButton** method.

mouseDouble

mouseDouble is called when the left mouse button is double-clicked. In Windows convention, a **mouseDown** and **mouseUp** are delivered first.

Field objects enter field view on a **mouseDouble**.

When the form handles a **mouseUp**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse. The form then dispatches the **mouseDouble** to the object that received the last click.

mouseRightDown
mouseRightUp
mouseRightDouble

These methods are duplicates of the three above, but they refer to the right mouse button.

In addition, the following field objects display a pop-up menu when they get a **rightMouseUp**: formatted memo, graphic, OLE, and unbound (undefined).

mouseClick

mouseClick is called when the logical left mouse button is pressed and released when the pointer is inside the boundaries of an object. **mouseClick** is not called if the user moves the mouse outside the object before releasing the mouse button.

mouseClick is actually generated from within the **mouseUp** method.

The mapping from **mouseUp** to **mouseClick** happens at the first container object which does something with **mouseUp**. In other words, **mouseUp** in a box bubbles to its container, and so forth. Only the field object, the button object, the list object, and the form intercept **mouseUp**, so those are the spots where the translation occurs. If you click on an object inside a button, that object's **mouseClick** will be called. If that object allows the default (bubbling), then the button will ultimately receive that **mouseClick**, triggering its own **pushButton**. In this way, you can have code execute on objects you click inside the button, but still trigger the button's **pushButton** method. Setting the error code in **mouseUp** will inhibit the **mouseClick**, and setting the error code in **mouseClick** will inhibit a **pushButton**.

mouseMove

mouseMove is called whenever the mouse moves within an object. The event packet for this method contains the coordinates of the pointer (in twips).

An active edit field checks the state of *Shift*. If *Shift* is down (physically or logically), the selection is extended. An active graphic field scrolls the graphic, if necessary.

When you press and hold the mouse button inside an object, **mouseMove** is called until you release it, even when the pointer moves outside the object.

When the form handles a **mouseMove**, it calls **mouseExit** for all objects no longer under the mouse, and calls **mouseEnter** for all objects now under the mouse.

keyPhysical

keyPhysical is called when a key is pressed and each time a key autorepeats. It goes to the form first, and the form dispatches it to the active object. Then, the active object's built-in code sorts out whether a keystroke represents an action or a character to display in a field, and calls the appropriate **action** or **keyChar** method (discussed below).

For example, suppose a field object within a table object is active, and the user presses *Enter*. The keystroke triggers **keyPhysical**, which interprets it as a request for an action and maps it to **action(FieldEnter)**, which in turn triggers the built-in **action** method. In contrast, when the user presses *K*, the keystroke triggers **keyPhysical**, which interprets it as a character and and triggers the **keyChar** method.

Technically, the event packet for **keyPhysical** contains information from the Windows WM_KEYDOWN message and an optional WM_CHAR. Therefore, it provides both the virtual key code (listed in Appendix C) as well as the ANSI character (listed in Appendix B). Although you can attach code to this method, it's best if you don't, except for special character handling. For example, if you want to intercept the *F9* key explicitly—rather than handle the eventual **action(DataToggleEdit)**—this is the method to use.

keyChar

keyChar is called when a **keyPhysical** is not interpreted by Paradox. That is, a **keyChar** gets called for every **keyPhysical** that *does not* map to an action (see **action**, below). It goes to the form first, and the form dispatches it to the active object.

When editing a field, the system locks the record before inserting the first character.

If a button receives a **keyChar** equal to pressing *Spacebar* (for example, **keyChar(VK_SPACE)**), it calls the button's **pushButton** method.

menuAction

menuAction is called whenever the user chooses an item from a menu (or clicks a SpeedBar button that executes a menu action). It goes to the form first, and the form dispatches it to the active object. See the *ObjectPAL Developer's Guide* for more information.

error

error is called when an error occurs. By default, objects (except the form) pass errors to their containers. You can attach code to the default method to make an object handle an error, or pass it, or both. For more information about errors and error handling, refer to Chapter 13 in the *ObjectPAL Developer's Guide*.

status

status is called before a message is displayed in one of the areas in the status bar. Among other things, you can attach code to the built-in **status** method to redirect messages to other areas, or to change the text of the message. See Chapter 6 of the *ObjectPAL Developer's Guide* for more information.

action

action is called when **keyPhysical** maps some keystroke to an action, when **menuAction** maps a menu choice to an action, or when other methods want some action performed. It goes to the form first, and the form dispatches it to the target object. For example, by default, pressing *F2* in a field triggers **action(EditToggleFieldView)** after its

keyPhysical method executes, and clicking the Forward navigation button triggers **action(DataNextRecord)** after its **menuAction** method executes.

action is very important method. It is discussed in detail in Chapter 7 in the *ObjectPAL Developer's Guide.*

Special built-in methods

Some objects have additional built-in methods. They are described in this section.

pushButton

Defined only for button objects and fields displayed as list boxes, **pushButton** is called when the user releases the mouse on a button. This method is actually not called directly by the form, but by the default **mouseUp** method for buttons. You can also call this method directly to accomplish the normal action associated with pressing a button object.

By default, buttons change their appearance when clicked. For example, a push button pushes in and pops out, check boxes check or uncheck, and radio buttons push in or pop out. Focus moves to a button when you click it (unless its Tab Stop property is set to False).

When a button's Tab Stop property is set to True and the button is the active object, there are two ways to trigger its **pushButton** method using the keyboard:

- Press *Spacebar.* The button keeps focus.
- Press *Enter.* Focus moves to the next object in the tab order.

changeValue

Defined only for field objects, **changeValue** asks for permission to change the value of a field. It is called *before* the value is stored, so you can check the value and do something with it (such as performing additional validity checks). It is not called when someone changes a value across a network or through a lookup with fill-all-corresponding.

The following statement triggers *Quant*'s **changeValue** method, even if *Quant* is already 10, and it triggers it immediately, without waiting for the method to finish executing.

```
Quant = 10
```

newValue

Defined only for field objects, **newValue** is called to report that a field object has a new value. For example, when scrolling through a table, moving to the next record triggers **newValue**. When a field is displayed as radio buttons, **newValue** is called when you click a button. Note that simply typing into a field object does not trigger **newValue** (but does set the Touched property to True). In any case,

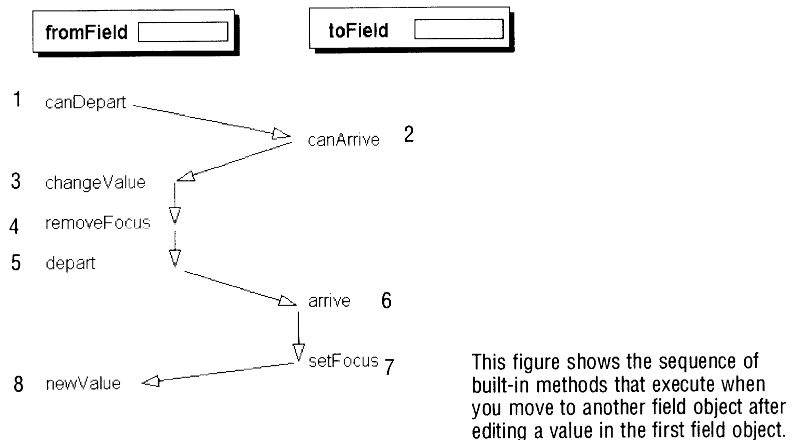
changeValue is not called until you try to move off the field object or otherwise try to commit changes. Also, a form's **open** method triggers **newValue** for each field object in the form.

Labeled and unlabeled field objects

Figure 2-3 shows the sequence in which built-in methods execute when you move from one field object (labeled or unlabeled) to another after editing a value. The field object you're moving from is named *fromField;* the one you're moving to is named *toField.*

Note **newValue** is called when Paradox needs to access the value of the field object (in this case, to update the display). That's why it is called after all the others, as shown in Figure 2-2.

Figure 2-2 Moving from field object to field object after editing a value



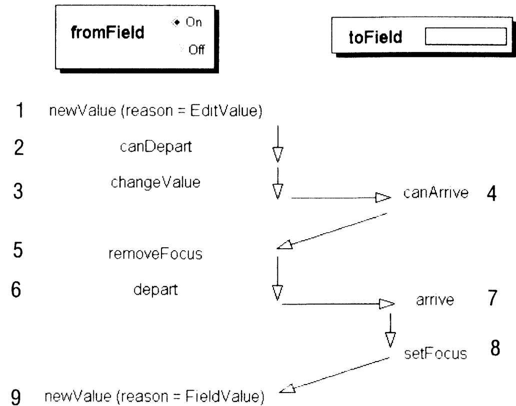
This figure shows the sequence of built-in methods that execute when you move to another field object after editing a value in the first field object.

Radio buttons and lists

Figure 2-3 shows the sequence in which built-in methods execute when you move from one field object (radio buttons, list, or drop-down edit list) to another after editing a value. The table also gives the ValueReason constant for each **newValue**. The field object you're moving from is named *fromField;* the one you're moving to is named *toField.*

Figure 2-3 Moving from a radio button or a list to a field object

The first **newValue** method is triggered when you click the radio button (or choose an item from a list). All other methods are triggered when you initiate the move.



Using changeValue with field objects

The built-in code for **changeValue** commits the changes to the value; until the built-in code executes, Paradox uses the old, unchanged value. For example, suppose a field object has a value of 10, and you move to it and enter a value of 23. Then, when you move off that field object, you trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; displays 10
   doDefault
   msgInfo("after the change", self.value) ; displays 23
endmethod
```

When this method executes, the first dialog box displays the old value, 10, because the built-in code has not yet executed. Then, the call to **doDefault** executes the built-in code, which commits the changed value, and the second dialog box displays the changed value.

Within an object's **changeValue** method, you can use the ValueEvent method **newValue** (not to be confused with the built-in **newValue** method, discussed previously) to get the incoming value before the built-in code executes. For example, suppose as before that a field object has a value of 10, and you move to it, enter a value of 23, and trigger its **changeValue** method, to which the following code has been attached:

```
method changeValue(var eventInfo ValueEvent)
   msgInfo("before the change", self.value) ; displays 10
   msgInfo("the new (incoming) value", eventInfo.newValue()) ; displays 23
   doDefault
   msgInfo("after the change", self.value) ; displays 23
endmethod
```

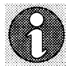
The first dialog box displays the old, unchanged value. The second dialog box calls **eventInfo.newValue** to display the new, incoming value—but that value has not yet been committed. The call to **doDefault** executes the built-in code, which commits the change, and the third dialog box displays the changed value.

For more information about using **changeValue**, refer to Chapter 6 in the *ObjectPAL Developer's Guide.*

Controlling the default behavior

The following basic language elements control when (and if) the built-in code executes:

- **doDefault** executes the built-in code immediately and prevents the built-in code from executing again at the end of the method.
- **disableDefault** blocks the built-in code from executing.
- **enableDefault** allows the built-in code to execute at the end of a method.
- **passEvent** passes the event to an object's container, whether or not the object has handled the event. It does not affect whether the built-in code executes at the end of the method.



You can use these elements only in code attached to built-in methods, not in custom methods or custom procedures.

For example, a new field object's built-in **keyChar** method is

```
method keyChar(var eventInfo KeyEvent)

endmethod
```

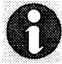
Characters appear when you type them into this field object, because the built-in code executes implicitly just before the **endmethod** keyword.

doDefault

If you attach the following code to the method you might expect each character you type to show up twice: once for your explicit call, and once for the implicit call, but actually, it happens only once.

```
method keyChar(var eventInfo KeyEvent)
   doDefault
endMethod
```

An explicit call to **doDefault** blocks the implicit call to the built-in code.



Although the compiler allows multiple calls to **doDefault**, only the first call has any effect; the others are effectively ignored.

disableDefault

You get the same effect—that is, the implicit code doesn't execute—when you use **disableDefault** or **enableDefault**.

However, if a keyword is in a method but it doesn't execute, the implicit call is not affected. For example,

```
method action(var eventInfo ActionEvent)
   if eventInfo.id() = DataPostRecord then
      if testValue > 10 then
         message("The value is too big.")
         disableDefault ; block the built-in code
      else
         message("The value is OK.") ; execute built-in code implicitly
      endIf
   endIf
endMethod
```

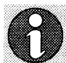
In the previous example, the **disableDefault** keyword executes and the built-in code is disabled only when the value of *testValue* is greater than 10. Otherwise, the built-in code is enabled and executes at the end of the method.

When you want to handle specific cases yourself and let ObjectPAL handle the rest, call **disableDefault** before the **switch** block, then call **enableDefault** or **doDefault** as appropriate in the body of the block. For example, the following code allows the default code to execute only when the action ID is DataNextRecord or DataPriorRecord:

```
method action(var eventInfo ActionEvent)
   disableDefault ; Assume we are not going to do the default

   ; But if the Action Id is either DataNextRecord or DataPriorRecord
   ; then do the default.

   switch eventInfo.Id()
      case DataNextRecord  : enableDefault
      case DataPriorRecord : enableDefault
   endSwitch
endMethod
```

This technique is useful for handling menu choices. For example,

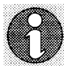
```
method menuAction(var eventInfo MenuEvent)
   var theChoice String endVar
   disableDefault
   theChoice = eventInfo.menuChoice()
   switch
     case theChoice = "Open" : doOpen() ; do custom method, don't do default
     case theChoice = "New"  : doNew()  ; do custom method, don't do default
     otherwise               : enableDefault
   endSwitch
endMethod
```



A **return** statement in a built-in method executes the built-in code immediately before the **return** unless you call **disableDefault** to

block it. For example, the following code calls **disableDefault** to prevent the field object from displaying the character X:

```
method keyChar(var eventInfo KeyEvent)
   if eventInfo.char() = "X" then
      disableDefault
      return
   endIf
endmethod
```



Do not use **disableDefault** in methods for internal events. Blocking the default code for these methods can cause unexpected results. Instead, use **setErrorCode** to set a nonzero error value. The default code checks the error value as part of normal execution; a nonzero error value indicates an error, and the default code takes it into account.

For example, suppose the following code is attached to a field object's built-in **canDepart** method. When this code executes, it compares the date value in the field object with today's date. If the date value in the field object is later than today's date, a dialog box displays a message to inform the user, and the call to **setErrorCode** uses a predefined ObjectPAL constant (CanNotDepart) to specify a nonzero error value. When the default code executes, it "sees" this value and prevents the cursor from leaving the field object.

```
method canDepart(var eventInfo MoveEvent)
   if Date(Self.value) > today() then
      msgStop("Stop", "That date is in the future.")
      eventInfo.setErrorCode(CanNotDepart)
   endIf
endmethod
```

Processing an error value is *not* the same as disabling the default code. For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide.*

enableDefault

enableDefault allows the built-in code to execute at the end of a method. The difference between **doDefault** and **enableDefault** is important: **doDefault** executes the built-in code immediately, and **enableDefault** sets a flag so the built-in code executes at the end of the method. For example, if you attach the following code to a field object, when you type a character into the field, Paradox will wait three seconds, beep, then display the character:

```
method keyChar(var eventInfo KeyEvent)
   enableDefault
   sleep(3000)
   beep()
endMethod
```

In contrast, the following code makes Paradox display the character *before* it sleeps and beeps:

```
method keyChar(var eventInfo KeyEvent)
   doDefault
   sleep(3000)
   beep()
endMethod
```

Calling **doDefault** sets a flag that prevents the built-in code from executing at the end of the method.

You can use these keywords anywhere in a built-in method, but keep in mind that an object's behavior may change depending on where the keywords are used. For example, suppose this method is attached to an undefined field:

```
method keyChar(var eventInfo KeyEvent)
   eventInfo.setChar("K")
   doDefault
endMethod
```

When you run this method and type characters into the field, it displays the letter K, no matter what character you type, because the method sets the character to K and *then* calls the built-in code, which displays the character in the field. If you reverse the order of the statements, the results are different:

```
method keyChar(var eventInfo KeyEvent)
   doDefault
   eventInfo.setChar("K")
endMethod
```

In the previous example, the call to **doDefault** executes the built-in code before the character gets set to K, so the field behaves normally.

passEvent

A **passEvent** statement says, in effect, "Suspend execution, and send the information about this event to my container." It has no effect on the built-in code. If the built-in code is enabled, either implicitly or by using **enableDefault**, it executes at the end of the method. If **passEvent** is called and the built-in code is disabled by **disableDefault** or called by **doDefault**, it doesn't execute at the end of the method.

A call to **passEvent** suspends execution of the calling method, and immediately triggers the appropriate method in the calling object's container. By default, the calling method resumes execution when the container's method finishes.

To see how this works, follow these steps to create and run a simple form:

1. Create a new form.
2. Place large box on the page.
3. Place a field object inside the box, so the box contains the field object.

4. Attach the following code to the field object's built-in **keyChar** method:

```
method keyChar(var eventInfo KeyEvent)
   msgInfo("keyChar", "Calling container's keyChar method.")
   passEvent
   msgInfo("keyChar", "We're back.")
endmethod
```

5. Attach the following code to the box's built-in **keyChar** method:

```
method keyChar(var eventInfo KeyEvent)
   self.color = self.color + 222
   sleep(1000)
endmethod
```

6. Run the form, and type a character into the field object.

Here's what happens when you run this form and type a character into the field object: first, the code attached to the field object's built-in **keyChar** method displays a dialog box. Then, the call to **passEvent** triggers the **keyChar** method of the field object's container, the box. The field object's **keyChar** method suspends execution. When the box's **keyChar** method executes, the box changes color and Paradox sleeps for one second. Then, when the box's **keyChar** method has finished executing, the field object's **keyChar** method resumes execution and displays another dialog box.

For more information, refer to the entries for **doDefault**, **disableDefault**, **enableDefault**, and **passEvent** in Chapter 3.

Built-in object variables

Along with built-in methods, ObjectPAL provides built-in object variables:

Self — *Self* refers to the object to which the currently executing code is attached. For example, when the following statement executes in the **mouseEnter** method attached to *theBox*, *Self* refers to *theBox*.

```
self.color = Red
```

But, suppose a method attached to *theBox* calls a custom method named **changeColor** attached to the page. Suppose the code for **changeColor** is

```
method changeColor()
   self.color = Blue
endMethod
```

When the method attached to *theBox* calls **changeColor**, the page turns blue, not *theBox*. Why? Because *Self* refers to the object to which the code is attached—regardless of which object actually called the code—and in this case, the code is attached to the page. The single

exception to this rule is when *Self* appears in a statement in a library. In this case, *Self* refers to the object that called the library routine.

When an event occurs, *Self* and **eventInfo.getTarget** may refer to the same object, but as events bubble up the containership chain, the target remains fixed while *Self* changes to refer to the object executing the method.

Note *Self* always refers to a UIObject, not to the object's value or the object's name.

Container — *Container* is the object that contains *Self*. For example, suppose a box contains a button, and the button's **pushButton** method is as follows:

```
container.color = Red
```

When this code executes, the box turns red, because the box contains the button, and the button is executing the code.

Active — *Active* is the currently active object—the last object to receive a **moveTo** (for more information about the **moveTo** method, see the UIObject section of Chapter 7 of the *ObjectPAL Developer's Guide*). Typically, the active object is highlighted.

When an active object is set to respond to keyboard events, it is said to have *focus*. Even when focus is removed from an object (for example, to activate another form), *Active* still refers to that object. Only when someone moves off that object is *Active* reset to the new object. Don't underestimate the importance of *Active*, since general routines can be written to operate on the active object without really knowing which one it is. For example, suppose a form contains two table frames, each bound to a different table. The following statement automatically operates on the active table frame:

```
active.action(DataNextRecord)
```

Subject — *Subject* specifies which object a custom method should operate on. For example, suppose a page in a form has a custom method **setColor**, and the code for **setColor** is:

```
method setColor()
   subject.color = red
endmethod
```

Any object on that page can make the following call, and the object named *someObject* will turn red. When **setColor** executes, it replaces *Subject* with *someObject*.

```
someObject.setColor()
```

For more information and examples, refer to Chapter 7 in the *ObjectPAL Developer's Guide*.

LastMouseClicked — *LastMouseClicked* refers to the last object to receive a **mouseDown**. It is reset when the mouse button is released, but only after the object has been given a chance to do its **mouseUp**.

LastMouseRightClicked — *LastMouseRightClicked* is the same as *LastMouseClicked*, but for the right mouse button.

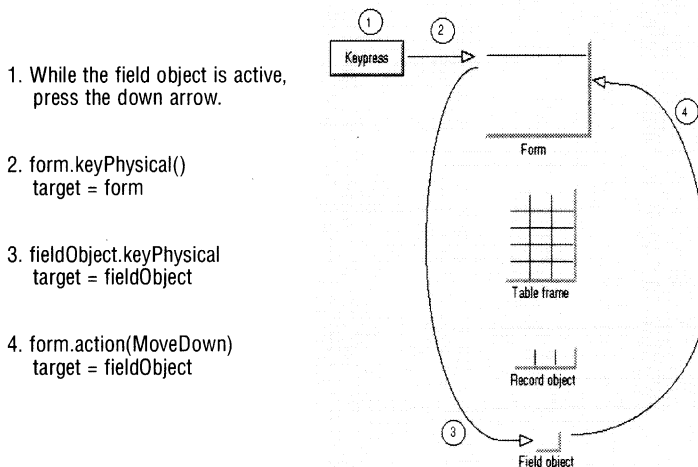
Advanced topic: Sample flow of method calls

Note This section presents technical information for advanced programmers. You don't need to master this material to use ObjectPAL effectively.

Here's a sample set of actions. Suppose you just finished typing into a field object in the third row of a table frame and pressed the ↓. Following is the flow of events and methods (diagrammed in Figures 2-4, 2-5, and 2-6):

Figure 2-4 Steps 1 through 4

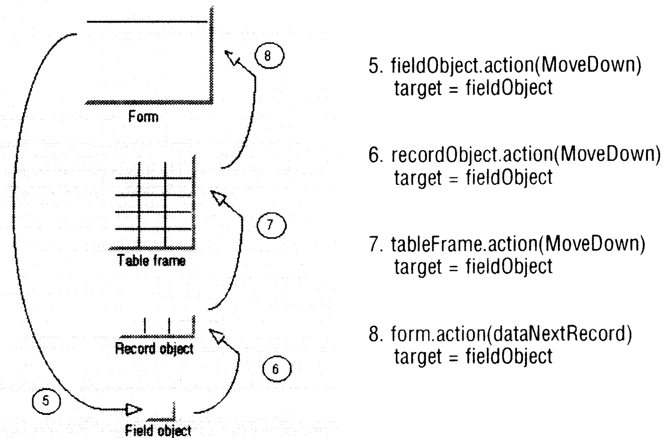
1. While the field object is active, press the down arrow.
2. form.keyPhysical()
   target = form
3. fieldObject.keyPhysical
   target = fieldObject
4. form.action(MoveDown)
   target = fieldObject



1. Press ↓.
2. Paradox constructs a KeyEvent event packet and calls the form's **keyPhysical** method. The form is the target.
3. In the KeyEvent event packet, the form sets the target to be the field object (because it's active) and dispatches the event to the field object's **keyPhysical** method.

4. The field object's default **keyPhysical** method maps the keystroke to an action. It constructs an ActionEvent event packet with the action constant MoveDown, and calls the form's **action** method. In other words, it sends **action(MoveDown)** to the form, and the field object is the target.

Figure 2-5 Steps 5 through 8



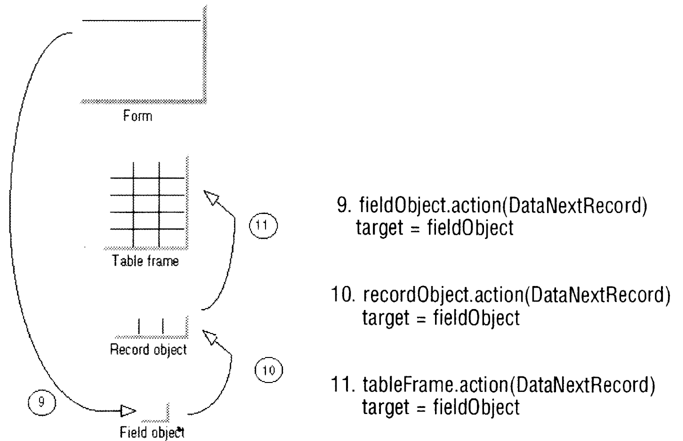
5. The form's **action** method dispatches the event to the field object's **action** method.

6. The field object's **action** method doesn't know what MoveDown means, so it bubbles it up one level to the surrounding record object.

7. The record object's **action** method doesn't know MoveDown, either, so it also bubbles it up one level to the table object.

8. The table object knows what MoveDown means and translates it to **action(DataNextRecord)**, calling the form's **action** method, again with the field object as the target.

Figure 2-6 Steps 9 through 11



9. The form dispatches the event to the field object's **action** method with the constant DataNextRecord.

10. The field object's **action** method bubbles the DataNextRecord to the record again.

11. The record bubbles it to the table frame.

What happens next depends on the size of the table frame. If it has a record below the current record, Paradox simply moves to the corresponding field object in that record. If the table frame does not display another record (in other words, if the current record is the last record displayed) and Paradox has to scroll the table frame, the table frame handles the DataNextRecord as follows:

- It generates a **moveTo** the table frame, causing a **canDepart**, a **removeFocus** and a **depart** to happen to the field object and the record object. This causes the field object's value to be posted to the record (but not to the underlying table) and the selection highlight to disappear.
- If the record has been modified, its **depart** method calls the **action** method of the table frame with the constant DataUnlockRecord. The default for this attempts to post the record to the underlying table, and, if the database engine accepts it, returns a success code of zero.
- It asks the database engine to move forward one record.
- If it succeeds, the table frame determines which record is now current onscreen.

- It generates a **moveTo** the next field object, causing a **canArrive**, an **arrive**, and a **setFocus** to be called for the new record and field objects.

This may seem like a lot of steps, but they execute quickly. The benefit to you, the programmer, is a great deal of flexibility and control.

CHAPTER 3

Basic language elements

This chapter presents the fundamental structural elements of ObjectPAL. Most of these elements are not bound to specific object types; they work for all object types. You can use these elements to assign values, call functions from DLLs, build control structures like **if**...**then**...**else**...**endIf** loops, **while**...**endWhile** loops, and **switch**...**case**...**endSwitch** structures. You can also declare methods, procedures, constants, variables, and data types.

The basic language elements are listed here and are described in the following sections:

=	passEvent
const	proc
disableDefault	quitLoop
doDefault	return
enableDefault	scan
for	switch
forEach	try
if	type
iif	uses
loop	var
method	while

=

Keyword

Syntax *itemSpec = expression*

Description

The = statement assigns the value of *expression* to *itemSpec*. Any previous value stored in *itemSpec* is lost. When assigning a value to an object, information in *itemSpec* can include the containership path.

In a statement containing more than one = statement, the first = does the assignment, all others compare two values or expressions.

When you use = with UIObjects, you assign the value of one UIObject to another UIObject. For example, suppose a form contains two fields, *fieldOne* and *fieldTwo*. The following statement copies the value of *fieldTwo* into *fieldOne*.

```
fieldOne = fieldTwo ; fieldOne gets the value of fieldTwo
```

You can also use = with UIObject variables. ObjectPAL uses **attach** the way C and Pascal use pointers. For example,

```
var ui UIObject endVar
ui.attach(fieldOne) ; tells ui to "point to" fieldOne
ui.view() ; displays the value of ui (same as fieldOne) in a dialog box.
ui = fieldTwo ; ui gets the value of fieldTwo (fieldOne value changes, too)
ui.view() ; displays the value of ui (same as fieldTwo) in a dialog box
ui.color = Red ; sets the color of ui and therefore of fieldOne to red
```

The following statement assigns to *ui* everything ObjectPAL knows about *fieldOne*:

```
ui.attach(fieldOne)
```

In contrast, the following statement assigns to *ui* (and to *fieldOne*) only the value of *fieldTwo*:

```
ui = fieldTwo
```

For more information about **attach**, see Chapter 7 in the *ObjectPAL Developer's Guide.*

Example

```
var
   x AnyType
   ar Array[5] AnyType
   w Logical
   y, z SmallInt
   fred, sam UIObject
endVar

x = 5.14               ; x gets a value of 5.14 (the data type is Number)
ar[1] = "Hello"        ; element 1 of ar gets the value of "Hello" (String)
y = 5                  ; y gets the value of 5
z = 12                 ; z gets the value of 12
x = "foo"              ; x gets a new value: the String "foo"
myTable.myField = y + z ; the field myField gets the value of y + z
amountField = tempAmountField
bigBox.bigCircle.smallBox.smallCircle.color = Blue
; the color property of smallCircle gets the value of Blue

; the first = assigns a value, all others compare
w = (y = z) ; w gets a value of True if y = z,
            ; otherwise, w gets a value of False
```

```
fred.attach(fieldOne) ; makes fred a "pointer" to fieldOne
sam = fred ; assigns the value of fred to sam
```

See also

- Chapters 5 and 7 in the *ObjectPAL Developer's Guide* for more information about containership, properties, and variables

const

Keyword

Declares constants.

Syntax

const
constName = *dataType*(*value*) | *value*
endConst

Description

const declares one or more constant values, where *dataType*, if included, specifies the data type of the constant. If *dataType* is omitted, the data type is inferred from value as either a LongInt, a Number, a SmallInt, or a String.

Refer to Chapter 5 in the *ObjectPAL Developer's Guide* for more information about using constants, including how to pass a value as a constant.

Example

```
const
   a = -1000                        ; SmallInt, inferred
   x = 123.45                       ; Number, inferred
   newYear = Date ("01/01/99")      ; Date, assigned
   companyName = String ("Borland") ; String, assigned
endconst
```

See also

- var

disableDefault

Keyword

Disables the default code for a built-in method.

Syntax

disableDefault

Description

disableDefault prevents an event's built-in code from executing. Normally, the built-in code executes implicitly at the end of a method, just before the **endmethod** statement. Using **disableDefault** in a method disables the implicit call to the built-in code.

Example

The following example sets the value of a field to "hello" when the user types a character. The call to **disableDefault** prevents the built-in code from executing, so the character does not display in the field. The **message** statement displays the character in the status window.

```
method keyChar(var eventInfo KeyEvent)
   self.value = "hello"       ; hello displays in the field
   disableDefault             ; disable the built-in code
   message(eventInfo.char())  ; displays the character in the status window
endMethod
```

See also

- doDefault, enableDefault, passEvent
- Chapter 6 in the *ObjectPAL Developer's Guide*

doDefault

Keyword

Executes the default code for a built-in method.

Syntax

doDefault

Description

doDefault executes the built-in code for an event immediately, instead of at the end of the method. Using **doDefault** in a method disables the implicit call to the built-in code. If a method contains more than one **doDefault** statement, only the first one executes; others are ignored.

Example

In the following method, the button pushes in, waits two seconds, then the system beeps and the button pops out. The built-in code is called implicitly, just before the **endMethod** statement:

```
method pushButton(var eventInfo Event)
   sleep(2000)
   beep()
endMethod
```

In the following method, the call to **doDefault** makes the button pop out before it sleeps and beeps, and it disables the implicit code at the end of the method:

```
method pushButton(var eventInfo Event)
   doDefault
   sleep(2000)
   beep()
endMethod
```

See also

- disableDefault, enableDefault, passEvent
- Chapter 6 in the *ObjectPAL Developer's Guide*

enableDefault

Keyword

Enables the default code for a built-in method.

Syntax

enableDefault

Description

enableDefault allows the built-in code to execute normally at the end of a method, just before the **endmethod** statement. Compare **enableDefault** to **doDefault**, which executes the built-in code immediately.

Example

```
method menuAction(var eventInfo MenuEvent)
var theChoice String endVar
disableDefault
theChoice = eventInfo.menuChoice()
switch
   case theChoice = "Open" : doOpen()
   case theChoice = "Quit" : doQuit()
   otherwise              : enableDefault
endSwitch
endMethod
```

See also

- disableDefault, doDefault, passEvent

for

Keyword

Executes a sequence of statements a specified number of times.

Syntax

for *counter* **[from** *startVal* **] [to** *endVal* **] [step** *stepVal* **]**
   Statements
endFor

startVal, *endVal*, and *stepVal* are values or expressions representing the beginning counter value, ending counter value, and the number by which to increment the counter each time through the loop. These counters can be any data type represented by AnyType, *except* Point, Memo, Graphic, String, OLE, and Binary.

Description

for executes a sequence of *Statements* as many times as is specified by a counter, which is stored in *counter* and controlled by the optional **from, to,** and **step** keywords. Any combination of these can be used to specify the number of times the statements in the loop are executed. You don't have to declare *counter* explicitly, but a **for** loop runs faster if you do.

You can use **for** without the **from, to,** and **step** keywords:

- If *startVal* is omitted, the counter starts at the current value of *counter*.
- If *endVal* is omitted, the **for** loop executes indefinitely.
- If *stepVal* is omitted, the counter increments by 1 each time through the loop.
- *startVal, endVal,* and *stepVal* are stored in a temporary buffer; they are not evaluated each time through the loop.

If **quitLoop** is used within the body of statements in the **for** loop, the **for**...**endFor** loop is exited. If **loop** is used within the body of statements, statements following **loop** are skipped, the counter is incremented, and iteration continues from the top of the **for** loop.

If **step** is positive and a **to** clause is present, iteration continues as long as the value of *counter* is less than or equal to the value of *endVal*. If **step** is negative, iteration will continue as long as the value of *counter* is greater than or equal to the value of *endVal*. In either case, once the value of *counter* reaches or exceeds the limit set by **step**, the **for** loop stops executing, but *counter* keeps its value, as shown in the example.

If *counter* has not previously been assigned a value, **from** creates the variable and assigns to it the value of *startVal*.

Example

Following is a simple **for** loop. Notice the value of the counter variable *i* after the **for** loop is completed:

```
var i SmallInt endVar
for i from 1 to 3
   i.view("Inside for loop") ; i = 1, i = 2, i = 3
endFor
i.view("Outside for loop")   ; i = 4
```

See also

- loop, quitLoop, while

forEach

Keyword

Repeats the specified statement sequence over elements within a DynArray.

Syntax

forEach *VarName* **in** *DynArrayName*
   Statements
endForEach

Description

forEach steps through the elements in a DynArray. In general, you cannot use the **for** statement to step through a DynArray because the indexes of a DynArray are not necessarily integers.

Because DynArray indexes are not integers, DynArray elements are not ordered sequentially. The **forEach** statement operates on DynArray elements in an arbitrary order. You should not rely on a specific ordering of indexes.

If *DynArrayName* does not exist, the **forEach** statement causes an error when the method is compiled.

If the **quitLoop** statement is used within the body of statements in the **forEach** loop, the **forEach**...**endForEach** loop is exited. If the loop statement is used within the body of *Statements*, the statements following **loop** are skipped and iteration continues from the top of the **forEach** loop.

Example

The following example uses the **forEach** statement to display the elements in the DynArray created by the sysInfo statement:

```
var
   SystemArray DynArray[] AnyType
   Element String
endVar
sysInfo(SystemArray)
forEach Element IN SystemArray
   message(Element, " : ", SystemArray[Element])
   sleep(1500)
endForEach
```

See also

- for, loop, quitLoop, while

if

Keyword

Executes one of two sequences of statements, depending on the value of a logical condition.

Syntax

if *Condition* **then**
   Statements1
[else
   Statements2 **]**
endIf

Description

When ObjectPAL comes to an **if** statement, it evaluates whether the *Condition* is true. If so, it executes the statements listed in *Statements1* in sequence. If not, it skips *Statements1* and, if the optional else

keyword is present, executes the statements in *Statements2*. In either case, execution continues after the **endIf** keyword.

An **if** construction can span several lines, especially if there are many statements in *Statements1* or *Statements2*. It is good practice to indent the **then** and **else** clauses to show the flow of control:

```
if Condition then
   Statements1
else
   Statements2
endIf
```

The following is an example of an **if** statement:

```
if Stock < 100 then
   AddStock()         ; execute a custom method called AddStock()
   Stock = Stock + 10 ; then, add 10 to the value of Stock
endIf
```

if statements can be nested; that is, any of the statements in *Statements1* or *Statements2* can also be **if** statements. Nested **if** statements must be fully contained within the controlling **if** structure, in other words, each nested **if** statement must have an **endIf** within the nest. Each **if...endIf** set must enclose code or code and another complete **if...endIf** set:

```
if Condition then
   if Condition then
      Condition
   endIf
endIf
```

Example

The following example provides code for a nested **if** statement:

```
if skillLevel = "Beginner" then
   if skillBox.color = "Red" or skillBox.color = "Yellow" then
      skillBox.color = "Green"
   endIf
endIf
```

See also

- for, iif, switch, and while

iif

Keyword

Returns one of two values, depending on the value of a logical condition.

Syntax

iif (*Condition, ValueIfTrue, ValueIfFalse* **)**

Description

iif (immediate **if**) allows branching within a single statement. You can use **iif** anywhere you can use any other expression. **iif** is

especially useful in calculated fields on forms or reports because **if...endIf** statements are illegal there.

Example

```
a = iif(x > 1, b, c) ; if x > 1, a = b; else a = c
```

See also

- if

loop

Keyword

Passes control to the top of the nearest enclosing **for, forEach, scan,** or **while** loop.

Syntax

loop

Description

When executed within a **for, forEach, scan,** or **while** structure, **loop** skips the statements between it and the **endFor, endForEach, endScan,** or **endWhile** and returns to the beginning of the structure. Otherwise, **loop** causes an error.

Example

```
var x SmallInt endVar
for x from 1
   if x <> 5 then
      loop ; go back to for statement, get next value of x
      message("This never appears") ; this statement never executes
   else
      quitLoop                     ; break out of the loop
   endIf
endFor
message(x) ; displays 5
```

See also

- for, forEach, quitLoop, scan, while

method

Keyword

Defines an ObjectPAL method.

Syntax

method *Name* (*parameterDesc* [,*parameterDesc*]*]) [*returnType*]
[**type** *section*]
[**const** *section*]
[**var** *section*]
Statements
endMethod

Description

method marks the beginning of a method. At a minimum, you must provide the following:

- The method name, in *Name*
- Parentheses, even if the method has no arguments
- The *Statements* that comprise the method

The definition ends with the mandatory **endMethod** keyword.

Additionally, you can declare constants, data types, variables and procedures before the **method** keyword, and you can declare variables and constants after **method**.

Also optional are one or more parameter descriptions, represented in the prototype by *parameterDesc*, where each description takes the form

[var|const] ***parameter type***

The optional *returnType* declares the data type of the value returned by the method. *returnType* is optional because a method may or may not return a value. However, if the method returns a value, you must specify the data type of the value.

Methods and procedures are similar. The key differences are:

- Methods are visible and exportable to other objects, while procedures are private within a containership hierarchy.
- A method can contain a procedure definition, but a procedure can't contain a method definition.

Note The scope of a method depends on where it is declared. For more information, refer to Chapters 5 and 7 in the *ObjectPAL Developer's Guide*.

Example

```
method pushButton (var eventInfo Event)
   var
      txt String
      myNum Number
   endVar
   myNum = 123.321
   txt = String(myNum)
   msgInfo("myNum = ", txt)
endMethod
```

See also

- proc

passEvent

Keyword

Passes the event to the object's container.

Syntax

passEvent

Description

passEvent passes the event packet to the object's container. Using **passEvent** in a method does not affect the implicit call to the built-in code.

Example

The code in the following example is attached to a field object. It executes when the pointer is in the field object. If *Shift* is held down when the mouse is pressed, the code calls **disableDefault** to prevent the built-in code from executing and calls **passEvent** to send the event to the field object's container. This technique is useful when you want several objects to respond the same way to a given event.

```
method mouseDown(var eventInfo MouseEvent)
   if eventInfo.isShiftKeyDown() then
      disableDefault
      passEvent ; let container handle it
   endIf
endMethod
```

See also

- disableDefault, doDefault, enableDefault
- Chapter 6 in the *ObjectPAL Developer's Guide*

---

proc

Keyword

Defines an ObjectPAL procedure.

Syntax

proc *ProcName* (*parameterDesc* [,*parameterDesc*]*]) [*returnType*]
[**const** *section*]
[**type** *section*]
[**var** *section*]
Statements
endProc

Description

proc begins the definition of a procedure. You provide the following:

- The procedure name, in *ProcName*
- Parentheses, even if the procedure has no arguments
- One or more parameter descriptions, represented in the prototype by *parameterDesc*, where each description takes the form

  [**var** | **const**] ***parameter type***

- Use *returnType* to declare the data type of the value returned by the procedure (if it returns a value)

- Sections to declare variables, constants, and types
- The *Statements* that comprise the procedure

The definition ends with the mandatory **endProc** keyword.

You can use return in the body of a procedure to return a value to the calling method or procedure.

A procedure used in an expression must return a value, such as

```
x = NumValidRecs("Orders")   ; NumValidRecs is a procedure
```

Procedures and methods are similar. The key differences are:

- Methods are visible and exportable to other objects, while procedures are private within a containership hierarchy.
- A method can contain a procedure definition, but a procedure can't contain a method definition.

Note The scope of a procedure depends on where it is declared. For more information, refer to Chapters 5 and 7 in the *ObjectPAL Developer's Guide*.

Example

```
proc inc (x SmallInt) SmallInt
   return x + 1
endProc
method pushButton(var eventInfo Event)
   var x SmallInt endVar
   x = 5
   x = inc(x) ; calls the procedure
   message(x) ; displays 6
endmethod
```

See also

- method
- Chapter 5 in the *ObjectPAL Developer's Guide*

quitLoop

Keyword

Terminates the **for**, **forEach**, **scan**, or **while** loop in which it appears.

Syntax

quitLoop

Description

quitLoop exits immediately from the closest enclosing **for**, **forEach**, **scan**, or **while** loop. The method continues with the statement following the closest **endFor**, **endForEach**, **endScan**, or **endWhile**.

quitLoop causes an error if executed outside of any **for**, **scan**, or **while** structure.

Example

In this example, **quitLoop** is used in a **for** loop that determines whether an array has any unassigned elements:

```
var
   myArray Array[12]
   notAssigned Logical
endVar
notAssigned = False
for i from 1 to myArray.length()
   if not isAssigned(myArray[i]) then
      notAssigned = True
      quitLoop
   endIf
endFor
```

See also

- loop, return

return

Keyword

Returns control from a method or procedure, optionally passing back a value.

Syntax

return [*Expression*]

Description

return is used to return control from the current procedure or method to the procedure or method that called it. The following apply to **return**:

- The procedure must be declared to return a value before you can use **return**.
- If **return** is executed within the body of a procedure, the procedure is exited.
- If **return** is executed within a method (but outside the body of a procedure), the method is exited.

You can optionally return the value of *Expression* when returning from either a procedure or a method. If a procedure is called in an expression, then the procedure must return a value, which becomes the value of the procedure call.

```
y = myProc(x) + 3      ; myProc is a procedure
```

If a procedure is called in a standalone context, then any returned value is ignored. For example:

```
myProc(x)
```

If no *Expression* is supplied, **return** must not be followed by anything else on the line other than a comment.

The following data types *cannot* be returned: DDE, Database, Query, Session, Table, or TCursor.

It is not necessary to use **return** to pass control back to a higher-level method or procedure since this happens automatically when a lower-level method or procedure finishes. However, if the method or procedure is declared to return a value, you must use **return** to return the value; the value won't be returned automatically.

Example

Following is a simple example that adds 1 to the value of a variable and returns the new value to the calling method:

```
proc addOne (x SmallInt) SmallInt
   return x + 1
endProc
```

In a built-in method, a **return** statement executes the built-in code unless you explicitly disable it. For example, the following code calls **return** when the user types a "**?**" into a field object. The call to **disableDefault** prevents the built-in code from displaying the "?" in the field object.

```
method keyChar(var eventInfo KeyEvent)
   if eventInfo.char() = "?" then
      disableDefault
      return
   endIf
endmethod
```

See also

- quitLoop

scan

Keyword

Scans the TCursor and executes ObjectPAL instructions.

Syntax

scan *tcVar* **[for** *booleanExpression***]** **:**
Statements
endScan

The colon is required to separate **scan** from a following **for** statement.

Description

scan scans *tcVar* (TCursor) and executes *Statements* (ObjectPAL instructions) for each record. **scan** always begins at the first record of the table, and steps through each record in sequence. When statements in the **scan** loop change an indexed field, that record moves to its sorted position in the table, so it's possible to encounter the same record more than once in the same loop.

If you supply the **for** clause, *Statements* execute only for those records that satisfy the condition; all others are skipped. If the table is empty or if no records meet the condition, the **scan** has no effect.

Note **for** is a keyword to **scan**, so it must be followed by a colon to differentiate it from a **for** loop.

scan is extremely powerful in that you can first prototype a statement sequence for a single record of a table, then place that sequence inside a **scan** loop to make it work on an entire table.

You can use **loop**, **return**, and **quitLoop** in the body of the **scan**. **loop** skips the remaining statements between it and **endScan**, moves to the next record, and returns to the top of the **scan** loop. **quitLoop** terminates the **scan** altogether, leaving the record being scanned as the current record.

Since **scan** repeats an entire statement sequence for each record, don't include actions that only need to be performed once for the table. Put those statements outside the **scan** loop. **scan** automatically moves from record to record through the table, so there's no need to call **nextRecord**.

Example

This example uses a **scan** loop to update the *Employee* table. It scans the Dept field of each record, and if the value is "Personnel", changes it to "Human Resources".

```
var
    empTC TCursor
endVar

empTC.open("employee.db") ; These statements need only be executed once,
empTC.edit()              ; so they're placed outside the loop.

scan empTC for empTC.Dept = "Personnel":  ; the colon is required
   empTC.Dept = "Human Resources"
endScan

empTC.endEdit()
empTC.close()
```

See also

❐ if, for, while

switch

Keyword

Executes one of a set of alternative statement sequences depending on which of several conditions is met.

Syntax

switch
  CaseList

[otherWise: ***Statements*]**
endSwitch

CaseList is any number of statements in the following form:

case Condition : Statements

Description

switch uses the values of the *Condition* statements in *CaseList* to determine which sequence of *Statements* should be executed, if any. **switch** works like multiple if statements, and each *CaseList* works like a single **if** statement.

The case *Conditions* are evaluated in the order in which they appear:

- If one has a value of True, the corresponding *Statements* sequence is executed and the rest are skipped.
- If none has the value True, and the optional **otherWise** clause is present, the *Statements* in **otherWise** are executed.
- If none has the value True and no **otherWise** clause is present, **switch** has no effect.

Thus, one set of *Statements* is executed at most. The method resumes with the next statement after **endSwitch**.

Example

This example creates an array of 100 random numbers, then uses the bubble sort algorithm to sort them in numerical order:

```
method pushButton(var eventInfo Event)
var
   sz, i , itmp, j,k SmallInt
   a Array[100] SmallInt
   tmp Number
endVar

   sz = 100
   a.fill(0)

for i from 1 to sz step 1
   tmp = Rand()
   switch
         case tmp < .1 : a[i] = 1
         case tmp < .2 : a[i] = 2
         case tmp < .3 : a[i] = 3
         case tmp < .4 : a[i] = 4
         case tmp < .5 : a[i] = 5
         case tmp < .6 : a[i] = 6
         case tmp < .7 : a[i] = 7
         case tmp < .8 : a[i] = 8
         case tmp < .9 : a[i] = 9
         otherwise:      a[i] = 10
   endSwitch
endFor

for i from 1 to sz-1 step 1
   for j from 1 to sz-i step 1
      if a[j] <> a[j+1] then
```

```
        a.exchange(j, j+1)
      endIf
   endFor
endFor

endMethod
```

See also

- if, while

try

Keyword

Marks a block of statements to try, and specifies a response should an error occur.

Syntax

try
  [*Statements*] ; the transaction block
onFail
  [*Statements*] ; the recovery block
  [reTry] ; optional
EndTry

Description

The mechanism for building error recovery into an application is the **try**...**onFail** block (described in Chapter 13 of the *ObjectPAL Developer's Guide*).

The transaction block is a set of *Statements*, all of which you want to succeed. If the transaction succeeds, the program skips to **endTry**. If the transaction fails, the recovery block executes. You can call **reTry** to execute the transaction block again.

A trial is caused to fail by the program calling the System procedure **fail** at some point within the transaction block, or within procedures called by the transaction block. This stops system functions from returning status errors or null values to their callers.

A **fail** call can be nested in several procedure calls deep from where the block began. Their local variables are removed from the stack, and any special objects (such as large Text blocks) are deallocated. If reference objects (such as tables) are in use, they are closed, and any pending updates are canceled. It's as if the transaction had never started. What remains are changes to variables outside the block, or data added successfully to tables and committed before the failure occurred.

If during a recovery block you decide that the error code is not one you expected, or is more serious than can be handled at this level, call **fail** again to pass that error code. If no higher-level **try**...**onFail**

block exists, the whole application fails, cancels existing actions, closes resources, and exits.

Example

This example tries to set the Color property of some design objects, and uses a **try**...**onFail** block to handle the situation if the property cannot be set.

```
method pushButton(var eventInfo Event)
var s String endVar
box1.box2.color = Blue                    ; this works
s = "box5"                                ; box5 doesn't exist

try
   box1.(s).color = Red                   ; try to set color of box5
onFail                                    ; handle the error
   msgStop("Error", "Couldn't find " + s)
   s = "box2"                             ; box2 exists
   reTry                                  ; try again
endTry

s = "box6"                                ; box6 doesn't exist
try
   box1.(s).color = Green
onFail
   fail(peObjectNotFound, "The object " + s + "does not exist.")
endTry
endMethod
```

See also

- fail in the System type

type

Keyword

Declares data types.

Syntax

type
  [*newTypeName* = *existingType*]*
endType

Description

Using **type**, you can define new data types. Once defined, you can use these types to declare variables in methods.

For example, an application to track the number of parts in a warehouse might declare a **type** *partQuantity*, then declare a variable to be of **type** *partQuantity*, like this:

```
type
   partQuantity = SmallInt ; declare a new type
endType

var                     ; use the new type to declare a variable
   pQty partQuantity ; pQty is a SmallInt
endVar                  ; because partQuantity is a SmallInt
```

Later, if the number of parts approaches 32,767 (the maximum value of a SmallInt), you need only change the **type** definition, for example,

```
type
   partQuantity = LongInt ; change the declaration
endType

var                    ; use the new type to declare a variable
   pQty partQuantity ; pQty is now a LongInt
endVar                 ; because partQuantity is a LongInt
```

Example

A useful **type** is the **record**. Records defined in an object's type window have no connection to tables. Instead, they are similar to records in Pascal and STRUCTs in C, because they allow you to join several related elements of data together under one name. For example, the following code declares a **record** *Employee* that you can use to declare variables in methods and procedures:

```
type
   Employee = record
              LastName   String
              FirstName  String
              Title      String
              Salary     Currency
              DateHired  Date
              endRecord
endType
```

See also

- var
- Chapter 5 of the *ObjectPAL Developer's Guide* for information about using the Type window to declare types
- The discussion of the **fileBrowser** procedure in Chapter 11 of the *ObjectPAL Developer's Guide* for information about declaring and using records
- The discussion of the Record type in Chapter 9 of the *ObjectPAL Developer's Guide*

uses

Keyword

Declares external library routines to use in a method or procedure.

Syntax

uses [*library* | *ObjectPAL*]
   routineName (*parameterList*)
endUses

Description

The **uses** block, declared in an object's Uses window, makes routines stored in external libraries available to ObjectPAL methods. The routines must fit one of the following descriptions:

- Routines written in ObjectPAL and stored in an ObjectPAL library. ObjectPAL libraries are described in Chapter 11 of the *ObjectPAL Developer's Guide.*
- Routines written in ObjectPAL and attached to a form. The syntax for calling methods attached to a form is the same as for calling them from a library.
- Routines written in assembly language, C, C++, or Pascal and stored in an ObjectPAL library or a Windows Dynamic-link library (DLL). A DLL is a library of executable code or data that you can link to your application at runtime. Using DLLs you can add features and functions without modifying your compiled ObjectPAL application.

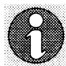
A **uses** block for an ObjectPAL library differs slightly from a **uses** block for a DLL, so they're discussed in separate sections.

uses block for an ObjectPAL library

To use methods stored in an ObjectPAL library or attached to a form, write a **uses** block in an object's Uses window. Here's the syntax:

uses ObjectPAL
  [*methodName* **(** [**var** | **const**] *argList* **)** [*returnType*]]*
endUses

The keyword **ObjectPAL** is required to indicate that you're calling methods from an ObjectPAL library or a form, rather than from a DLL.

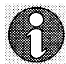


You must open a library before calling a method from it; you must open and run a form before calling a method from it.

Next, *methodName* represents the name of the method to call, and *argList* represents a comma-separated list of argument/data type pairs, optionally preceded by the keywords **var** and **const**, as appropriate.

The optional argument *returnType* specifies the data type of the value (if any) returned by the method.

Within a single Uses window, you can declare more than one library method, and methods from more than one library.



Arguments and data types declared in the Uses window must be declared exactly as they are in the library.

The following example declares two library methods, **buildMenu** and **calcInterest**. **buildMenu** takes one argument, a String constant named *pageName*. **calcInterest** takes two arguments: a Number

variable named *intRate* (passed by reference) and a SmallInt variable named *nPeriods* (passed by value).

```
uses ObjectPAL
   buildMenu(const pageName String)
   calcInterest(var intRate Number, nPeriods SmallInt) Number
endUses
```

For information about passing arguments, refer to Chapter 5 in the *ObjectPAL Developer's Guide.*

The following code, attached to a button's Uses window, declares the **calcInterest** method so the button can use it. Notice that you don't have to declare every method in a library, just the ones you want to use.

```
uses ObjectPAL
      calcInterest(var intRate Number, nPeriods SmallInt) Number
endUses
```

The following code, attached to a button's built-in **pushButton** method, opens the library and calls these methods.

```
method pushButton(var eventInfo Event)
   var
      mathLib Library
      intRate Number
      nPeriods SmallInt
      interest Number
   endVar

   if mathiLib.open("mathlib.lsl") then
      intRate = mortgage.intRate.value
      nPeriods = mortgage.nYears.value * 12
      interest = mathLib.calcInterest(intRate, nPeriods)
      interest.view("Interest")
   endIf
endmethod
```

In this example, dot notation specifies where to find the **calcInterest** method. The following statement says to look in the library represented by the Library variable *mastLib.*

```
interest = mathLib.calcInterest(intRate, nPeriods)
```

Calling custom methods attached to other forms

The concept for calling a custom method attached to another form is the same: use dot notation to specify the form to search for the method. The following example assumes that the Form variable *codeForm* has been previously declared, the form has been opened, and the method **getObjHelp** has been declared in an appropriate Uses window.

```
interest = codeForm.getObjHelp(self.name)
```

For more information about using ObjectPAL libraries, refer to Chapter 11 of the *ObjectPAL Developer's Guide.*

uses block for a DLL

To use routines stored in a DLL, write a **uses** block in one of the following places:

- A design object's Uses window
- A window for a built-in method
- A window for a custom method
- A window for a custom procedure

Where you write the block depends on the desired scope (availability) of the routine. (Scoping rules are described in Chapter 5 of the *ObjectPAL Developer's Guide*.) No matter where you write it, the basic structure (shown in the following pseudocode) is the same:

uses ***libraryName***
   routineName* (*parameterList*) *returnType
endUses

The argument *libraryName* specifies the DLL file name, where *libraryName* is a valid DOS filename of up to eight characters. Paradox assumes a file-name extension of .DLL or .EXE. Windows searches for the file in this order:

1. The current directory.
2. The Windows directory (the directory containing WIN.COM). You can use the FileSystem procedure **windowsDir** to get this information (typically, it's C:\WINDOWS).
3. The Windows system directory (the directory containing such system files as KERNEL.EXE). You can use the FileSystem procedure **windowsSystemDir** to get this information (typically, it's C:\WINDOWS\SYSTEM).
4. The directories listed in the PATH environment variable. Refer to your DOS documentation for more information.
5. The list of directories mapped in a network.

Note to advanced Windows programmers

If you're calling a routine from a previously loaded DLL (for example, a DLL loaded automatically by Windows), you can use *libraryName* to specify the DLL's module name instead of the file name. Consult your programming language's documentation for more information about DLL module names.

A **uses** block can contain one or more *routineNames*, and each *routineName* can have its own *parameterList*. A *parameterList* specifies one or more argument names and data types. If the routine returns a value, the *returnType* specifies the return value's data type. ObjectPAL checks your specifications for these arguments for *exact* matches with those declared in the routine; that's all the checking it does.

Declare a **uses** block in an object's Uses window, and within that window, declare one **uses** block for each library or DLL you want to

use. You don't have to declare every routine the library or DLL contains, just the ones you want to use. Once declared, routines are available to all methods attached to that object, and to all objects that object contains.

In a **uses** block, declare data types using the following keywords:

Data type	ObjectPAL	C	Pascal
16-bit integer	CWORD	int	Integer
32-bit integer	CLONG	long	Longint
64-bit floating-point number	CDOUBLE	double	Double
80-bit floating-point number	CLONGDOUBLE	long double	Extended
pointer	CPTR	char far *	String
binary or graphic data	CHANDLE	Handle (Windows)	THandle (Windows)

These keywords are valid only within a **uses** block. Don't use them anywhere else.

To use a routine in a method, declare variables to use as arguments, then call the routine. For example,

```
; this goes in an object's Uses window
uses myStuff ; reads routines from MYSTUFF.DLL
   doSomething(thisNum CLONG, thatNum CLONG) CDOUBLE ; declare a routine
endUses

; this modifies an object's mouseUp method
method mouseUp(var eventInfo MouseEvent)
var
   thisNum, thatNum LongInt ; declare variables to pass to the routine
   myResult Number
endVar

thisNum = 3,155,111
thatNum = 5,535,345
myResult = doSomething(thisNum, thatNum) ; call the routine, return a result
```

In the previous example, arguments in the **uses...endUses** block are declared using CLONG and CDOUBLE, and variables in the method were declared using LongInt and Number. That's because ObjectPAL data types are more complex (and powerful) than corresponding data types in C or Pascal:

- CWORD corresponds to SmallInt.
- CLONG corresponds to LongInt.
- CDOUBLE and CLONGDOUBLE correspond to Number.
- CPTR corresponds to String.
- CHANDLE corresponds to Binary and Graphic.

Note Do not modify the contents of a passed CPTR.

Example

This example uses routines from MINMAX.DLL, written using Borland's Turbo Pascal for Windows. The Pascal code for the DLL is:

Pascal code to define a DLL

```
{ this is the Pascal code that defines the DLL }
library MinMax;

function Min(X, Y: Integer): Integer; export;
begin
   if X < Y then Min := X else Min := Y;
end;

function Max(X, Y: Integer): Integer; export;
begin
   if X > Y then Max := X else Max := Y;
end;

exports
   Min index 1,
   Max index 2;

begin
end.
```

Following is the ObjectPAL code to use the routines in the DLL. First is the code for the Uses window, then comes code that modifies a button's **pushButton** method:

ObjectPAL code in the Uses window

```
; the following goes in a button's Uses window
 uses MinMax ; load routines from MINMAX.DLL
   Min (x CWORD, y CWORD) CWORD ; declare the routines to use
   Max (x CWORD, y CWORD) CWORD
endUses
```

ObjectPAL code in the method window

The following code modifies a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
   var
      x, y, z SmallInt
   endVar
   x = 2
   y = 6
   z = Min(x, y)          ; call Min from the DLL
   msgInfo("Min", z)
   z = Max(x, y)          ; call Max from the DLL
   msgInfo("Max", z)
endMethod
```

More about calling C routines

This section presents more information about calling C routines from ObjectPAL. It's organized as follows:

- Passing by value to a C procedure
- Passing by pointer to a C procedure
- Returning values
- Notes on using graphic and binary data
- Using C++

- The .DEF file

Windows uses the same calling convention used by Pascal. The Pascal calling convention entails the following:

- Parameters are pushed onto the stack in the order in which they appear in the function call.
- The code that restores the stack is part of ther called function (rather than the calling function).

The Pascal calling convention differs from the calling convention used in C. In C, parameters are pushed onto the stack in reverse order, and the calling function is responsible for restoring the stack. When writing a DLL in a language that does not ordinarily use the Pascal calling convention, such as C, you must ensure that the Pascal calling convention is used for any function that is called by Windows. In C, this requires the use of the **PASCAL** keyword when the function is declared.

The examples in the following tables assume that these ObjectPAL variables have been declared:

si SmallInt
li LongInt
nu Number
gr Graphic
st String

When passing a value to a C procedure, the ObjectPAL variable must be declared and typed explicitly. However, AnyType is not allowed.

Passing by value

Table 3-1 presents the syntaxes for passing various data types by value to a C procedure. ObjectPAL passes and returns floating point values by value, as required by the Borland C++ compiler. Other C compilers may have different requirements. To ensure compatibility, pass values by pointer, as described in the next section.

Table 3-1 Passing by value

C data type	C syntax	In USES block	ObjectPAL call
Long double	void pascal far _loadds cproc(long double value)	cproc(value CLONGDOUBLE)	cproc(si) cproc(li) cproc(nu)
Double	void pascal far _loadds cproc(double value)	cproc(value CDOUBLE)	cproc(si) cproc(li) cproc(nu)
Long int	void pascal far _loadds cproc(long int value)	cproc(value CLONGINT)	cproc(si) cproc(li)
Int	void pascal far _loadd cproc(int value)	cproc(value CWORD)	cproc(si)
String	void pascal far _loadds cproc(char * value)	cproc(value CPTR)	cproc(st)
Graphic	void pascal far _loadds cproc(HANDLE value)	cproc(value CHANDLE)	cproc(gr)
Binary	void pascal far _loadds cproc(HANDLE value)	cproc(value CHANDLE)	cproc(gr)

Passing by pointer

This section explains how to pass information if the C procedure takes pointers to information. The pointer points directly to the corresponding value in the ObjectPAL variable. For example, if you want an int* and you pass a SmallInt, you will get a pointer that points directly to the int inside the SmallInt variable. You will be able to modify the value of the SmallInt. This could be extremely dangerous for novice C programmers, since you can corrupt ObjectPAL by overwriting memory (writing past the bounds of the memory pointer). Use pointers to

- Change the information (this should be done by function return values if possible).
- Pass floating-point values to C procedures that were not compiled using the Borland C compiler. Different C compilers use different conventions for passing and returning floating-point values (double and long double). The only way to pass compiler-independent information is by pointer.

Table 3-2 presents the syntaxes for passing various data types by pointer to a C procedure.

Table 3-2 Passing by pointer

C data type	C syntax	In USES block	ObjectPAL call
LONG DOUBLE *	void pascal far _loadds cproc(long double * value)	cproc(value CPTR)	cproc(nu)
LONG INT *	void pascal far _loadds cproc(long int * value)	cproc(value CPTR)	cproc(li)
INT *	void pascal far _loadds cproc(int * value)	cproc(value CPTR)	cproc(si)
STRING *	void pascal far _loadds cproc(char * value)	cproc(value CPTR)	cproc(st)

Table 3-3 presents the syntaxes for returning values of various data types from a C procedure.

Table 3-3 Returning values

C data type	C syntax	In USES block	ObjectPAL call
Long double	long double pascal far _loadds cproc(void)	cproc() CLONGDOUBLE	nu = cproc()
Double	double pascal far _loadds cproc(void)	cproc() CDOUBLE	nu = cproc()
Long int	long int pascal far _loadds cproc(void)	cproc() CLONGINT	nu = cproc() li = cproc()
Int	long int pascal far _loadds cproc(void)	cproc() CWORD	nu = cproc() li = cproc() sm = cproc()
Char	char * pascal far _loadds cproc(void)	cproc() CPTR	st = cproc()

Note on Graphic and Binary (CHANDLE)

Graphic and Binary data are passed via CHANDLE. In C terms this is a HANDLE typedef. A CHANDLE is a handle to Windows memory. To use it, include:

```
void pascal far _loadds cproc(HANDLE value)
{
   huge * ptr = (huge *) GlobalLock(value);
   // .... use ptr.
   // ...  do not use GlobalFree(value)
   GlobalUnlock(value)
}
```

For a Binary variable, HANDLE is a handle to memory that directly contains the information contained in the binary BLOB. There is no "header" information. You can read or modify the data, but you cannot change its size.

For a Graphic variable, HANDLE is Windows Bitmap.

Use this as you would any other bitmap HANDLE.

Using C++

If you are using C++, surround your C code with the following syntax:

```
extern "C"
{
// Your procs
}
```

The .DEF file

All C functions that you want ObjectPAL to call must be exported in the .DEF file.

See also

- proc

var

Keyword

Declares variables.

Syntax

var
[*varName* [, *varName*] * *varType*]*
endVar

Description

The **var...endVar** block declares variables by associating a variable name *varName* with a data type *varType*. When you declare more than one variable of the same type on the same line, use commas to separate the names.

Note A variable's scope depends on where it is declared. For more information about using variables, including how to pass a value as a variable, see Chapter 5 in the *ObjectPAL Developer's Guide.*

Example

```
var
   myChars, xx String
   myNum Number
   orders, sales, parts TCursor
   proteus AnyType
   myBox UIObject
   a, b Array[5] SmallInt
   myOtherNum Number
endVar
```

See also

- method
- Chapter 5 in the *ObjectPAL Developer's Guide*

while

Keyword

Repeats a sequence of statements as long as a specified condition is True.

Syntax

while *Condition*
[*Statements*]
endWhile

Description

while starts by evaluating the logical expression *Condition.* If *Condition* is False, the *Statements* are not executed. If the value is True, the *Statements* between the *Condition* and **endWhile** are executed in sequence. Control then returns to the top of the loop, and the *Condition* is evaluated again. The steps are repeated until the

Condition evaluates to False, at which point the loop is exited and control advances to the next statement after **endWhile**.

You can use **loop** within the body of the **while** to force control back to the top of the **loop**, skipping the statements between **loop** and **endWhile**. You can also use **quitLoop** to jump out of the loop altogether. You can also nest **while** statements within each other to any level.

while and **for** are similar but are generally used for different reasons. Use **for** to execute a sequence of statements a known number of times. Use **while** to execute a sequence of statements an arbitrary number of times.

Example

```
; this example creates an array of last names
var
   myNames TCursor
   namesArray Array[] String
   n SmallInt
endVar

myNames.open("names.db")
namesArray.grow(1)
namesArray[1] = myNames."Last name"
n=1

while myNames.nextRec()
   n = n + 1
   namesArray.grow(1)
   namesArray[n] = myNames."Last name"
endWhile
```

See also

- for, forEach, if, loop, quitLoop, scan

CHAPTER 4

Object type reference

This chapter presents the methods and procedures associated with each ObjectPAL object type. For example, the procedures and methods for working with tables are grouped under one heading (Table); those for working with disk files are under another heading (FileSystem); and those for working with numeric data are under another heading (Number).

Types are presented in alphabetical order, from ActionEvent to ValueEvent, and within each type, methods, procedures, and structures are listed alphabetically. When you search for a particular method or procedure, ask, "What *object* am I working with?" and turn to that section of the chapter. Then ask, "What to I want that object to *do*?" (If you need to search for a method by name, turn to the index.)

Note This chapter also describes ObjectPAL keywords associated with specific types. For example, the CREATE, INDEX, and SORT statements act on tables, so they're described in the Table section.

At the end of the book are appendixes, a glossary, and an index.

Syntax notation

Syntax statements (also called prototypes) are presented for each method and procedure. This section describes the syntax notation used throughout this reference. The prototype for each ObjectPAL method and procedure includes elements required as part of the syntax and elements presented for your information, as shown in Example 4-1.

Example 4-1 ObjectPAL prototype

sample (var ***argOne*** Type [, const ***argTwo*** Type]) Type

Required elements

The following elements are required when shown as part of the prototype:

- *Name*. The name of the method or procedure.
- *Parentheses*. Parentheses are required, even if the method or procedure takes no arguments.
- *Argument*. If an argument is shown as part of the syntax, it must be included unless it is surrounded by square brackets. An argument can be a variable, an expression, or a literal (hard-coded) value. Lists of arguments are separated by commas.

Square brackets

Square brackets indicate an optional argument. If a prototype shows an argument enclosed in square brackets, you can include that argument or not, depending on what you want to do. There is one exception to this rule: when an argument is an array, the syntax for the argument shows square brackets following the array. For instance, the following syntax indicates that the method or procedure takes a resizable array:

sample (var ***arrayName*** Array[] Type **)** Type

Informational elements

Informational elements are not part of the syntax you type in for the method or procedure; they just tell you how the method or procedure works.

- *Keywords*. Keywords (shown in lightface type) provide information about the arguments for a method or procedure.

  An argument preceded by the keyword **var** is passed by reference. An argument preceded by the keyword **const** is passed as a constant. An argument by itself, without either keyword, is passed by value.

  The keyword that follows each argument specifies its data type (for example, String, Number, Table, or Logical).

  If a method or procedure returns a value, the keyword at the end of the prototype specifies its data type. Most, but not all, ObjectPAL methods and procedures return values.

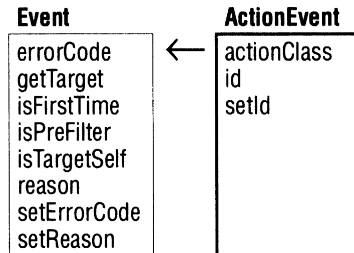
From prototype to ObjectPAL

Given the prototype presented in Example 4-1, the following statements are valid:

```
x = sample(custName)        ; one argument, variable x stores the return value

y = sample(custName, custAddress) ; two arguments, variable y stores the return
                                  ; value
```

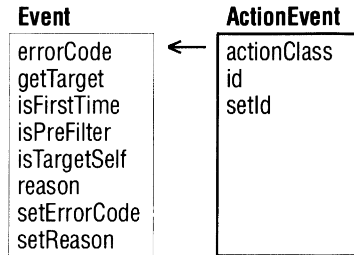
Derived methods

Many object types include methods and procedures derived from other types. For example, the ActionEvent type includes methods derived from the Event type, as follows:



This diagram shows that the ActionEvent type includes eleven methods: eight methods derived from the Event type, and three methods defined specifically for the ActionEvent type. The methods derived from the Event type are described in the Event section of the chapter, the methods defined specifically for the ActionEvent type are described in the ActionEvent section. At the beginning of each type section, diagrams show which methods (if any) are derived from other types, and which methods are defined specifically for the type.

ActionEvent



ActionEvents are generated primarily by editing and navigating in a table. The ActionEvent type includes several methods defined for the Event type.

The only built-in method that is triggered by an ActionEvent is **action**. This method, along with the rest of the built-in methods, is discussed in Chapter 2. For information about the Event model, see Chapter 6 in the *ObjectPAL Developer's Guide.*

Typically, when you work with ActionEvents, you'll also work with ObjectPAL's action constants. For example, to prevent users from editing a table, you could do something like this:

```
; thisTableFrame::action
method action(var eventInfo ActionEvent)
; if the user tries to switch to Edit mode, display a dialog box
if eventInfo.id() = DataBeginEdit then   ; DataBeginEdit is a constant
  msgStop("Stop", "You can't edit this table.")
  eventInfo.setErrorCode(1)
endif
endMethod
```

Action constants are listed in the Constants dialog box. To display the list, open an ObjectPAL Editor window and choose Language | Constants. Then, from the Types of Constants column, choose an item beginning with Action; for example, ActionDataCommands. The constants appear in the Constants column.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide.*

actionClass

ActionEvent

Method

Returns the class number of an ActionEvent.

Syntax

actionClass ( ) SmallInt

Description

Returns the class number of an ActionEvent. ObjectPAL defines constants for these class numbers; see ActionClasses in the Constants dialog box.

Example

This example uses **actionClass** to prevent the user from making any changes to a field object. This code is attached to a field's built-in **action** method. See **id** for an example that traps for the user entering Edit mode.

```
; Site_Notes::action
method action(var eventInfo ActionEvent)
; check for any attempt to edit, and block it
if eventInfo.actionClass() = EditAction then
  ; allow user to start and end field view
  if NOT (eventInfo.id() = EditEnterFieldView) AND
     NOT (eventInfo.id() = EditToggleFieldView) AND
     NOT (eventInfo.id() = EditExitFieldView) then
    eventInfo.setErrorCode(1)
    beep()
    message("Sorry. Can't make changes to this field.")
  endif
endif
endmethod
```

See also

- id, setId
- getTarget in the Event type
- Chapter 6 in the *ObjectPAL Developer's Guide* for a discussion of user-defined action constants

---

id

ActionEvent

Beginner

Method

Returns the ID number of an ActionEvent.

Syntax

id ( ) SmallInt

Description

Returns the action ID of an ActionEvent. ObjectPAL defines constants for the action ID numbers (for example, DataBeginEdit); see the categories beginning with Action (for example, ActionDataCommands) in the Constants dialog box.

You can also send user-defined actions to a built-in **action** method. For more information about creating and using user-defined constants, see Chapter 6 in the *ObjectPAL Developer's Guide*.

Example

This example uses **id** to prevent the user from entering Edit mode on a form. This code is attached to a form's built-in **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    if eventInfo.id() = DataBeginEdit then
      eventInfo.setErrorCode(1)     ; don't start Edit mode
      msgStop("Sorry", "View only - can't edit this form")
    endif
endif
endmethod
```

See also

- actionClass, setId
- Chapter 6 in the *ObjectPAL Developer's Guide* for a discussion of user-defined action constants

setId

ActionEvent

Method

Specifies the ID of an ActionEvent.

Syntax

setId (const ***actionId*** SmallInt)

Description

Specifies the ActionEvent represented by the constant *actionId*. ObjectPAL provides constants for *actionId*; see the categories beginning with Action (for example, ActionDataCommands) in the Constants dialog box.

You can also send user-defined actions to a built-in **action** method. For more information about creating and using user-defined constants, see Chapter 6 in the *ObjectPAL Developer's Guide*.

Example

In this example, the SpeedBar record-movement buttons are remapped to move within a memo field. Assume that a form contains a multi-record object, *SITES*, bound to the *Sites* table. The following code is attached to the **action** method for the *Site_Notes* field object:

```
; Site Notes::action
method action(var eventInfo ActionEvent)
var
  actID  SmallInt
endVar
; if Site_Notes is in Field View, remap record-movement
; actions to move within the memo field
if self.Editing then
  actID = eventInfo.id()
  switch
    case actID = DataPriorRecord  : eventInfo.setId(MoveBeginLine)
    case actID = DataNextRecord   : eventInfo.setID(MoveEndLine)
    case actID = DataFastBackward : eventInfo.setID(MoveBegin)
    case actID = DataFastForward  : eventInfo.setID(MoveEnd)
```

```
   case actID = DataBegin        : eventInfo.setID(FieldBackward)
   case actID = DataEnd          : eventInfo.setID(FieldForward)
 endswitch
endif
endmethod
```

See also

- id
- Chapter 6 in the *ObjectPAL Developer's Guide* for a discussion of user-defined action constants

AnyType

AnyType

AnyType
blank
dataType
isAssigned
isBlank
isFixedType
unAssign
view

An AnyType value can be any one of the data types listed in the following table.

Type	Description
AnyType	A catch-all for basic data types
Array	An indexed collection of data
Binary	Machine-readable data
Currency	Used to manipulate currency values
Date	Calendar data
DateTime	Calendar and clock data combined
DynArray	A dynamic array
Graphic	A bitmap image
Logical	True or False
LongInt	Used to represent relatively large integer values
Memo	Holds lots of text
Number	Floating-point values
OLE	A link to another application
Point	Information about a location on the screen
Record	A user-defined structure
SmallInt	Used to represent relatively small integer values
String	Letters
Time	Clock data

An AnyType can never be a complex type such as TCursor or TextStream. An AnyType variable inherits characteristics from the value assigned to it. That is, it behaves like a String when assigned a String value, like a Number when assigned a Number value, and so forth.

AnyType data objects are included in ObjectPAL so you can use variables for basic data types without declaring them first. (Remember that it's better to declare variables whenever possible.)

For more information, refer to the *ObjectPAL Developer's Guide.*

blank

AnyType

Beginner

AnyType

Method/Procedure

Returns a blank value.

Syntax

1. (Method) **blank** ( )
2. (Procedure) **blank** ( ) AnyType

Description

Generates a blank value to assign to a variable or field. A blank value is not the same as a numeric value of zero, but you can use Session type method **blankAsZero** to treat blank values as zeros in certain calculations. You can use the Session type method **isBlankZero** to find out whether Blank=Zero is on or off.

Example

This example assumes that a form has a table frame bound to the *Lineitem* table, and a button named *thisButton*. When a user clicks *thisButton*, the code scans the *Qty* field in *Lineitem*, and replaces nonblank values with blank values. This code is attached to the built-in **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

if tc.attach(LINEITEM) then                  ; attach tc to table frame
  tc.edit()                                  ; edit the table frame
  scan tc for tc.Qty.isBlank() = False :     ; look for non-blank Qty fields
    tc.Qty.blank()                           ; put a blank value in Qty
  endScan
  tc.endEdit()                               ; end edit mode
endif

endmethod
```

See also

- isBlank

dataType

AnyType

Method

Returns a string representing the data type of a variable.

Syntax

dataType ( ) String

Description

Returns a string representing the data type of a variable or expression: Binary, Currency, Date, DateTime, Graphic, Logical, LongInt, OLE, Memo, Number, Point, SmallInt, String, or Time. In

comparison statements, you need to use one of the string values shown here. For example, the following is coded incorrectly because it compares "String" with "string".

```
var s AnyType endVar
s = "This is a String data type."
msgInfo("Test", s.dataType() = "string")  ; displays False- should use "String"
```

Note This method works for all ObjectPAL types, not just AnyType.

Example

This example assumes a form has a button and a graphic field named *bmpField*. The following code loads a DynArray with several different types of data, then uses **dataType** to display the data type of each value in the DynArray. This code is attached to the button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  mixedTypes DynArray[] AnyType
endVar

mixedTypes["Make"]  = "Ford"           ; String
mixedTypes["Model"] = "Cobra"          ; String
mixedTypes["Year"]  = 1969             ; SmallInt (not Date)
mixedTypes["Color"] = Black            ; LongInt - used here as a constant
mixedTypes["Photo"] = bmpField.value   ; Graphic

foreach element in mixedTypes    ; display a message for each element

  msgInfo("dataType(" + element + ")", dataType(mixedTypes[element]))

endforeach

endmethod
```

See also

- isAssigned

isAssigned

AnyType

Method

Reports whether a variable has been assigned a value.

Syntax

isAssigned ( ) Logical

Description

Returns True if the variable has been assigned a value; otherwise, it returns False.

Note This method works for all ObjectPAL types, not just AnyType.

Example

This example uses **isAssigned** to test the value of *i* before assigning a value to it. If *i* has been assigned, this code increments *i* by one. The following code goes in a button's Var window:

```
; thisButton::var
var
  i SmallInt
endVar
```

This code is attached to the button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if i.isAssigned() then ; if i has a value
  i = i + 1            ; increment i
else
  i = 1                ; otherwise, initialize i to 1
endif
                       ; now show the value of i
message("The value of i is : " + String(i))

endmethod
```

See also

- dataType, unAssign

isBlank

AnyType

Beginner

Method

Reports whether an expression has a blank value.

Syntax

isBlank ( ) Logical

Description

Returns True if the expression has a blank value; otherwise, it returns False. Blank string values are denoted by "". Other blank values can be generated using **blank**. Note that blank values are not the same as 0, spaces ("   "), or unassigned values.

Example

The following code (attached to a button's **pushButton** method) uses **isBlank** to test various values, and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

msgInfo("Is the empty string blank?", isBlank(""))      ; True
msgInfo("Is a string of spaces blank?", isBlank("   ")) ; False
msgInfo("Is 5 a blank?", isBlank(5))                    ; False
msgInfo("Is blank blank?", isBlank(blank()))            ; True

endmethod
```

See also

- blank

isFixedType

AnyType

Method

Reports whether a variable's data type has been explicitly declared.

Syntax

isFixedType ( ) Logical

Description

Returns True if the variable has been declared using a **var...endvar** block; otherwise, it returns False.

Example

The following code demonstrates when **isFixedType** returns True. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  x SmallInt                  ; declare x
endVar

message(x.isFixedType())      ; displays True
sleep(2000)

testMe = 4                    ; testMe was not declared
message(testMe.isFixedType()) ; displays False

endmethod
```

See also

- dataType, isAssigned

unAssign

AnyType

Method

Sets a variable's state to unassigned.

Syntax

unAssign ( ) Logical

Description

Sets a variable's state to unassigned. The unassigned state is not the same as a value of 0, nor is it the same as Blank.

Example

The following example demonstrates **unAssign**. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  x AnyType
endVar
msgInfo("Is x assigned?", x.isAssigned()) ; displays False
x = 5
msgInfo("Is x assigned?", x.isAssigned()) ; displays True
x.unAssign()
```

```
msgInfo("Is x assigned?", x.isAssigned()) ; displays False
endmethod
```

See also

- blank, isAssigned

view

Beginner

AnyType

Method

Displays in a dialog box the value of a variable.

Syntax

view ([const ***title*** String])

Description

Displays, in a modal dialog box, the value of a variable. ObjectPAL execution suspends until the user closes this dialog box. You have the option to specify, in *title*, a title for the dialog box. If you don't specify a title, the variable's data type appears.

The user can change the value displayed in a **view** dialog box, as long as the data type is not an Array, DynArray, or Record. **view** cannot display Binary, Graphic, Memo, or OLE AnyTypes. The following table shows AnyTypes that can be viewed or modified.

Type	Can be viewed	Can be modified
Array	•	
Binary		
Currency	•	•
Date	•	•
DateTime	•	•
DynArray	•	
Graphic		
Logical	•	•
LongInt	•	•
Memo		
Number	•	•
OLE		
Point	•	•
Record	•	
SmallInt	•	•
String	•	•
Time	•	•

Example

This example uses **view** to display in a dialog box, the value of several variables. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] AnyType
  x AnyType
endVar

dy["Name"]      = "Jasbir Atwal"
dy["Hire Date"] = Date("5/2/84")
dy["Zip"]       = 97509
dy["Title"]     = "Engineer"

dy.view()          ; displays DynArray indexes and values

x = 5
x.view()           ; displays 5
x = "Hello"
x.view("Hi")       ; displays Hello, title is Hi
endmethod
```

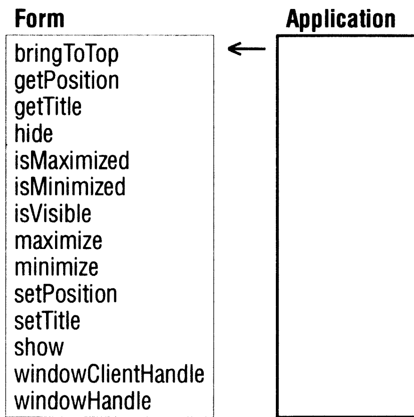
The following example uses a **view** dialog box to prompt the user for a date. If the user enters a valid date, the code displays the day of the week for that date; otherwise, an error message is displayed.

```
; showDOW::pushButton
method pushButton(var eventInfo Event)
var
  theDate AnyType
  fullDays Array[7] String
endvar

fullDays[1] = "Sunday"
fullDays[2] = "Monday"
fullDays[3] = "Tuesday"
fullDays[4] = "Wednesday"
fullDays[5] = "Thursday"
fullDays[6] = "Friday"
fullDays[7] = "Saturday"

 ; initialize theDay variable
theDate = today()
 ; now show today's date in a dialog and prompt the user to enter a new date
theDate.view("Enter a Date")

 ; it's possible the user could enter an invalid date (like "Saturday")
 ; so this try..fail block attempts to convert theDate to a Date with
 ; dateVal() and if successful, displays the day of the week that
 ; theDate falls on
try
  msgInfo("Day of the week", String(theDate) + " falls on a\n" +
             fullDays[dowOrd(dateVal(theDate))])
onfail
  msgStop("Error!", theDate + " is not a valid date.")
endtry

endmethod
```

See also

- isAssigned, unAssign

Application



An Application variable provides a handle for working with the Desktop window of the current Paradox application. You can use an Application variable in your code to control the size, position, and appearance of the Desktop. The Application type includes methods defined for the Form type.

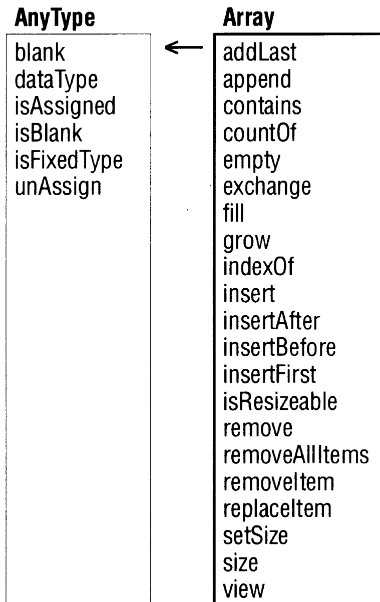
Although you can have more than one application running at the same time, Application objects can't communicate or operate on each other. An Application variable refers to the current Paradox desktop only; you can, however, use Session variables to open multiple channels to the database engine (see the Session type).

Since there can be only one current application, to get an application handle, you merely declare a variable of type Application. While an Application variable is in scope, it serves as a handle: you use that variable to access the methods in the Application type. For instance, in the following example, an Application variable called *thisApp* is declared, then used in the method's code.

```
; downSize::pushButton
method pushButton(var eventInfo Event)
var
  thisApp     Application
  x, y, w, h  LongInt
endVar
thisApp.getPosition(x, y, w, h)           ; get current position
thisApp.setPosition(x, y, w * 0.9, h * 0.9)  ; shrink desktop by 10%
endmethod
```

For more information and examples, refer to Chapter 8 in the *ObjectPAL Developer's Guide.*

Array



An Array holds values (called *items* or *elements*) in *cells* similar to the way mail slots hold mail. An ObjectPAL array is one-dimensional, like a single row of slots, where each slot holds one item. The Array type also includes methods defined for the AnyType type.

To use arrays in methods, you must declare them by specifying a name, size (number of items), and a data type for the items.

Note In ObjectPAL, array items are counted beginning with 1, not with 0, as in some other languages.

For more information and examples, refer to the *ObjectPAL Developer's Guide.*

Note ObjectPAL also supports dynamic arrays. See also methods and procedures for the DynArray type.

addLast

Array

Method Inserts an item at the end of a resizeable array.

Syntax **addLast** (const ***value*** AnyType)

Description

Inserts *value* after the last item in a resizeable array. The array grows, if necessary, to make room for the new item. If you need to add more than one element to an array, it is usually preferable to use **grow** or **setSize** to allocate more space in the array rather than several **addLast** statements. For example, the following code uses **addLast** in a **for** loop to add 10 new elements to the *ar* array. Note that this use of **addLast** forces ObjectPAL to re-allocate space in the array 10 times; once each cycle through the loop.

```
for i from 11 to 20
  ar.addLast(i * 10)
endfor
```

The following code accomplishes the same as the previous code, but executes faster because ObjectPAL allocates space only once:

```
ar.grow(10)    ; increase array size by 10 elements
for i from 11 to 20
  ar[i] = (i * 10)
endfor
```

Example

This example adds an element to a resizeable array each time *thisButton* is pressed. The **pushButton** method for *thisButton* increments the value of the newest element by 10 and displays the contents of the array in a **view** dialog box. The code immediately following goes in the Var window for *thisButton*:

```
; thisButton::Var
var
  ar Array[] SmallInt  ; declare ar as a resizeable array
  i  SmallInt          ; incrementing variable
endVar
```

The following code is attached to the built-in **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

                       ; initialize or increment i
i = iif(isAssigned(i), i + 10, 0)

if ar.size() = 0 then  ; true if this is the first time the button was pressed
  ar.setSize(0)        ; initialize size
endif

ar.addLast(i)          ; add another element to ar, and assign
                       ; the new element with the value of i

  ; display size of array in the title, and the value of
  ; each element in a view dialog box
ar.view("Size of ar array is " + strVal(ar.size()))

endmethod
```

See also

- append, insert, insertAfter, insertBefore, insertFirst

append

Array

Method

Appends the contents of one array to another.

Syntax

append (const ***newArray*** Array[] AnyType)

Description

Appends the items of *newArray* to a resizeable array. The array grows, if necessary, to make room for the added items.

Example

The following code creates two resizeable arrays, *addMe* and *baseArray*, and loads them with numeric values. This example demonstrates **append** by appending the *addMe* array to *baseArray*, then displays the results in a **view** dialog box. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   baseArray, addMe Array[] SmallInt
   i SmallInt
endVar

baseArray.setSize(3)
addMe.setSize(3)        ; now both arrays can store 3 values
for i from 1 to 3
   baseArray[i] = i     ; baseArray[1] = 1, [2] = 2, [3] = 3
   addMe[i] = (i + 3)   ;     addMe[1] = 4, [2] = 5, [3] = 6
endFor

baseArray.append(addMe) ; add the addMe array to baseArray
                        ; this grows baseArray to 6 elements

  ; now display the size of baseArray in the title of a view dialog
  ; and show baseArray elements within the dialog
baseArray.view("baseArray size: " + strVal(baseArray.size()))
endmethod
```

See also

- addLast, insert, insertAfter, insertFirst

contains

Array

Method

Searches the items of an array for a pattern of characters.

Syntax

contains (const ***value*** AnyType) Logical

Description

Returns True if any item of an array exactly matches *value*; otherwise, it returns False.

Example

This example defines and loads a resizeable array named *dogs* when a form opens. Once the form's **open** method loads the array with dog names, the code displays the contents of the array in a dialog box. A button on the form contains code that uses the **contains** method to search the array for a particular name. If **contains** doesn't find the name, the built-in **pushButton** method attached to the button uses **insertFirst** to add the name to the top of the array.

The following code is attached to the form's Var window:

```
; thisForm::Var
var
  dogs Array[] String   ; resizeable array
endVar
```

The following code is attached to the form's built-in **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    dogs.setSize(4)     ; now dogs can store 4 values
    dogs[1] = "Bruno"   ; add some dog names
    dogs[2] = "Frodo"
    dogs[3] = "Yipper"
    dogs[4] = "Juneau"

      ; show the contents of the dogs array in a view dialog
    dogs.view("dogs is initialized with these values")
endif
endmethod
```

This code is attached to the button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if dogs.contains("Bandit") = False then
  dogs.insertFirst("Bandit")  ; add new name to the top of the list
                              ; display contents of the array in a dialog
  dogs.view("dogs size: " + strVal(dogs.size()))
else                          ; "Bandit" must already exist
  msgInfo("Once is enough", "The dogs array already contains Bandit.")
endif

endmethod
```

See also

- countOf

---

countOf

Array

Method

Counts the occurrences of a value in an array.

Syntax **countOf** (const ***value*** AnyType) LongInt

Description Compares *value* to each item in an array and returns the number of exact matches, or 0 if no match is found.

Example This code (attached to a button's **pushButton** method) creates and loads a fixed array, then uses **countOf** to display the number of like values in the array:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  zoo Array[4] String
  i   SmallInt
endVar
for i from 1 to 3
  zoo[i] = "cat"       ; add three "cat" values
endFor
zoo[4] = "dog"         ; add one "dog" value

msgInfo("How many cats?", zoo.countOf("cat"))   ; displays 3
msgInfo("How many dogs?", zoo.countOf("dog"))   ; displays 1
msgInfo("How many apes?", zoo.countOf("ape"))   ; displays 0

endmethod
```

See also

- contains

empty

Array

Method Removes all items from an array.

Syntax **empty** ( )

Description Removes all items from an array. A fixed-size array stays the same size, and all items become unassigned. A resizeable array is reset to a size of 0.

Example This example shows how **empty** functions for a fixed array. The code immediately following declares a fixed array in a form's Var window. This array is global to all objects on the form.

```
; thisForm::Var
Var
  ar Array[5] AnyType  ; declare a fixed array
endVar
```

The following code is attached to a button's **pushButton** method. When this button (*fillButton*) is pressed, the code assigns numeric values to each element in the *ar* array:

```
; fillButton::pushButton
method pushButton(var eventInfo Event)
ar[1] = 234    ; load the array with numbers
ar[2] = 356
ar[3] = 98
ar[4] = 989
ar[5] = 2341
               ; view the contents of the array
ar.view("Contents of the ar array")
endmethod
```

The following code is attached to a button's **pushButton** method. When this button (*emptyButton*) is pressed, the code empties the *ar* array and displays the contents of the array. Since *ar* is a fixed array, the number of elements does not change; there are still five elements, but each value becomes unassigned.

```
; emptyButton::pushButton
method pushButton(var eventInfo Event)
ar.empty()       ; empty the ar array
                 ; view the contents of the array
ar.view("Contents of the ar array")
endmethod
```

See also

❒ remove, removeAllItems

exchange

Array

Method

Swaps the contents of two cells in an array.

Syntax

exchange (const ***index1*** LongInt, const ***index2*** LongInt)

Description

Swaps the contents of the cells at *index1* and *index2* in an array.

Example

See the example for indexOf.

See also

❒ contains, countOf, indexOf

fill

Array

Method

Fills an array with a value.

Syntax

fill (const ***value*** AnyType)

Description

Assigns *value* to every item of an array.

Example

This code creates a fixed array and fills the array with string values. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[4] String
endVar

myArray.fill("Hello")  ; fill myArray with Hello
myArray.view()         ; display four Hello's in a dialog

endmethod
```

See also

- append, insert

---

grow

Array

Method

Increases the size of a resizeable array.

Syntax

grow (const ***increment*** LongInt)

Description

Appends *increment* cells to a resizeable array, or removes cells if the value of *increment* is negative. If you try to remove more cells than the array has, an error occurs.

Example

The following example uses **grow** to increase and shrink the size of a resizeable array. This code is attached to a button's **pushButton** method.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[] SmallInt
endVar

ar.setSize(2)
ar[1] = 6
ar[2] = 123
message(ar.size()) ; displays 2
sleep(1000)
ar.grow(3)
message(ar.size()) ; displays 5
sleep(1000)
ar.grow(-3)
message(ar.size()) ; displays 2
sleep(1000)

endmethod
```

See also

- addLast, insert, insertFirst, isResizeable

indexOf

Array

Method

Returns the position of an item in an array.

Syntax

indexOf (const ***value*** AnyType) LongInt

Description

Returns the index of the first occurrence of *value* in an array, or 0 if an exact match is not found.

Example

The following example assumes a form has an undefined field object named *thisField*. When a user right-clicks on the field, a pop-up menu appears, offering a list of payment types. The item selected is inserted into the field. When the user next right-clicks on the field, the last menu item selected is the first in the list of menu choices. The following code goes in the Var window for *thisField*:

```
; thisField::Var
Var
  payArray  Array[5] String
  payMenu   PopUpMenu
endVar
```

The following code is attached to the **open** method for *thisField*. When the field first opens, this code assigns values to the array that is used for the pop-up menu:

```
; thisField::open
method open(var eventInfo Event)
payArray[1] = "Check"      ; initialize array elements
payArray[2] = "Cash"
payArray[3] = "Visa"
payArray[4] = "MasterCard"
payArray[5] = "AmEx"
endmethod
```

The following code is attached to the **mouseRightUp** method for *thisField*. This code displays the pop-up menu and inserts the selection into *thisField*. The **indexOf** method is used here to get the ordinal value of the selected menu item; the selection is then moved, with the **exchange** method, to the beginning of the array.

```
; thisField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  choiceIndex SmallInt
  choice      String
endVar

disableDefault                ; don't display the normal menu
payMenu.addArray(payArray)    ; add the array to the pop-up menu
choice = payMenu.show()       ; show the menu- assign selection to choice
self.value = choice           ; enter menu selection into field

  ; now prepare the pop-up menu for the next right click
payMenu.empty()                         ; empty the menu
```

```
choiceIndex = payArray.indexOf(choice) ; get the array index of the selection
payArray.exchange(choiceIndex, 1)      ; move the selection to the top
endmethod
```

See also

- contains, countOf

insert

Array

Method

Inserts one or more empty cells into an array.

Syntax

insert (const ***Index*** LongInt [, const ***numberOfItems*** LongInt])

Description

Inserts *numberOfItems* empty cells into a resizeable array. If *numberOfItems* is not specified, one cell is inserted. Indexes of subsequent items are increased by the number of inserted cells.

Example

The following example inserts empty elements at two locations in a resizeable array and displays the results. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[] SmallInt
endVar
myArray.setSize(20)    ; allocates space for 20 items
myArray.fill(1)        ; fills the array with 1's
myArray.insert(5)      ; inserts an empty cell at position 5
myArray.insert(12, 4) ; inserts 4 empty cells at position 12
myArray.view()
endmethod
```

See also

- insertAfter, insertBefore

insertAfter

Array

Method

Inserts an item into an array after a specified item.

Syntax

insertAfter (const ***keyItem*** AnyType, const ***insertedItem*** AnyType)

Description

Inserts *insertedItem* into a resizeable array at a position one greater than the first occurrence of *keyItem*. If *keyItem* is not found, *insertedItem* is not inserted, and indexes do not change. If *insertedItem* is inserted, indexes of subsequent items increase by 1.

Example

This example loads a resizeable array, then uses **insertAfter** to insert a new element after an existing array element. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  zoo Array[] String
endVar
zoo.setSize(0)
zoo.addLast("ape")      ; [1] = "ape"
zoo.addLast("cow")      ; [2] = "cow"
zoo.addLast("dog")      ; [3] = "dog"

zoo.insertAfter("ape", "bear")
  ; displays size: 4 in the title; zoo[ape, bear, cow, dog]
zoo.view("zoo size: " + strVal(zoo.size()))

endmethod
```

See also

❒ insert, insertBefore

Array

insertBefore

Array

Method

Inserts an item into an array before a specified item.

Syntax

insertBefore **(** const ***keyItem*** AnyType, const ***insertedItem*** AnyType **)**

Description

Searches a resizeable array for *keyItem*, and inserts *insertedItem* at *keyItem*'s position. Indexes of *keyItem* (and subsequent items) are increased by 1. If *keyItem* is not found, *insertedItem* is not inserted, and indexes do not change.

Example

This example adds an element to a resizeable array with **insertBefore**. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  foodChain Array[] String
endVar

foodChain.grow(3)          ; start array out with 3 elements
foodChain[1] = "Hawk"
foodChain[2] = "Snake"
foodChain[3] = "Fly"

  ; insert an element- this increases the array to 4 elements
foodChain.insertBefore("Fly", "Frog")
  ; displays size: 4 in title; [Hawk, Snake, Frog, Fly]
foodChain.view("foodChain size: " + strVal(foodChain.size()))

endmethod
```

See also

❐ insert, insertAfter

insertFirst

Array

Method

Inserts an item at the beginning of an array.

Syntax

insertFirst (const ***value*** AnyType)

Description

Inserts value at the beginning of a resizeable array. Indexes of subsequent items are increased by 1.

Example

This example creates a resizeable array, then adds a new element to the beginning of the array. This code is attached to a button's built-in **pushButton** method:

```
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar
myZoo.setSize(2)   ; start the array with two elements
myZoo[1] = "lion"
myZoo[2] = "tiger"

                   ; insert an element at beginning of array-
                   ; this increases the array to three elements
myZoo.insertFirst("bear")
                   ; displays size: 3 in title; [bear, lion, tiger]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endmethod
```

See also

❐ addLast, append, insert, insertAfter, insertBefore

isResizeable

Array

Method

Reports whether an array can be resized.

Syntax

isResizeable ( ) Logical

Description

Returns True if an array can be resized; otherwise, it returns False.

Example

This code checks to see if a particular array can be resized before attempting to increase its size. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myArray Array[] String
endVar
if myArray.isResizeable() = True then  ; if array can be resized
  myArray.grow(5)                       ; add 5 cells to it
else
  msgStop("Problem", "Array cannot be resized.")
endif
endmethod
```

See also

- grow, size

remove

Array

Method

Removes one or more items from an array.

Syntax

remove (const ***index*** SmallInt
[, const ***numberOfItems*** SmallInt])

Description

Deletes *numberOfItems* items (or 1 item, if *numberOfItems* is not specified) at *index* in an array. Indexes of subsequent items are decreased by *numberOfItems* (or 1, if *numberOfItems* is not specified).

Example

This example removes a single item from a resizeable array. Note that it is common to use the **indexOf** method to determine which element you want to remove. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar

myZoo.setSize(3)          ; start myZoo out with three elements
myZoo[1] = "lion"
myZoo[2] = "tiger"
myZoo[3] = "bear"

myZoo.remove(myZoo.indexOf("tiger"))  ; same as myZoo.remove(2)

                          ; title displays size: 2
                          ; dialog diasplays myZoo[lion, bear]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endmethod
```

The following example shows how to use **remove** to eliminate more than one element from a resizeable array. This code is attached to a button's **pushButton** method:

```
; thatButton::pushButton
method pushButton(var eventInfo Event)
var
  myNums Array[] SmallInt
  i      SmallInt
endVar

myNums.grow(9)      ; start myNums with nine elements
for i from 1 to 9   ; assign nine elements
   myNums[i] = i
endFor

                    ; displays myNums[1, 2, 3, 4, 5, 6, 7, 8, 9]
myNums.view("Before removing elements")
                    ; remove four items, starting with third element
myNums.remove(3, 4) ; myNums = [1, 2, 7, 8, 9]
                    ; displays myNums[1, 2, 7, 8, 9]
myNums.view("After removing elements")
endmethod
```

See also

- insert, removeAllItems, removeItem

---

removeAllItems

Array

Method

Removes all occurrences of an array item.

Syntax

removeAllItems (const ***value*** AnyType)

Description

Deletes all occurrences of *value* from an array. Indexes of subsequent items are decreased by 1.

Example

This example shows how **removeAllItems** works with a resizeable array. The following code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   myZoo Array[] String
endVar
myZoo.setSize(5)
myZoo[1] = "ape"
myZoo[2] = "cow"
myZoo[3] = "pig"
myZoo[4] = "cow"
myZoo[5] = "lion"

   ; display current contents of array in a dialog
myZoo.view("Before removing elements")

   ; removes all occurrences of cow
myZoo.removeAllItems("cow")

   ; now,
   ; myZoo[1] = "ape"
```

```
  ; myZoo[2] = "pig"
  ; myZoo[3] = "lion"

  ; display new contents of array in a dialog
myZoo.view("After removing elements")

endmethod
```

See also

❐ remove, removeItem

removeItem

Array

Method

Deletes a specified item from an array.

Syntax

removeItem (const ***value*** AnyType)

Description

Deletes the first occurrence of *value* from an array. Indexes of subsequent items are decreased by 1.

Example

This example uses **removeItem** to eliminate an item from a resizeable array. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myZoo Array[] String
endVar

myZoo.setSize(4)
myZoo[1] = "ape"
myZoo[2] = "lion"
myZoo[3] = "tiger"
myZoo[4] = "lion"

  ; this displays [ape, lion, tiger, lion]
myZoo.view("Before removing a lion")

  ; remove first occurrence of "lion"
myZoo.removeItem("lion")

  ; this displays [ape, tiger, lion] in a dialog
myZoo.view("After removing a lion")

endmethod
```

See also

❐ remove, removeAllItems, replaceItem

replaceItem

Array

Method

Overwrites an item in an array with another item.

Syntax **replaceItem** (const ***keyItem*** AnyType, const ***newItem*** AnyType)

Description Searches an array for *keyItem*, and replaces the first occurrence of *keyItem* with *newItem*.

Example This example replaces an item in a resizeable array, and displays the initial value and the results in a dialog box. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  foodChain Array[] String
endVar

foodChain.setSize(3)
foodChain[1] = "Shark"
foodChain[2] = "Elephant"
foodChain[3] = "Minnow"

  ; display contents of array in a dialog box
foodChain.view("Before replaceItem...")

foodChain.replaceItem("Elephant", "Tuna")
  ; display contents of array in a dialog box ([Shark, Tuna, Minnow])
foodChain.view("After replaceItem...")

endmethod
```

See also

- removeItem

---

setSize

Array

Method Specifies the size of an array.

Syntax **setSize** (const ***size*** LongInt)

Description Saves space for *size* items in a resizeable array. If **setSize** makes the array smaller, the array is truncated.

Example This example declares a resizeable array in the variable declaration section, then uses **setSize** to initialize the size of the array to three elements. The code fills each element of the array, then issues **setSize** again, this time to resize the array to two elements. The result of making the array smaller (shown in a dialog box) is the elimination of the third (and last) element. This code is attached to a button's built-in **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
```

```
var
  myArray Array[] SmallInt
endVar

myArray.setSize(3)          ; size is 3

myArray[1] = 123
myArray[2] = 2353
myArray[3] = 18

  ; display size: 3 in title; [123, 2353, 18] in a dialog box
myArray.view("myArray size: " + strVal(myArray.size()))

myArray.setSize(2)          ; size is 2- myArray[3] truncated

  ; display size: 2 in title; [123, 2353] in a dialog box
myArray.view("Now myArray size: " + strVal(myArray.size()))

endmethod
```


See also ❐ grow, isResizeable

size

Array

Method Returns the number of items in an array.

Syntax **size** ( ) LongInt

Description Returns the total number of items in an array, even if one or more elements are blank.

Example See the example for setSize.

See also ❐ grow, setSize

view

Array

Method Displays in a dialog box the contents of an array.

Syntax **view** ([const ***title*** String])

Description Displays in a modal dialog box the contents of an array. ObjectPAL execution suspends until the user closes this dialog box. You have the option to specify, in *title*, a title for the dialog box. If you omit title, the title is "Array."

Unlike many other data types, Array values displayed in a **view** dialog can not be changed interactively. See "AnyType" earlier in this chapter for more information regarding other data types and the **view** method.

Example

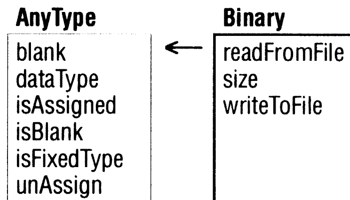
This example displays the contents of an array in a dialog box without a custom title, then with a custom title. Note that *title* can be any expression that evaluates to a string. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[]  SmallInt
  i           SmallInt
endVar

ar.setSize(10)
for i from 1 to 10
  ar[i] = i * 10
endfor

ar.view()        ; displays 10, 20, 30, etc (no title)
                 ; this displays "ar size: 10" in the title
ar.view("ar size: " + strVal(ar.size()))

endmethod
```

See also

- contains

Binary



A binary object (sometimes called a binary large object, or BLOB) contains data that only a computer can read and interpret. An example of a binary object is a sound file: a human can't read or interpret the file in its raw form, but a computer can.

When you declare a Binary variable, you create a handle to a binary object, a variable you can refer to in your code to move binary data back and forth between a disk file and a table, or from a disk file or a table to a method or procedure.

The Binary type also includes methods defined for the AnyType type.

readFromFile

Binary

Method

Reads data from a file and stores it in a Binary variable.

Syntax

readFromFile (const ***fileName*** String) Logical

Description

Reads binary data from the disk file named in *fileName*. This method returns True if successful; otherwise, it returns False.

Example

The following statements declare a Binary variable *theSound*, read binary data from a file into *theSound*, then assign the value of the variable to a Binary field in a table. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.)

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  soundsTC TCursor
  theSound Binary
endVar
if theSound.readFromFile("noise.bin") then ; True if readFromFile succeeds
  if soundsTC.open("sounds.db") then
     soundsTC.edit()
     soundsTC.insertRecord()
     soundsTC.SoundName = "Noise"
     soundsTC.SoundData = theSound  ; put file contents in a binary field
```

```
    soundsTC.endEdit()
    soundsTC.close()
  endIf
endIf

endmethod
```

See also

- size, writeToFile
- Methods and procedures defined for the FileSystem type

size — Binary

Method

Returns the number of bytes in a Binary variable.

Syntax

size ( ) LongInt

Description

Returns a value representing the number of bytes stored in a Binary variable.

Example

The following example steps through the records in a table that contain Binary fields. The example tests the size of each Binary field. If there's enough free disk space, the code writes the data to a disk file. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.) This code is attached to a custom method named *writeBinFiles*:

```
method writeBinFiles()
var
  binVar    Binary
  fs        FileSystem
  soundsTC  TCursor
  freeSpace LongInt
endVar

if soundsTC.open("Sounds.db") then
  scan soundsTC for not isBlank(soundsTC.SoundData) :
    binVar = soundsTC.SoundData               ; binVar = SoundData field value
    freeSpace = fs.freeDiskSpace("B")
    if freeSpace > binVar.size() then          ; if there's room on B:
      binVar.writeToFile(soundsTC.SoundName)    ; write binVar to file
    else                                      ; else the file won't fit on B:
      msgStop("Stop", "The disk in drive B: is full.")
      return
    endIf
  endScan
endif

endmethod
```

See also

- readFromFile, writeToFile
- Methods and procedures defined for the FileSystem type

writeToFile

Binary

Method

Writes the data stored in a Binary variable to a disk file.

Syntax

writeToFile (const ***fileName*** String) Logical

Description

Writes the data stored in a Binary variable to the disk file specified in *fileName*. This method returns True if successful; otherwise, it returns False.

Example

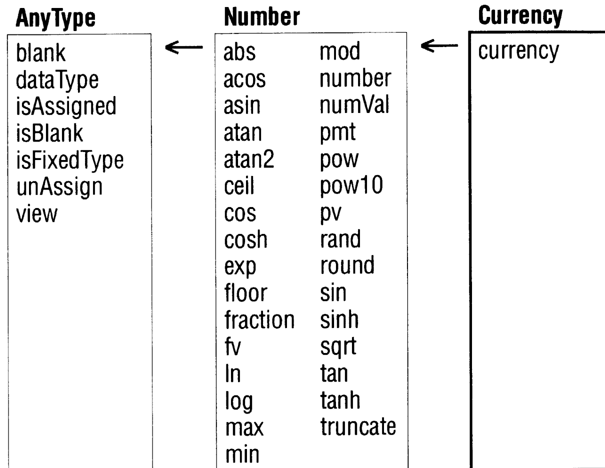
The following example steps through the records in a table that contains Binary fields. It tests the size of each Binary field. If there's enough free disk space, the code writes the data to a disk file. (Assume SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, B.) This code is attached to a custom method named *writeBinFiles*:

```
method writeBinFiles()
var
  binVar    Binary
  fs        FileSystem
  soundsTC  TCursor
  freeSpace LongInt
endVar

if soundsTC.open("Sounds.db") then
  scan soundsTC for not isBlank(soundsTC.SoundData) :
    binVar = soundsTC.SoundData                     ; binVar = SoundData field
value
    freeSpace = fs.freeDiskSpace("B")
    if freeSpace > binVar.size() then                ; if there's room on B:
      binVar.writeToFile(soundsTC.SoundName)        ; write binVar to file
    else                                            ; else the file won't fit
                                                    ; on B:
      msgStop("Stop", "The disk in drive B: is full.")
      return
    endIf
  endScan
endif

endmethod
```

See also

- readFromFile, size
- Methods and procedures defined for the FileSystem type

Currency



Currency values can range from ± $3.4 * 10^{-4930}$ to ± $1.1 * 10^{4930}$ precise to six decimal places. The number of decimal places displayed depends on the user's Control Panel settings. However, the value stored in a table does not—a table stores the full six decimal places. The Currency type also includes methods defined for the AnyType type and the Number type.

currency

currency

Method

Casts a value as Currency.

Syntax

currency (const ***value*** AnyType) Currency

Description

Casts (converts) the data type of *value* to Currency.

Example

In this example, a number is stored to a String variable, then cast to a Currency type for use in a calculation. The **pushButton** method for *showDouble* displays the type of the variable, then calculates and displays the result of the string cast as Currency and multiplied by 2:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)

var
  numstr   String
endVar
```

```
numStr = "12.34"
msgInfo("The data type of numStr is:", dataType(numStr))
; before multiplying numStr by two, it must be cast
; to a numeric type
msgInfo("Double " + numStr, currency(numStr) * 2)
endmethod
```

In the next example, the **pushButton** method for the *watchPrecision* button calculates a number using variables of type Number, then performs the same calculation with the values cast to Currency. The result of the two calculations varies slightly.

```
; watchPrecision::pushButton
method pushButton(var eventInfo Event)

var
  x, y, z Number
endVar

x = 1.2 / 3.323         ; stores greatest precision
y = 4.9 / 7.3
z = 2.0 * x * y         ; calculates on full values
msgInfo("Result of Number calculation",
         format("W14.6", z))     ; displays .484790
x = currency(1.2 / 3.323)        ; stores precision to 6th decimal place
y = currency(4.9 / 7.3)
z = 2.0 * x * y                  ; calculates on 6 decimal precision values
msgInfo("Result of Currency calculation",
         format("W14.6", z))     ; displays .484791

endmethod
```

See also

- Chapter 9 of the *ObjectPAL Developer's Guide* for information about the Currency type

Database

Database

close
delete
executeQBE
executeQBEFile
executeQBEString
isAssigned
isTable
open
writeQBE

A Database variable provides a handle to a database (a directory). When you start a Paradox application, Paradox opens the *default database* (the working directory). The default database stores the path to the current working directory. If all you want to do is work with those tables, you don't have to open any other database. To work with tables stored elsewhere, declare a Database variable and use an **open** statement to create a handle to another database. (You could specify the full path to each table each time you wanted to use it, but code that uses Database variables is easier to maintain.)

Using **open** and an alias, you can specify which database to open, as shown in the following example:

```
var
   custInfo Database
endVar
; addAlias is defined for the Session type
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo") ; opens the CustomerInfo database
                              ; CustomerInfo must be a valid alias
```

Paradox now knows about two databases: the default database and CustomerInfo. The variable *custInfo* is a *handle* to the CustomerInfo database—that is, you can use *custInfo* in statements to refer to the CustomerInfo database. For example, suppose you have two files named ORDERS.DB (one in your working directory, and one in CustomerInfo), and you want to find out if these files are tables. The following example tests ORDERS.DB in the working directory first, then uses *custInfo* as a handle for the CustomerInfo database and tests ORDER.DB there:

```
var
   custInfo Database
endVar
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")
custInfo.open("CustomerInfo")

if isTable("orders.db") then        ; test ORDERS.DB in the default database
   msgInfo("Working directory", "ORDERS.DB is a table.")
endIf
```

```
if custInfo.isTable("orders.db") then   ; use myDB as a handle for
                                        ; the CustomerInfo database
   msgInfo("CustomerInfo", "ORDERS.DB is a table.")
endIf
```

If you use **open** but don't specify a database, Paradox assumes you want a handle for the default database. For example, this syntax gives you a handle for the default database, which you could pass to a custom method that requires a database handle.

```
var defaultDb Database endVar
defaultDb.open() ; opens the default database
```

Using a handle to the default database can also make code more readable, especially when you're working with several databases at once.

For more information and examples, refer to Chapter 10 in the *ObjectPAL Developer's Guide*.

close

Database

Method

Closes a database.

Syntax

close ( ) Logical

Description

Ends the association between a Database variable and a database, making the variable unassigned. **close** returns True if it succeeds; otherwise, it returns False.

Example

The following code opens the database with the alias *someTables*. If the *Orders* table doesn't exist in *someTables*, this code closes *someTables* and opens another database with the alias *moreTables*. This code assumes that both aliases have been defined elsewhere and are valid.

```
; sumButton::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tc TCursor
endVar
db.open("someTables")               ; open the database alias someTables
if db.isTable("Orders.db") then     ; if Orders.db is in the database,
  tc.open("Orders.db", db)          ; open a TCursor for it
                                    ; calculate the total balance due
  msgInfo("Balance Due", tc.cSum("Balance Due"))
else
  db.close()                        ; close someTables database
  db.open("moreTables")             ; and open another one
  if db.isTable("Orders.db") then
    tc.open("Orders.db", db)
    msgInfo("Balance Due", tc.cSum("Balance Due"))
  endIf
```

```
endIf
endmethod
```

See also

- open

delete

Database

Method/Procedure

Deletes a table from a database.

Syntax

1. **delete** (const ***tableName*** String [, const ***tableType*** String]) Logical
2. **delete** (const ***tableVar*** Table) Logical

Description

Removes a table and any associated index files or table view files from the database without asking for confirmation. If you use syntax 1, and if the file extension is not standard or not supplied, you can use the optional argument *tableType* to specify the type of the table to delete ("Paradox" or "dBASE"). If *tableType* is not specified or not standard, "Paradox" is assumed. If you use syntax 2, you can use the argument *tableVar* to specify a Table variable. However, this method uses only the name and type of the table described by the Table variable, not its database association.

The operation cannot be undone. This method returns True if the table is successfully deleted; otherwise, it returns False. If the table is open, **delete** fails.

Example

In this example, the **pushButton** method for *delTable* deletes a table from the database with the alias *megaData.*

```
; delTable::pushButton
method pushButton(var eventInfo Event)
var
  myDb Database
  tableName String
endVar
tableName = "OldTable"
myDb.open("megadata")
if isTable(tableName) then
  myDb.delete(tableName, "dBASE") ; removes OldTable.dbf from megadata
endif
endmethod
```

See also

- isTable

executeQBE

Database

Method/Procedure

Executes a QBE query.

Syntax

1. **executeQBE** (const ***qbeVar*** Query) Logical
2. **executeQBE** (const ***qbeVar*** Query, const ***ansTbl*** String) Logical
3. **executeQBE** (const ***qbeVar*** Query, const ***ansTbl*** Table) Logical
4. **executeQBE** (const ***qbeVar*** Query, var ***ansTbl*** TCursor) Logical

Description

Executes a query created in an ObjectPAL method and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system memory only; a table is not created on disk.

If **executeQBE** is successful—if *ansTbl* or ANSWER.DB is created—this method returns True (even if the resulting table is empty); otherwise it returns False.

A query in a method begins with a Query variable, the = sign, and the keyword **query** followed by a blank line. Next comes the body of the query, and another blank line. The query ends with the keyword **endquery**. Double backslashes are not required to specify a path.

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this method with **executeQBEString**). You can use absolute paths or aliases in the query definition.

Example

The code in this example modifies the built-in **pushButton** method for *findName*. When the button is pushed, the method defines a Query variable, then uses **executeQBE** to execute the query and store the results in the CUSTNAME.DB table.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  qq query
  cName String
  tv TableView
endVar

cName = "Unisco"
qq = query
     c:\pdoxwin\sample\customer | Customer No | Name         |
                                | Check       | Check ~cName |
```

```
     endquery
executeQBE(qq, "CustName.db") ; put results into Custname.db
tv.open("CustName")           ; view the table
endmethod
```

The next example adds an alias, uses the alias to open a database, then executes a query on a table in that database:

```
; thisButton
method pushButton(var eventInfo Event)
var
  db Database
  qq query
  tc TCursor
  tv TableView
endVar

qq = query
     contacts.db | Last Name   | First Name | Company | Phone       |
                 | Check       | Check      | Check   | Check 808.. |
     endquery

; create the sampData alias then open it
addAlias("sampData", "Standard", "C:\\pdoxwin\\sample")
db.open("sampData")

; now query CONTACTS.DB in the sampData database and write
; results to PHONE.DBF in the default database
db.executeQBE(qq, ":work:phone.dbf")
tv.open("phone.dbf")     ; open the table in the default database
endmethod
```

See also

- executeQBEFile, executeQBEString

executeQBEFile

Database

Method/Procedure

Opens and executes a QBE file.

Syntax

1. **executeQBEFile** (const ***qbeFileName*** String) Logical
2. **executeQBEFile** (const ***qbeFileName*** String, const ***ansTbl*** String) Logical
3. **executeQBEFile** (const ***qbeFileName*** String, const ***ansTbl*** Table) Logical
4. **executeQBEFile** (const ***qbeFileName*** String, var ***ansTbl*** TCursor) Logical

Description

Opens *qbeFileName* (created with the **writeQBE** method or interactively using the Query Editor), executes it, and writes the results to the table specified in *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBEFile** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string;

if you don't include a file extension, *ansTbl* is a Paradox table by default (use .DBF to write to a dBASE table). In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system memory only; a table is not created on disk.

If **executeQBEFile** is successful—if *ansTbl* or ANSWER.DB is created—this method returns True (even if the resulting table is empty); otherwise it returns False.

Example

This code demonstrates how each form of the syntax works:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tb Table
  tc TCursor
endVar
addAlias("myData",  "Standard", "c:\\PdoxWin\\MyTables")
db.open("myData")
tb.attach("Custl.db", db)

; this writes results into :priv:ANSWER.DB
executeQBEFile("GetCust.qbe")

; this writes results into MyCust.db in myData database
executeQBEFile("GetCust.qbe", "MyCust.db")

; this writes results into Custl.db in myData database
executeQBEFile("GetCust.qbe", tb)

; this writes results into the tc TCursor
executeQBEFile("GetCust.qbe", tc)

endmethod
```

See also

- executeQBE, executeQBEString

---

executeQBEString

Database

Method/Procedure

Executes a QBE string.

Syntax

1. **executeQBEString** (const ***QBEString*** String) Logical
2. **executeQBEString** (const ***QBEString*** String, const ***ansTbl*** String) Logical
3. **executeQBEString** (const ***QBEString*** String, const ***ansTbl*** Table) Logical

4. executeQBEString (const ***QBEString*** String,
var ***ansTbl*** TCursor) **Logical**

Description

Executes a QBE string, and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBEString** writes to ANSWER.DB in the user's private directory. In syntax 2, you specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, the Table variable must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system memory only; a table is not created on disk.

If **executeQBE** is successful—if *ansTbl* or ANSWER.DB is created—this method returns True (even if the resulting table is empty); otherwise it returns False.

A QBE string can be a combination of quoted strings and string variables.

executeQBEString is useful when you're building a QBE string from smaller strings. Double backslashes are required when specifying a path.

Because a QBE string is a quoted string, it cannot contain tilde variables (but you can use string variables to get the same effect). If you want to use tilde variables in a query, use **executeQBE**.

Example

For this example, the **pushButton** method for *findName* defines a query as a string value, then uses **executeQBEString** to execute the query.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  qs String
  tv TableView
  tc TCursor
  stkNo String
endVar

; add the sampData alias then open the database
addAlias("sampData", "Standard", "c:\\pdoxwin\\sample")
db.open("sampData")

; open a TCursor for the Stock table
tc.open("Stock.db", db)

; if locate finds Krypton Flashlight in the Description field
if tc.locate("Description", "Krypton Flashlight") then

  ; now use the Stock No field value in Stock.db in a query string
  qs = "Query\n\n" +
       ":sampData:Lineitem | Order No | Stock No |\n" +
                          "| _ordNo |" + tc."Stock No" + "|\n\n" +
       ":sampData:Orders | Order No | Customer No |\n" +
```

```
                    "|  ordNo |  cust |\n\n" +
         ":sampData:Customer | Customer No | Name | Phone |\n" +
                      "|  cust | Check | Check |\n\n" +
         "EndQuery"

  ; note that the vertical lines (|) don't have to be aligned

  if executeQBEString(qs) then       ; writes to :PRIV:ANSWER
    tv.open(":priv:answer.db")       ; display the answer table
  else
    msgStop("Error", "Query failed") ; otherwise, query failed
  endif

else
  msgStop("Error", "Can't find Krypton Flashlight")
endif

endmethod
```

See also

- executeQBE, executeQBEFile

isAssigned

Database

Method

Reports whether a Database variable has been assigned a value.

Syntax

isAssigned ( ) Logical

Description

Returns True if the Database variable has been assigned a value; otherwise, it returns False.

Example

For this example, a form has an unassigned field named *coRating* and a button named *showRating*. Code attached to *showRating*'s **pushButton** method uses **isAssigned** to determine whether the Database variable *db* is assigned. If it's not, an alias is established and assigned to the Database variable. Once the variable is defined, the code opens a TCursor for the *NewCust* table contained in the database. The TCursor locates a value in the Company field, then displays that company's credit rating in the *coRating* field on the form. Following is the code attached to the **pushButton** method for *showRating*:

```
; showRating::pushButton
method pushButton(var eventInfo Event)
var
  db Database
  tc TCursor
endVar

if not isAssigned(db) then
  addAlias("myTables", "Standard", "c:\\pdoxwin\\myTables")
  db.open("myTables")
endif
```

```
tc.open("NewCust.dbf", db)
if tc.locatePattern("Company", "Thompson's..") then
  coRating.value = tc.Rating
else
  message("Error", "Thompson's.. not found.")
endif

endmethod
```

See also

- isTable

isTable

Database

Method/Procedure

Reports whether a table exists in a database.

Syntax

1. **isTable** (const ***tableName*** String [, const ***tableType*** String]) Logical
2. **isTable** (const ***tableVar*** Table) Logical

Description

Returns True if a specifed table is found in the database; otherwise, it returns False.

If you use syntax 1, you can specify a table name and a table type in arguments *tableName* and *tableType*. If you use syntax 2, you can specify a Table variable in *tableVar*. However, this method uses only the name and type of the table described by the Table variable, not the database association.

Example

The following code uses **isTable** to determine whether the *Orders* table exists in a given database. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  db        Database
  testMe    String
  testMeToo Table
  myTable   TableView
endVar

db.open()                          ; opens the default database
testMe = "Orders.db"
if db.isTable(testMe) then
  myTable.open(testMe)
else
  message(testMe, " is not a table!")
endIf

testMeToo.attach("sales.db")
if testMeToo.isTable() then
  tot = testMeToo.cSum("Total sales")
```

```
  msgInfo("total sales:", tot)
endIf
endmethod
```

See also

- isAssigned

open

Database

Method

Opens a database.

Syntax

1. **open** ( ) Logical
2. **open** (const ***aliasName*** String) Logical
3. **open** (const ***ses*** Session) Logical
4. **open** (const ***aliasName*** String, const ***ses*** Session) Logical

Description

Opens a database. In syntax 1, where no arguments are given, **open** opens the default database in the current session. In syntax 2, you specify in *aliasName* a database to open in the current session. Syntax 3 lets you open the default database in the session specified in *ses*. Use syntax 4 to open a specified database in a specified session.

If you use syntax 2 or syntax 4, *aliasName* must be a valid alias in the current session or the *ses* session. The colons around the alias name are optional.

Syntax 3 and 4 require that a valid session variable has been opened; the current session is assumed in syntax 1 and 2.

open returns true if it is able to open the specified database; otherwise, it returns False.

Example

For this example, the **pushButton** method for *thisButton* opens three databases in the current session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dDb, myDb, pDb Database
endVar

dDb.open()             ; associate dDb with the default database
myDb.open("custInfo") ; associate myDb with the CustInfo database
                       ; (custInfo is an alias)
pDb.open("PRIV")       ; associate pDb with the Private directory

endmethod
```

See also

- close

- methods in the Session type

writeQBE

Database

Method/Procedure

Writes a query statement or a query string to a file.

Syntax

1. **writeQBE** (const ***qbeVar*** Query, const ***fileName*** String) Logical
2. **writeQBE** (const ***str*** String, const ***fileName*** String) Logical

Description

Writes a previously defined query statement or query string to the file specified in *fileName*. If *fileName* exists, it is overwritten without asking for confirmation. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Example

For this example, assume a form has a button named *getNames*. When the form opens, this example determines whether the GETNAMES.QBE file exists in the current directory (or in the private directory). If the file does not exist, the form's built-in **open** method uses **writeQBE** to write a query string to GETNAMES.QBE. The built-in **pushButton** for *getNames* runs the query with **executeQBEFile**, then views the results in a TableView. The code immediately following is attached to the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
Var
  qs String    ; a query string
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form
else
  ;code here executes just for form itself

  if not isfile("GetDest.qbe") then    ; if the query file dosn't exist
                                       ; construct a query string

    qs = "Query\n\n" +
         ":mastApp:Dest | Destination Name | Avg Temp (F) |\n" +
                      "| Check | Check 70 |\n\n" +
         "EndQuery"

    ; write the query string to GetNames.qbe
    writeQBE(qs, "GetDest.qbe")
  endif

endif
endmethod
```

The following code is attached the built-in **pushButton** method for the *getNames* button. This code does not check whether GETNAMES.QBE exists because the form's **open** method ensures the file is available.

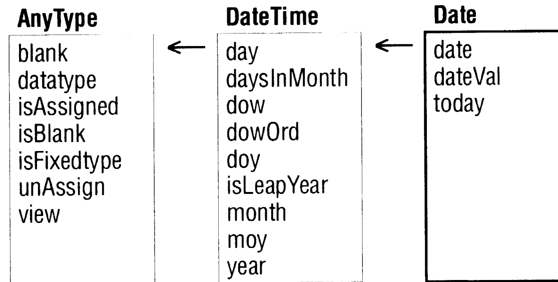
```
; getDest::pushButton
method pushButton(var eventInfo Event)
var
  tv TableView
endVar

; execute the query file and store results in MyDest.db
executeQBEFile("GetDest.qbe", "MyDest.db")
; display the table
tv.open("MyDest")

endmethod
```

Another use for this method is to use ObjectPAL to create and save a query that the user can run interactively using the Query Editor.

See also

- executeQBE, executeQBEFile, executeQBEString

Date



In ObjectPAL, date values can be represented in either month/day/year, day-Month-year, or day.month.year format. Dates must be cast (explicitly declared). For example,

```
var
  d Date
endVar
d = date("12/21/1997")
```

assigns to *d* the date December 21, 1997. Don't omit the quotes around the date value—if you do, ObjectPAL performs division on the values.

The Date type includes methods defined for the AnyType type and the DateTime type.

Date values are formatted as specified by the **formatSetDateDefault** method (System type), or by ObjectPAL formatting statements.

Although you can use ObjectPAL to perform calculations on any valid date, date values stored in a Paradox table must range from Jan. 1, 100, to Dec. 31, 9999.

Dates in the 20th century can be specified using two digits for the year, as in

```
myDay = date("11/09/59")
```

Dates in the 2nd through the 10th centuries must include three digits of the year (as in 12/17/243); dates in the 11th through 19th centuries must have four digits (12/17/1043). The year cannot be omitted completely.

The Date type includes several methods defined for the DateTime type. For more information and examples, refer to Chapter 9 in the *ObjectPAL Developer's Guide.*

date

Beginner

date

Procedure

Casts a value as a Date.

Syntax

date ([const ***value*** AnyType]) Date

Description

Casts (converts) *value* as a date. If the the date supplied in *value* is invalid, the method fails. If you do not supply *value,* **date** returns the current date as a Date data type.

Example

This example casts a string value as a date, uses the date value in a calculation, then displays the result in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  s String
  d Date
endVar

s = "11/11/99"    ; s is a String value
d = date(s) + 7   ; convert String type to a Date type

d.view()          ; show value of d in a dialog box (11/18/99)
                  ; dialog box title displays "Date"
endmethod
```

See also

- dateVal

dateVal

Date

Procedure

Returns a value as a date.

Syntax

dateVal (const ***value*** AnyType) Date

Description

Returns a value as a date.

Example

In the following example, the **pushButton** method for a button uses **dateVal** to get the date equivalent of a String value and displays the value in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  s String
  d Date
endVar
```

```
s = "11/11/99"    ; s is a String value
d = dateVal(s)    ; d holds the date equivalent of s

d.view()          ; show value of d in a dialog box (11/11/99)
                  ; dialog box title displays "Date"

endmethod
```

See also

- date

today

Date

Procedure

Returns the current date.

Syntax

today ( ) Date

Description

Returns the current date, according to the system clock/calendar.

Example

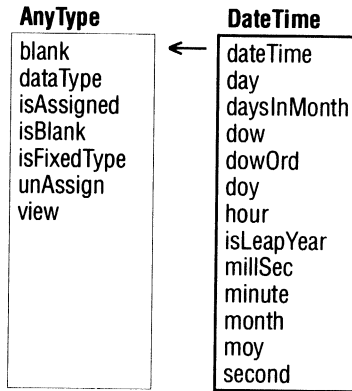
This example displays the current date in a dialog box:

```
; CurrentDate::pushButton
method pushButton(var eventInfo Event)
msgInfo("Today's Date", today())      ; displays the current date
endmethod
```

See also

- date
- day, month, year in DateTime type

DateTime



A DateTime variable stores data in the form hour-minute-second-millisecond year-month-day. DateTime values are used only in ObjectPAL calculations; you cannot store a DateTime value in a Paradox table. DateTime values must be cast (explicitly declared). For example, the following statements assign to the DateTime variable *dt* a time of 10 minutes and 40 seconds past eleven o'clock and a date of December 21, 1997.

```
var dt DateTime endVar
dt = DateTime("11:10:40 am 12/21/97")
```

The quotes around the value are required.

You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;). DateTime values are formatted as specified by the **formatSetDateTimeDefault** method (System type), or by ObjectPAL formatting statements.

You must specify a DateTime value completely; you can't omit any of the fields, but you can specify a value of zero for any field.

See also methods and procedures defined for the Date type and the Time type.

The DateTime type includes methods defined for the AnyType type. Also, both the Date and Time types include methods defined for the DateTime type.

dateTime

dateTime

Beginner

Method

Casts a value as a DateTime data type.

Syntax

dateTime ([const ***value*** AnyType]) DateTime

Description

Casts (converts) *value* as a DateTime data type. If *value* is not supplied, **dateTime** returns the current time and date as a DateTime data type.

Example

The following statements assign to the DateTime variable *dt* a time of 10 minutes and 40 seconds past eleven o'clock and a date of December 21, 1997. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
var dt DateTime endVar
dt = dateTime("11:10:40 am 12/21/97")
```

The quotes around the value are required.

You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;). DateTime values are formatted as specified by the **formatSetDateTimeDefault** method (System type), or by ObjectPAL formatting statements.

You must specify a DateTime value completely; you cannot omit any of the fields, but you can specify a value of zero for any field.

See also

- day, hour, month, year

day

DateTime

Beginner

Method

Extracts the day of the month from a DateTime.

Syntax

day ( ) SmallInt

Description

Extracts the day of the month from a DateTime value and returns a value between 1 and 31. If the DateTime is invalid, the method fails.

Example

In this example, a button's **pushButton** method displays the current day of the month in a dialog box. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  theDay DateTime
endVar
theDay   DateTime("12:00:00 am 12/22/92")

  ; displays 22 in a dialog box
msgInfo("Day of the month", theDay.day())

endmethod
```

See also

- dow, dowOrd, doy, month, moy, year

---

daysInMonth

DateTime

Beginner

Method

Returns the number of days in a month.

Syntax

daysInMonth ( ) SmallInt

Description

Given a valid DateTime value, **daysInMonth** returns the number of days in that month. If the DateTime is not valid, the method fails.

Example

In the following example, the **pushButton** method for the *FebDays* button displays the number of days in February 1992. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; FebDays::pushButton
method pushButton(var eventInfo Event)
var
  daysInFeb SmallInt
endVar
daysInFeb   daysInMonth(DateTime("5:15:35 AM 2/1/92"))
msgInfo("Number of days", "There are " + String(daysInFeb) +
           " days in February 1992")

  ; displays "There are 29 days in February 1992" in a dialog box
  ; (1992 is a leap year)
endmethod
```

See also

- day, dow, dowOrd, moy

---

dow

DateTime

Beginner

Method

Returns the day of the week of a DateTime.

Syntax **dow ( )** String

Description

Given a valid DateTime value, **dow** returns the first three letters of the day of the week of that DateTime. If the DateTime is not valid, the method fails.

Example

This example displays, in a dialog box, the day of week for a given DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; showDay::pushButton
method pushButton(var eventInfo Event)
var
  theDate DateTime
endVar

theDate = DateTime("11:20:15 pm 3/9/93")

  ; displays "Tue" in a dialog box
msgInfo("Day of Week", strVal(theDate) + " falls on a " + dow(theDate))

endmethod
```

See also

- DateTime, day, dowOrd, doy, moy

---

dowOrd

Beginner

DateTime

Method

Returns the number of a day of the week.

Syntax **dowOrd ( )** SmallInt

Description

Given a valid DateTime value, **dowOrd** returns an integer from 1 to 7 representing that day's position in the week. Sunday is day 1, Monday is day 2, and so on. If the DateTime is not valid, the method fails.

Example

The following example displays the day of the week as an entire word (such as "Monday") rather than an abbreviation or a number. This code uses **dowOrd** to retrieve the appropriate subscript of a fixed array, then displays the value of the array element in a dialog box. This code is attached to the **pushButton** method for the *fullDay* button. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; fullDay::pushButton
method pushButton(var eventInfo Event)
var
  fullDays Array[7] String
```

```
  givenDate         DateTime
endVar

fullDays[1] = "Sunday"
fullDays[2] = "Monday"
fullDays[3] = "Tuesday"
fullDays[4] = "Wednesday"
fullDays[5] = "Thursday"
fullDays[6] = "Friday"
fullDays[7] = "Saturday"

givenDate = DateTime("5:35:20 AM 12/25/93")
  ; this displays "Saturday" in a dialog box
msgInfo("Day of the week", fullDays[dowOrd(givenDate)])

endmethod
```

See also

❐ DateTime, day, dow, doy, moy

doy

Beginner

DateTime

Method

Returns the number of a day of the year.

Syntax

doy ( ) SmallInt

Description

Given a valid DateTime, **doy** returns an integer from 1 to 366 representing that day's position in the year. January 1 is day 1, February 1 is day 32, and so on. If the DateTime is not valid, the method fails.

Example

This example displays a day's position in a specified year. This code is attached to a button's **pushButton** method. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  theDate DateTime
endVar

theDate = DateTime("5:35:20 AM 6/1/92")

  ; this displays "5:35:20, 6/1/92 is
  ; 153 days past the first of the year"
msgInfo("Date", String(theDate) + " is " + String(theDate.doy()) +
                " days past the first of the year.")

endmethod
```

See also

❐ dow, moy

DateTime

hour

Beginner

DateTime

Method

Extracts as a number the hour from a DateTime.

Syntax

hour ( ) SmallInt

Description

Given a valid DateTime, **hour** returns an integer representing the hour of the day in 24-hour format. This method fails if the DateTime is not valid.

Example

The following code extracts the hour from a given DateTime and displays it in a dialog box. Note that even though the DateTime given is in 12-hour format, **hour** returns the 24-hour equivalent.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
endVar

dt = DateTime("8:15:18 pm 12/29/92")
msgInfo("Hour", dt.hour())     ; displays 20 in a dialog

endmethod
```

See also

- day, millisec, minute, month, year

isLeapYear

Beginner

DateTime

Method

Reports whether a year has 366 days.

Syntax

isLeapYear ( ) Logical

Description

Given a valid DateTime, **isLeapYear** returns True if the year within DateTime has 366 days; otherwise, it returns False. This method fails if the DateTime is not valid.

Example

For this example, the **pushButton** method for the *testLeapYr* button displays a True if the given DateTime is a leap year; otherwise the method displays False. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  bDay     DateTime
```

```
  leapYear Logical
endVar

bDay = DateTime("5:35:20 AM 6/1/92")

leapYear = bDay.isLeapYear()
leapYear.view("bDay")          ; displays True

endmethod
```

See also

- year

milliSec

Beginner

DateTime

Method

Extracts as a number the milliseconds from a DateTime.

Syntax

milliSec ( ) SmallInt

Description

Given a valid DateTime, **milliSec** returns an integer representing the milliseconds. This method fails if the DateTime is not valid.

Example

This example constructs a DateTime value from integer calculations, then displays the milliseconds portion of the DateTime in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
  oneSecond, oneMinute, oneHour LongInt
endVar
oneSecond = 1000               ; milliseconds
oneMinute = oneSecond * 60
oneHour   = oneMinute * 60

  ; the following statement assigns dt a DateTime value
  ; of "1:20:30.4 pm 00/00/00" (the statement does not
  ; assign a date, so DateTime sets date portion to 0)
dt = DateTime(13 * oneHour   +
              20 * oneMinute +  ; specifies 1:20 pm
              30 * oneSecond +  ; + 30 seconds
              400)              ; + 400 milliseconds

msgInfo("Milliseconds", dt.milliSec())   ; displays 400

endmethod
```

See also

- hour, minute, second

minute

DateTime

Beginner

Method

Extracts as a number the minutes from a DateTime.

Syntax

minute ( ) SmallInt

Description

Given a valid DateTime, **minute** returns an integer representing the minutes. This method fails if the DateTime is not valid.

Example

For this example, the **pushButton** method for *thisButton* displays the minutes portion of a given DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
endVar

dt = DateTime("9:20:15 am 8/2/93")

msgInfo("Minutes", dt.minute())    ; displays 20

endmethod
```

See also

- hour, milliSec, second

month

DateTime

Beginner

Method

Extracts as a number the month from a DateTime.

Syntax

month ( ) SmallInt

Description

Given a valid DateTime, **month** returns an integer representing the position in the year of that date's month. January is month 1, February is month 2, and so on. This method fails if the DateTime is not valid.

Example

The following example displays the month of the year as an entire word (such as "August") rather than an abbreviation or a number. This code uses **month** to retrieve the appropriate subscript of a fixed array, then displays the value of the array element in a dialog box. This code is attached to the **pushButton** method for the *fullMonth* button. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; fullMonth::pushButton
method pushButton(var eventInfo Event)
var
  fullMonth Array[12] String
  orderDate DateTime
endVar

fullMonth[1] =  "January"
fullMonth[2] =  "February"
fullMonth[3] =  "March"
fullMonth[4] =  "April"
fullMonth[5] =  "May"
fullMonth[6] =  "June"
fullMonth[7] =  "July"
fullMonth[8] =  "August"
fullMonth[9] =  "September"
fullMonth[10] = "October"
fullMonth[11] = "November"
fullMonth[12] = "December"

orderDate = DateTime("5:35:20 AM 9/18/93")

  ; this displays "September" in a dialog box
msgInfo("Order Month", fullMonth[month(orderDate)])

endmethod
```

See also

- moy

---

moy

Beginner

DateTime

Method

Extracts as a string the month from a DateTime.

Syntax

moy ( ) String

Description

Given a valid DateTime, **moy** returns the first three letters of the name of that date's month. This method fails if the DateTime is not valid.

Example

For this example, the **pushButton** method for *thisButton* displays the abbreviated month name of a specified DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  orderDate DateTime
endVar

orderDate = DateTime("2:09:00 AM 3/3/97")
msgInfo("Order date", orderDate.moy())    ; displays Mar

endmethod
```

See also

- month

second

DateTime

Beginner

Method

Extracts as a number the seconds from a DateTime.

Syntax

second ( ) SmallInt

Description

Given a valid DateTime, **second** returns an integer representing the seconds. This method fails if the DateTime is not valid.

Example

This example constructs a DateTime value from integer calculations then displays the seconds portion of the DateTime in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dt DateTime
  oneSecond, oneMinute, oneHour LongInt
endVar
oneSecond = 1000                ; milliseconds
oneMinute = oneSecond * 60
oneHour   = oneMinute * 60

  ; the following statement assigns dt a DateTime value
  ; of "1:20:30.4 pm 00/00/00" (the statement does not
  ; assign a date, so DateTime sets date portion to 0)
dt = DateTime(13 * oneHour   +
              20 * oneMinute +   ; specifies 1:20 pm
              30 * oneSecond +   ; + 30 seconds
              400)               ; + 400 milliseconds

msgInfo("Seconds", dt.second())  ; displays 30

endmethod
```

See also

- hour, milliSec, minute

year

DateTime

Beginner

Method

Extracts as a number the year from a DateTime.

Syntax

year ( ) SmallInt

Description

Given a valid DateTime, **year** returns an integer representing the year within the DateTime. If the DateTime is invalid, this method fails.

Example

For this example, the **pushButton** method for the *yearButton* button displays the four-digit year for a specified DateTime. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

```
; yearButton::pushButton
method pushButton(var eventInfo Event)
var
  orderDate DateTime
endVar

orderDate = DateTime("2:15:24 pm 3/3/97")
msgInfo("Order date", orderDate.year())   ; displays 1997

endmethod
```

See also

- day, isLeapYear, month, moy

DDE

DDE

close
execute
open
setItem

Dynamic data exchange (DDE) is a Windows protocol that lets Paradox share data with other applications that behave according to the DDE protocol. Using DDE methods, you have access to data created and stored in another application. You can also use DDE methods to send commands and data to other applications.

Refer the *ObjectPAL Developer's Guide* for information about using DDE and ObjectPAL.

Note Paradox and ObjectPAL also support OLE, another protocol for sharing data between applications. Refer to the "OLE" section in this chapter, and to the *ObjectPAL Developer's Guide* for information about OLE and ObjectPAL. Refer to the *User's Guide* for information about using Paradox and OLE objects interactively.

close

DDE

Method

Closes a DDE link.

Syntax

close ( ) Logical

Description

Ends a DDE conversation. It closes documents in one application, but the other application stays open.

Example

The following code opens the ObjectVision form ADDRESS.OVD, gets the values of the Name field and the Company field, then calls **close** to close the DDE link:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  getNames DDE
  name, company AnyType
endVar

; link to an ObjectVision form
getNames.open("vision", "C:\\vision\\sample\\Address.ovd")

getNames.setItem("Name")          ; item is field "Name"
name = getNames                   ; set name = field "Name"
```

```
getNames.setItem("Company")         ; item is field "Company"
company = getNames                  ; sets company = field "Company"

msgInfo("From Address.ovd",         ; display info from ObjectVision
        "Name : " + name + "\n" +
        "Company : " + company )

getNames.close()                    ; close the link

endmethod
```

See also

- open, setItem

execute

DDE

Method

Sends a command via a DDE link.

Syntax

execute (const ***command*** String) Logical

Description

Sends the string *command* to an application via a DDE link. The nature of *command* will vary from one application to another. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Example

The following code uses the ObjectVision function @SETTITLE to specify the text to display in the ObjectVision title bar.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dl DDE
endVar

; open a link to the ObjectVision form named Address
dl.open("vision", "C:\\vision\\forms\\Address.ovd")

; execute the ObjectVision @SETTITLE command
dl.execute("[@SETTITLE(\"I'm in charge!\")]")

endmethod
```

See also

- close, open, setItem

DDE

open

DDE

Method

Opens a DDE link to another application.

Syntax

1. **open** (const ***server*** String) Logical
2. **open** (const ***server*** String, const ***topic*** String) Logical
3. **open** (const ***server*** String, const ***topic*** String, const ***item*** String) Logical

Description

Creates a DDE link to the application *server*, and tells *server* to open the document *topic* (optional) at a location specified in *item* (optional).

This method returns True if application *server* is successfully opened; otherwise it returns False. If the server application cannot open *topic*, or if *item* cannot be found in *topic*, this method fails.

The nature of *item* varies from one application to another. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Example

The following code opens two DDE links (represented by the DDE variables *d1* and *d2*) to the ObjectVision form ADDRESS.OVD.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  d1, d2 DDE
  name, company AnyType
endVar

d1.open("vision", "C:\\vision\\forms\\Address.ovd", "Name")
d2.open("vision", "C:\\vision\\forms\\Address.ovd", "Company")

name = d1            ; name takes value from "Name"
company = d2         ; company takes value from "Company"

                     ; display info from ObjectVision fields
msgInfo("Address.ovd Info",
        "Name : " + name + "\n" +
        "Company : " + company )

d1.close()                    ; close DDE links
d2.close()

endmethod
```

See also

- close, execute, setItem

setItem

DDE

Method

Specifies an item in a DDE conversation.

Syntax

setItem (const ***server*** String)

Description

Is used in a DDE link with application and topic established. *server* specifies a new item. The nature of *server* varies from application to application. For example, a string that makes perfect sense to a word processing program may not be understood by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Example

The following code opens a DDE link to the ObjectVision form ADDRESS.OVD, then calls **setItem** to link first to the Name field, then to the Company field in the ObjectVision form.

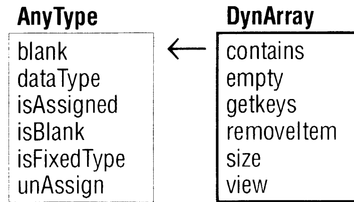
```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  getNames DDE
  name, company AnyType
endVar

; link to an ObjectVision form
getNames.open("vision", "C:\\vision\\sample\\Address.ovd")

getNames.setItem("Name")          ; item is field "Name"
name = getNames                   ; set name = field "Name"

getNames.setItem("Company")       ; item is field "Company"
company = getNames                ; sets company = field "Company"

msgInfo("From Address.ovd",       ; display info from ObjectVision
        "Name : " + name + "\n" +
        "Company : " + company )

getNames.close()                  ; close the link

endmethod
```

See also

- close, execute

DDE

DynArray



A DynArray is a flexibly structured dynamic array. A dynamic array is a compact storage structure for any combination of data types. Using a DynArray, you can look up values quickly, even when the dynamic array contains a large number of items.

These arrays are dynamic because you do not specify their size; the dimensions of a DynArray automatically change as items are added to it or released from it. A DynArray's size is limited only by system memory.

Note ObjectPAL also supports fixed-size and resizeable arrays. See the description of the Array type for more information.

Unlike fixed-size arrays, the indexes of dynamic arrays are not integers; dynamic array indexes can be any valid ObjectPAL expression that evaluates to a String. Each index in a dynamic array is associated with a value. For more information and examples, refer to the *ObjectPAL Developer's Guide.*

The DynArray type also includes methods defined for the AnyType type.

contains

DynArray

Method

Searches the indexes in a DynArray for a value.

Syntax

contains (const ***value*** AnyType) Logical

Description

Returns True if the index of any element in a DynArray matches *value* character for character; otherwise, it returns False. **contains** is case sensitive.

Example

The following example uses **contains** to test whether a dynamic array index corresponds to a menu item. In this example, the form's **open** method creates a menu and assigns several values to a dynamic array. When the user selects an item from the menu, the form's **menuAction** method compares the menu selection with indexes in the DynArray. If a DynArray index is defined for the selected menu item, the **menuAction** method displays the value associated with that DynArray element; otherwise it displays the value of another element.

This code goes in the form's Var window:

```
; thisForm::Var
var
  msg DynArray[] AnyType    ; stores messages
  m1             Menu       ; menu bar
  p1             PopUpMenu  ; pop-up attached to menu item
  choice         String     ; user's menu selection
endVar
```

The code immediately following is attached to the **open** method of a form:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    p1.addText("Time")              ; add items to the pop-up menu
    p1.addText("Date")
    p1.addText("Colors")

    m1.addPopUp("&Utilities", p1)  ; attach the pop-up to a menu bar item
    m1.show()                       ; show the menu bar

      ; Now initialize the msg dynamic array. msg Indexes correspond to
      ; the pop-up menu items generated above. msg values are values that
      ; appear in a dialog box when the user selects a menu. Note that
      ; msg does NOT contain a "Colors" index.
    msg["Time"] = time()            ; show current date for "Time" selection
    msg["Date"] = date()            ; show current date for "Date" selection
    msg["Error"] = "Sorry, this menu selection is not implemented."

endif
endmethod
```

This code is attached to the **menuAction** method of a form:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    choice = eventInfo.menuChoice()

    if isBlank(choice) = False then    ; if user selected a menu
      if msg.contains(choice) then     ; if selection matches an index in
                                       ; the msg dynamic array
        msgInfo(choice, msg[choice])   ; display the value of that element
```

```
      else                            ; else selection didn't match an element
        msgStop("Stop!", msg["Error"]) ; display the value of another element
      endif
    endif

  else
    ;code here executes just for form itself
endif
endmethod
```

See also

- getKeys, view

empty

Array

Method

Removes all items from a dynamic array.

Syntax

empty ( )

Description

Removes all items from an dynamic array. The size of the DynArray becomes 0.

Example

This example shows how **empty** functions for a dynamic array. The code immediately following declares a dynamic array in a form's Var window. This dynamic array is global to all objects on the form.

```
; thisForm::Var
Var
  myCar DynArray[] AnyType  ; declare a dynamic array
endVar
```

The following code is attached to the **pushButton** method of the *fillButton*. When this button is pressed, the code assigns several elements of the *myCar* DynArray.

```
; fillButton::pushButton
method pushButton(var eventInfo Event)

myCar["Make"]  = "Porshe"  ; load the DynArray
myCar["Model"] = "911 sc"
myCar["Color"] = "Dark Blue"
myCar["Year"]  = 1986
  ; display myCar DynArray and indicate size in the title (4)
myCar.view("myCar size: " + String(myCar.size()))
endmethod
```

The following code is attached to the **pushButton** method of the *emptyButton* button. When this button is pressed, the code empties the *myCar* array and displays its contents.

```
; emptyButton::pushButton
method pushButton(var eventInfo Event)
myCar.empty()     ; empty the myCar DynArray

  ; display myCar DynArray and indicate size in the title (0)
```

```
myCar.view("myCar size: " + String(myCar.size()))
endmethod
```

See also

- contains

getKeys

DynArray

Method

Loads a resizable array with indexes of an existing DynArray.

getKeys (var ***keyNames*** Array[] String)

Description

Creates the resizable array specified in *keyNames* and assigns to the values of each element the index in the DynArray. In other words, this method stores all index values from a DynArray in a resizable array. If *keyNames* exists, it is overwritten without asking for confirmation. Index values are sorted into the new array such that the lowest index value becomes *keyNames*[1], and so on.

Example

This example assigns several elements to the *myCar* DynArray, then uses **getKeys** to create an array that stores *myCar* indexes. The results are displayed in a **view** dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myCar DynArray[] AnyType
  ar    Array[] String
endVar

; add some elements to the DynArray
myCar["Make"]  = "Porshe"  ; load the DynArray
myCar["Model"] = "911 sc"
myCar["Color"] = "Dark Blue"
myCar["Year"]  = 1986

; now grow ar to 4 items then view the
; new array in a dialog box
myCar.getKeys(ar)
ar.view()

; displays
;  Color     (ar[1])
;  Make      (ar[2])
;  Model     (ar[3])
;  Year      (ar[4])

endmethod
```

See also

- contains

removeItem

DynArray

Method

Deletes a specified item from a DynArray.

Syntax

removeItem (const ***value*** AnyType)

Description

Deletes the element (specified by its index) in *value* from a DynArray. **removeItem** is case insensitive.

Example

The following example concatenates two values in a dynamic array, then uses **removeItem** to remove the obsolete element.

The code immediately following is attached to a form's Var window:

```
; thisForm::Var
var
  CustInfo DynArray[] AnyType
endVar
```

This code is attached to the **pushButton** method for the *getCustInfo* button. This code loads the dynamic array with street address information. Your application might have a custom method that loads the dynamic array from a table, or from information entered by the user.

```
; getCustInfo::pushButton
method pushButton(var eventInfo Event)
  ; load the DynArray
CustInfo["Company"] = "Ultra-Fast Computers"
CustInfo["Street"]  = "1234 Able Street"
CustInfo["City"]    = "Anywhere"
CustInfo["State"]   = "Your State"
CustInfo["Zip"]     = "99444"
CustInfo["ZipExt"]  = "9344"

  ; display contents of the CustInfo Dynarray
CustInfo.view("Contents of CustInfo")
endmethod
```

In the code that follows, the value of the ZipExt element (if it exists) is concatenated to the value of the Zip element. Since the ZipExt element is no longer needed, this code removes it from the dynamic array. The following code is attached to the **pushButton** method for the *catZipExt* button.

```
; catZipExt::pushButton
method pushButton(var eventInfo Event)
if CustInfo.contains("ZipExt") then
  CustInfo["Zip"] = CustInfo["Zip"] + "-" + CustInfo["ZipExt"]
  CustInfo.removeItem("ZipExt")       ; remove obsolete element
else
  msgInfo("Once is enough", "Zip code has been concatenated")
endif
  ; display the results
```

```
CustInfo.view("Contents of CustInfo")
endmethod
```

See also

- contains, empty

size

DynArray

Method

Returns the number of elements in a DynArray.

Syntax

size ( ) LongInt

Description

Returns the number of elements in a DynArray.

Example

For this example, the **pushButton** method for *thisButton* creates a dynamic array, then displays its size in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] String
endVar

dy["Name"]     = "MAST"          ; load the DynArray
dy["Business"] = "Diving"
dy["Contact"]  = "Jane Doherty"

  ; this displays "dy has 3 elements"
msgInfo("dy", "dy has " + string(dy.size()) + " elements.")
endmethod
```

See also

- contains

view

DynArray

Method

Displays the contents of a DynArray in a dialog box.

Syntax

view ([const ***title*** String])

Description

List the indexes and elements of a DynArray in a modal dialog box. ObjectPAL execution suspends until the user closes this dialog box. You can specify a title for the dialog box in *title*, or you can omit *title* to display "DynArray" instead. **view** sorts the DynArray on its index before displaying the dialog box.

Unlike many other data types, DynArray values displayed in a **view** dialog can not be changed interactively. See "AnyType" earlier in this

chapter for more information regarding other data types and the **view** method.

Example

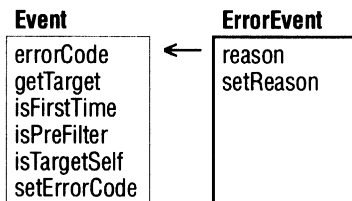
For this example, the **pushButton** method for the *thisButton* button creates a dynamic array, then displays its contents sorted in a dialog box.

```
;thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dy DynArray[] String
endVar

dy["one"] = "first"
dy["two"] = "second"
dy["three"] "third"
dy.view("This DynArray contains:")
  ; displays the following:
  ; This DynArray contains:
  ; one     first
  ; three   third
  ; two     second
endmethod
```

See also

- contains

ErrorEvent



The ErrorEvent type provides methods you can use to get and set information about errors that occur as ObjectPAL code executes. The ErrorEvent type includes several methods defined for the Event type.

The only built-in method triggered by an ErrorEvent is **error**. This method, along with the rest of the built-in methods, is discussed in Chapter 2. For information about the event model, see the *ObjectPAL Developer's Guide.*

For more information about errors and error handling, see Chapter 13 in the *ObjectPAL Developer's Guide.*

reason

ErrorEvent

Method

Reports why an error occurred.

Syntax

reason **( )** SmallInt

Description

Returns an integer value to report why an ErrorEvent occurred. ObjectPAL provides the following constants for testing the value returned by **reason** (also listed in the Constants dialog under ErrorReasons):

- ErrorCritical indicates that an error message will appear in a dialog box.
- ErrorWarning means that an error message will appear in the status line.

Note Do not confuse **reason** with **errorCode** (see the Event type for a description of **errorCode**).

Example

The following code is attached to the built-in **error** method for the form. This code reports the error code, the reason, and the message associated with the error.

```
; thisForm::error
method error(var eventInfo ErrorEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.reason() = ErrorWarning then
      msgInfo("Warning Error", errorMessage())
    else
      msgInfo("Critical Error", errorMessage())
    endif
    disableDefault
  else
    ; code here executes just for form itself

endif
endmethod
```

See also

- setReason
- errorCode in the Event type

setReason

ErrorEvent

Method

Specifies a reason for generating an ErrorEvent.

Syntax

setReason (const ***reasonId*** SmallInt)

Description

Specifies a reason for generating an ErrorEvent. This method takes one of the following reason constants as an argument (also listed in the Constants dialog under ValueReasons):

- ErrorCritical indicates that an error message will appear in a dialog box.
- ErrorWarning means that an error message will appear in the status line.

Example

The following example creates an ErrorEvent, sets the reason to ErrorCritical, then sends the ErrorEvent to the form.

```
; sendAnError::pushButton
method pushButton(var eventInfo Event)
var
  ev  ErrorEvent
endVar
ev.setErrorCode(1)        ; set an error code of 1 (any nonzero will do)
ev.setReason(ErrorWarning) ; set the reason to ErrorWarning
thisForm.error(ev)        ; send the error to the form
endmethod
```

See also

- reason

Event

Event

errorCode
getTarget
isFirstTime
isPreFilter
isTargetSelf
reason
setErrorCode
setReason

The Event type is the base type from which the other event types (fo· example, ActionEvent) are derived. Many of the methods listed in this section are used by the other event types.

The following built-in methods are triggered by Events: **open**, **close**, **setFocus**, **removeFocus**, **newValue**, and **pushButton**. These methods, along with the rest of the built-in methods, are discussed in Chapter 2. For information about the event model and more examples, see Chapter 6 in the *ObjectPAL Developer's Guide.*

errorCode

Event

Beginner

Method

Reports the status of an error flag.

Syntax

errorCode ( ) LongInt

Description

Returns an error code if there is an error; otherwise, **errorCode** returns 0. ObjectPAL provides constants for Event error codes; see EventErrorCodes in the Constants dialog box.

Example

In this example, assume that a form contains a table frame bound to the *Customer* table, a button named *startEdit*, and another button named *sendAnError*. The *startEdit* **pushButton** method examines the event; if the event doesn't have an error, the method starts Edit mode on CUSTOMER. For the purposes of example only, the *sendAnError* button creates an event, sets its error to 1, then calls the **pushButton** method of *startEdit* with the event. The following code is attached to the **pushButton** method for *startEdit*:

```
; startEditButton::pushButton
method pushButton(var eventInfo Event)
; check the event to see if it has an error
if eventInfo.errorCode() = 0 then
  CUSTOMER.action(DataBeginEdit)  ; if no error, then start Edit mode
```

```
endif
endmethod
```

The following method is attached to the **pushButton** method for *sendAnError*:

```
; sendAnError::pushButton
method pushButton(var eventInfo Event)
var
  ev Event                        ; the event to dispatch
endVar
ev.setErrorCode(1)                ; set an error of 1
startEditButton.pushButton(ev)    ; send the event to the startEditButton
endmethod
```

See also

- setErrorCode
- Chapter 13 in the *ObjectPAL Developer's Guide*

getTarget

Event

Method

Creates a handle to the target of an Event.

Syntax

getTarget (var ***target*** UIObject)

Description

Returns in *target* the handle of the UIObject that was the target of the most recent Event.

Example

This example assumes that a number of fields from the *Customer* table are placed on a form. As the user moves from field to field, the **setFocus** method on the form identifies the target of the Event, finds out if the target is a field, and, if so, changes the current field's color to light blue. This provides a more dramatic visual clue to the user than the normal highlight. The field's previous color is stored in the global variable *oldFieldColor*. When the focus is removed from the field, the form's **removeFocus** method restores the field to its original color. The previous field color is stored in a variable declared in the Var window of the form, as shown in the following code:

```
; thisForm::Var
Var
  oldFieldColor LongInt     ; to store the previous color of the field
endVar
```

The following code is attached to the **setFocus** method of the form:

```
; thisForm::setFocus
method setFocus(var eventInfo Event)
var
  targObj    UIObject
endVar
if eventInfo.isPreFilter()
```

```
  then
    ; code here executes for each object in form
    ; get the target
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then  ; if it's a field, change its color
      oldFieldColor = targObj.Color  ; save old color in var global to form
      targObj.Color = LightBlue      ; highlight field on focus
    endif
  else
    ; code here executes just for form itself

endif
endmethod
```

This code is attached to the form's **removeFocus** method:

```
; thisForm::removeFocus
method removeFocus(var eventInfo Event)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; get the target
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then  ; if it's a field,
      targObj.Color = oldFieldColor  ; restore color from global var
    endif
  else
    ; code here executes just for form itself

endif
endmethod
```

See also

- isPreFilter

isFirstTime

Event

Method

Reports whether the form is handling an Event for the first time before dispatching it.

Syntax

isFirstTime ( ) Logical

Description

Reports whether the form is handling an Event before dispatching it to the target object, or whether the event has been dispatched and has subsequently bubbled up the containership hierarchy. This method returns True if the form is handling the Event for the first time; otherwise, it returns False. Use **isFirstTime** in built-in methods attached to the form.

Example

This example shows how you can use **isFirstTime** with **isTargetSelf** to evaluate an event in a form-level method. This code replaces the

default code for the form's **pushButton** method, which normally tests **isPreFilter**.

```
; thisForm::pushButton
method pushButton(var eventInfo Event)
var
  targObj   UIObject
endVar
; This example breaks out isFirstTime and isTargetSelf from isPreFilter.
; Three valid possibilities.
; Form's own event                       : isTargetSelf True, isFirstTime True
; Dispatched events (prefiltered events): isTargetSelf False, isFirstTime True
; Bubbled events (explicitly passed)    : isTargetSelf False, isFirstTime False
; For the form, isTargetSelf is never True when isFirstTime is False.

eventInfo.getTarget(targObj)    ; get the target to targObj
switch
  case eventInfo.isTargetSelf() AND eventInfo.isFirstTime() :
    ; This happens only when the form is handling its own event.
  msgInfo("Status", "This line will not execute for pushButton events.")

  case NOT eventInfo.isTargetSelf() AND eventInfo.isFirstTime() :
    ; This happens only when the form is dispatching an event
    ;   for another object. isPrefilter returns True in this situation.
  msgInfo("Status", "About to dispatch a pushButton event to "
                    + targObj.Name + ".")

  case NOT eventInfo.isTargetSelf() AND NOT eventInfo.isFirstTime() :
    ; This happens when event has been explicitly bubbled back to the form.
    ;   isPrefilter returns False in this situation.
  msgInfo("Status", "This dialog appears only when a pushButton Event " +
          "has been explicitly bubbled back to the form.")
endswitch

endmethod
```

The following code is attached to the **pushButton** method for the form's *testPassEvent* button. When the form's **pushButton** method has prefiltered the event and dispatched it back to the button, the button's **pushButton** method returns it to the form with the command **passEvent**. When the event returns to the form, the methods **isTargetSelf**, **isFirstTime**, and **isPrefilter** all return False.

```
; testPassEvent::pushButton
method pushButton(var eventInfo Event)
passEvent    ; bubble the event up the heirarchy
endmethod
```

See also

- isPreFilter

isPreFilter

Event

Method

Reports whether the form is handling an Event on its own behalf.

Syntax

isPreFilter ( ) Logical

Description

Reports whether the form is handling an Event on its own behalf or on behalf of another object. It returns True only when the target is some object other than the form, and the form has not already handled this Event. **isPreFilter** is logically equivalent to the form evaluating the following statement:

```
if (NOT eventInfo.isTargetSelf()) AND eventInfo.isFirstTime()
```

This method returns True for all internal methods, and for all external methods when they first reach the form.

When the external methods bubble back to the form, this method returns False.

Note Form methods are *not* prefiltered. In other words, when an Event occurs for the form, **isPreFilter** returns False.

Example

See the example for getTarget.

See also

- isFirstTime
- The discussion of the event model in Chapter 6 in the *ObjectPAL Developer's Guide*

---

isTargetSelf

Event

Method

Reports whether an object is the target of an Event.

Syntax

isTargetSelf ( ) Logical

Description

Reports whether an object is the target of an Event. Use **isTargetSelf** in built-in methods attached to the form.

Example

See the example for isFirstTime.

See also

- isFirstTime

---

reason

Event

Method

Reports why an Event occurred.

Syntax

reason ( ) SmallInt

Description

Returns an integer value to report why an Event occurred. **reason** returns valid reason constants only for Events generated for the built-in **newValue** method (see the examples). ObjectPAL provides the following constants for testing the value returned by **reason** (also listed in the Constants dialog under ValueReasons):

- FieldValue means that the value of a field was changed by scrolling, by refresh across the network, by an ObjectPAL statement, or by a user changing the value of a field.
- EditValue means that a value was specified by clicking a radio button or choosing an item from a list.
- StartupValue means that a value was specified when the form was opened.

Note Reason constants are also defined for ErrorEvents, MenuEvents, MoveEvents, and StatusEvents. See the entry for **reason** in those sections for examples. The **reason** method is valid for the other Event types (ActionEvent, KeyEvent, MouseEvent, and ValueEvent), but it returns zero. **setReason** is also valid for ActionEvent, KeyEvent, MouseEvent, and ValueEvent, but you can use it only to set user-defined Reason constants (an advanced technique).

Example

In this example, assume that a form contains a multi-record object bound to the *Orders* table, and that the *Ship_VIA* field is a set of radio buttons. The following **newValue** method for *Ship_VIA* displays a message indicating why **newValue** was called. When the form opens, the Reason constant will be StartupValue.

```
; Ship_VIA::newValue
method newValue(var eventInfo Event)
; show why the newValue method was called
msgInfo("newValue reason",
       iif(eventInfo.reason() = StartupValue, "StartupValue",
       iif(eventInfo.reason() = FieldValue, "FieldValue", "EditValue")))
endmethod
```

When the user scrolls through the table or clicks the *nextRec* button, the Reason will be FieldValue.

```
; nextRec::pushButton
method pushButton(var eventInfo Event)
action(DataNextRecord)    ; this triggers a newValue for Ship_Via
                          ; with a Reason constant FieldValue
endmethod
```

When the user chooses a different radio button on *Ship_VIA* or clicks the *changeRadio* button, the Reason will be EditValue.

```
; changeRadio::pushButton
method pushButton(var eventInfo Event)
ORDERS.Ship_Via = "US Mail"    ; this triggers a newValue for Ship_Via
                               ; with a Reason of EditValue
endmethod
```

See also

- setReason

setErrorCode

Event

Beginner

Method

Sets the error code for an Event.

Syntax

setErrorCode (const ***errorId*** LongInt)

Description

Sets the error code. If *errorId* is 0, it means "no error." Any nonzero value for *errorId* indicates an error.

ObjectPAL provides constants for *errorId*; see EventErrorCodes in the Constants dialog.

Example

See the example for errorCode.

See also

- errorCode
- Chapter 13 in the ObjectPAL Developer's Guide

setReason

Event

Method

Specifies a Reason for an event.

Syntax

setReason (const ***reasonId*** SmallInt)

Description

Specifies a Reason for an Event. This method takes one of the following Reason constants as an argument (also listed in the Constants dialog under ValueReasons):

- FieldValue means that the value of a field was changed by scrolling, by refresh across the network, by an ObjectPAL statement, or by a user changing the value of a field.
- EditValue means that a value was specified by clicking a radio button or choosing an item from a list.
- StartupValue means that a value was specified when the form was opened.

Note Reason constants are also defined for ErrorEvents, MenuEvents, MoveEvents, and StatusEvents. See the entry for **setReason** in those sections for examples. The **setReason** method is valid for the other

Event

event types (ActionEvent, KeyEvent, MouseEvent, TimerEvent, and ValueEvent), but you can use it only to set user-defined reason constants (an advanced technique).

Example

In this example, assume that a form contains a multi-record object bound to the *Orders* table, and that the *Ship_VIA* field is a set of radio buttons. The following **newValue** method for *Ship_VIA* displays a message indicating why **newValue** was called.

```
; Ship_VIA::newValue
method newValue(var eventInfo Event)
; show why the newValue method was called
msgInfo("newValue reason",
        iif(eventInfo.reason() = StartupValue, "StartupValue",
        iif(eventInfo.reason() = FieldValue, "FieldValue", "EditValue")))
endmethod
```

The following code demonstrates how to set a Reason for an Event and send the Event to an object.

```
; triggerValReason::pushButton
method pushButton(var eventInfo Event)
var
  ev  Event
endVar
ev.setReason(FieldValue)       ; set a reason constant for the event
ORDERS.Ship_VIA.newValue(ev)   ; send the event to the Ship_VIA field
endmethod
```

See also

- reason

FileSystem

FileSystem

accessRights	isRemote
copy	isRemovable
delete	makeDir
deleteDir	name
drives	privDir
enumFileList	rename
existDrive	setDir
findFirst	setDrive
findNext	setFileAccessRights
freeDiskSpace	size
fullName	splitFullFileName
getDir	statUpDir
getDrive	time
getFileAccessRights	totalDiskSpace
getValidFileExtensions	windowsDir
isDir	windowsSystemDir
isFile	workingDir
isFixed	

FileSystem variables provide access to and information about disk files, drives, and directories. A FileSystem variable provides a handle, a variable you can use in ObjectPAL statements to work with a directory or a file. In many cases, the first step in working with FileSystem variables is using **findFirst** to see if any information is present. It may be helpful to think of this step as initializing the FileSystem variable.

For more information and examples, refer to the *ObjectPAL Developer's Guide.*

Also, refer to the entry for the **fileBrowser** procedure defined for the System type, which lets you display and get input from the Paradox Browser.

accessRights

FileSystem

Method

Reports access rights (also called file attributes) of a file.

Syntax

accessRights ( ) String

Description

Returns a string describing access rights. Return values can be one or more of the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If the returned value is

FileSystem

an empty string, the file has no attributes set. You must use **findFirst** before using **accessRights**.

Example

This example checks the attributes of the file MEMO14.TXT. It calls **findFirst** to make sure the file exists, then calls **accessRights**. If the file is not marked read-only, this code runs Windows Notepad to edit the file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fileName String
  fs       FileSystem
endvar

fileName = "c:\\pdoxwin\\myfiles\\memo14.txt"

if fs.findFirst(fileName) then

  ; if file attributes include R (read only)
  if search(fs.accessRights(), "R") > 0 then
    msgStop(fileName, "This file is marked read-only.")
  else
    ; run notepad editor for the file
    execute("NotePad.exe " + fileName)
  endIf
else
  msgStop("Error", "Can't find " + fileName)
endIf

endmethod
```

See also

- getFileAccessRights, setFileAccessRights

---

copy

FileSystem

Method

Copies a file.

Syntax

copy (const ***srcName,*** String ***dstName*** String) Logical

Description

Returns True if **copy** has been successful in copying source file *srcName* to destination file *dstName*; otherwise, it returns False. If *dstName* exists, **copy** overwrites it without asking for confirmation. **copy** can only copy one file at a time. You can't use DOS wildcard characters with **copy**.

Example

The code in this example searches the current directory for the file specified in the variable *sourceFile*. If the file exists, this code copies it and gives the new file the name specified in *destFile*.

```
; copyButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  sourceFile, destFile String
endvar

sourceFile = "memo14.txt"
destFile = "memo14.bak"

if fs.findFirst(sourceFile) then
  if fs.copy(sourceFile, destFile) then
    message(sourceFile + " copied to " + destFile)
  else
    message("Copy failed...")
  endif
else
  msgInfo(sourceFile, "File not found.")
endIf

endmethod
```

See also

- delete, rename

delete

FileSystem

Method

Deletes a file.

Syntax

delete (const ***name*** String) Logical

Description

Returns True if **delete** has been successful in deleting the file specified in *name*; otherwise, it returns False. **delete** can only delete one file at a time. You can use DOS wildcard characters, but **delete** will only delete the first file it encounters that matches the file specification.

Example

The first example displays a dialog box asking if the user wants to delete the file specified in the variable *fileName*. If the user chooses Yes, the call to **delete** deletes the file:

```
; delOne::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  oldFile String
endvar

fileName = "MyText.old"

if fs.findFirst(fileName) then
  if msgYesNoCancel("Delete?", fileName) = "Yes" then
    fs.delete(fileName)
  endIf
else
```

FileSystem

```
  msgInfo(fileName, "File not found.")
endIf

endmethod
```

The next example uses a **while** loop to delete all files in the current directory that have an extension of .OLD.

```
; delAll::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

if fs.findFirst("*.old") then
  fs.delete(fs.name())
  while fs.findNext()
    fs.delete(fs.name())
  endWhile
else
  msgInfo("*.OLD", "File not found.")
endIf

endmethod
```

See also

- deleteDir

deleteDir

FileSystem

Method

Deletes a directory.

Syntax

deleteDir (const ***name*** String) Logical

Description

Returns True if **deleteDir** was successful in deleting the directory specified in *name*; otherwise, it returns False. **deleteDir** does not prompt for confirmation.

Example

The first example tries to delete the directory C:\DOS. If the operation fails (for example, because the directory is not empty), a dialog box displays an error message.

```
; delDOS::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

if fs.findFirst("c:\\dos") then
 if not fs.deleteDir("c:\\dos") then
   msgStop("Error", "Could not delete directory.")
 endIf
endIf

endmethod
```

You can delete a directory only if the directory is empty (contains no files). The following examples use **enumFileList** to find if a directory is empty. As the example shows, the way to use **enumFileList** depends on the directory path and file specification. In the first example, **enumFileList** creates an array containing one item (the directory name) when the directory is empty:

```
; delDir1::pushButton
method pushButton(var eventInfo event)
var
  fs FileSystem
  fileNames Array[] String
endvar

fs.enumFileList("c:\\scan\\subscan", fileNames)

; compare size to 1 because directory has no filespec
if fileNames.size() = 1 then
  fs.deleteDir("c:\\scan\\subscan")
else
  msgStop("Stop", "Directory is not empty.")
endIf

endMethod
```

In the second example, **enumFileList** creates an array containing two items (one each for the current directory and its parent directory) because of the *.* file specification at the end of the path.

```
; delDir2::pushButton
method pushButton(var eventInfo event)
var
  fs FileSystem
  fileNames Array[] String
endvar

fs.enumFileList("c:\\scan\\subscan\\*.*", fileNames)

; compare size to 2 because directory the *.* filespec
if fileNames.size() = 2 then ; size = 2 because of *.* filespec
  fs.deleteDir("c:\\scan\\subscan")
else
  msgStop("Stop", "Directory is not empty.")
endIf

endmethod
```

See also

- delete, makeDir

drives

FileSystem

Method

Returns the letters of the drives attached to the system and known to Windows.

Syntax

drives ( ) String

Description

Returns a string containing only the letters (no colons) of the drives attached to the system and known to Windows.

Example

The following example displays a dialog box listing the drive ID letters of the drives attached to the system:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

; this displays a list of attached drives
; example:  ABCHJKXY
msgInfo("Drives", fs.drives())

endmethod
```

See also

- existDrive

---

enumFileList

FileSystem

Method

Writes information about files to a table or an array.

Syntax

1. **enumFileList** (const ***fileSpec*** String,
   var ***arrayName*** Array[] String)
2. **enumFileList** (const ***fileSpec*** String,
   const ***tableName*** String)

Description

Writes information about files matching the criteria in *fileSpec* to the array named in *arrayName* (syntax 1) or to the table named in *tableName* (syntax 2).

If *fileSpec* is *.*, the array or table includes records for the current directory (.) and the parent directory (..).

You must declare the array named in *arrayName* before calling this method. The resulting array contains file names and extensions only.

If *tableName* does not exist, it is created automatically. If *tableName* is created in the directory you're listing, the table does not appear in the list. If *tableName* does exist, information is added to it, and it appears in the list.

Here is the structure of the table:

Field name	Type	Size
Name	A	20

Size	N	
Attributes	A	10
Date	A	10
Time	A	10

File names are listed in the order they're listed in the directory—not necessarily in alphabetical order.

Example

The following example demonstrates both syntaxes of **enumFileList**. First, **enumFileList** searches the specified directory for forms and uses syntax 1 to create an array of file names, which is displayed in a pop-up menu. Then **enumFileList** uses syntax 2 to create a table of information on the files and displays it in a table window.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  formDir, theForm String
  formNames Array[] String
  tv tableView
  p PopUpMenu
endvar

formDir = "C:\\pdoxwin\\sample\\*.fsl"

if fs.findFirst(formDir) then              ; if one *.f?l is found
  fs.enumFileList(formDir, formNames)  ; create an array of *.f?l files
  p.addarray(formNames)                ; show the array in a pop-up menu
  theForm = p.show()                   ; display a pop-up menu of filenames
endIf

if fs.findFirst(formDir) then              ; if one *.f?l is found
  fs.enumFileList(formDir, "forms.db") ; create FORMS.DB listing *.f?l files
  tv.open("forms.db")                  ; display FORMS.DB table
endIf

endmethod
```

See also

- getValidFileExtensions, isFile

existDrive

FileSystem

Method

Reports whether a drive is attached to the system.

Syntax

existDrive (const ***driveLetter*** String) Logical

Description

Returns True if *driveLetter* is attached to the system; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

FileSystem

Example

This example calls **existDrive** to test for the existence of drive P. If P exists, the call to **setDrive** makes it the default drive.

```
; checkDrive::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  driveName String
endvar

driveName = "P"

if fs.existDrive(driveName) then
  fs.setDrive(driveName)
else
  msgStop("Stop", "Drive " + driveName + " is not attached.")
endIf

endmethod
```

See also

- drives, getDrive, isRemote, isRemovable, setDrive

---

findFirst

FileSystem

Method

Searches a file system for a file name.

Syntax

findFirst (const ***pattern*** String) Logical

Description

Returns True if a file, whose name matches *pattern*, is found; otherwise, it returns False. *pattern* may contain the DOS wildcard characters * and ?, as used with the DOS command DIR. Examples of *pattern* include:

"C:*.*"

"..\\myDir*.*"

"*.txt"

"fr*.db?"

Use **findFirst** to find if a file or directory exists, and to initialize a FileSystem variable before calling another FileSystem method or procedure.

Note **findFirst** finds file and directory names in the order they're listed in the directory, which is not necessarily alphabetical. The first value returned by **findFirst** depends on the path and file specification.

Example

This example demonstrates how **findFirst** behaves depending on the file specification in *pattern*:

```
; buttonOne::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

; search in the root directory for a file
; or directory named PDOXWIN
fs.findFirst("c:\\pdoxwin")

; this displays PDOXWIN (findFirst finds the directory)
msgInfo("findFirst test", fs.name())


; search in the root directory for a file
; or directory named PDOXWIN
fs.findFirst("c:\\pdoxwin\\")

; this displays PDOXWIN (findFirst finds the directory)
msgInfo("findFirst test", fs.name())


; search in the root directory for a file
; or directory named PDOXWIN
fs.findFirst("c:\\pdoxwin\\*.*")

; this displays one dot (.) because the
; first file in a directory is a single dot (.)
msgInfo("findFirst test", fs.name())

endmethod
```

See also

- findNext, name

findNext

FileSystem

Method

Searches a file system for multiple instances of a file name.

Syntax

findNext ([const ***fileSpec*** String]) Logical

Description

After **findFirst** succeeds, **findNext** searches for the next file whose name matches the pattern. **findNext** returns True if successful; otherwise, it returns False.

As a shortcut, you can use the optional argument *fileSpec* to specify a path and file specification. If you do, the call to **findFirst** is unnecessary.

Example

The first example calls **findFirst** and **findNext** to fill a list with the names of the tables in the current directory. The example assumes that a drop-down list object has already been placed in the form:

```
; fillList::pushButton
method pushButton(var eventInfo Event)
```

```
var
  fs FileSystem
endvar

if fs.findFirst("c:\\pdoxwin\\sample\\*.db") then
  ; initialize the list in the drop-down edit box
  fileListField.fileList.list.count = 0

  ; this while loop fills the list in the drop-down edit
  ; box with *.db files in the default sample directory
  while fs.findNext()
    fileListField.fileList.list.selection =
    fileListField.fileList.list.selection + 1
    fileListField.fileList.list.value = fs.name()
  endWhile
else
  msgStop("*.db?", "File not found.")
endIf

endmethod
```

The following example uses **findNext** with a file specification as an argument and displays a pop-up menu listing the files in the C:\PDOXWIN directory:

```
; editText::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  p  PopUpMenu
  choice String
endvar

; search for *.txt files in the PDOXWIN directory
; then add their names to a pop-up menu
while fs.findNext("c:\\pdoxwin\\*.txt")
  p.addText(fs.name())
endWhile

choice = p.show()                    ; show the pop-up menu
if not choice.isBlank() then         ; if user selected a file
  execute("Notepad.exe " + choice)  ; edit the file in Notepad
endif

endmethod
```

See also

- findFirst

freeDiskSpace

FileSystem

Method

Returns the amount of free space on a drive.

Syntax

freeDiskSpace (const ***driveLetter*** String) LongInt

Description

Returns the number of bytes available on drive *driveLetter*. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example

The first example displays a dialog box listing the number of bytes available on drive C:

```
; showCSpace::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

msgInfo("Free bytes on drive C:", fs.freeDiskSpace("C"))

endmethod
```

The next example compares the size of the file MEMO14.TXT and the amount of space available on the current drive. If there's enough space, the code calls **copy** to copy the file.

```
; copyFile::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

; if MEMO14.TXT exists in current directory
if fs.findFirst("memo14.txt") then

  ; if there's enough disk space for a a copy of MEMO14.TXT
  if fs.size()  fs.freeDiskSpace(fs.getDrive()) then

    ; copy the file
    fs.copy("memo14.txt", "memo14.bak")

  else
    msgStop("Copy", "Not enough disk space to copy file.")
  endIf
else
  msgStop("MEMO14.TXT", "File not found.")
endIf

endmethod
```

See also

- findFirst, getDrive, totalDiskSpace

fullName

FileSystem

Method

Returns the full path of a file.

Syntax

fullName ( ) String

Description

After a successful **findFirst** or **findNext**, **fullName** returns the full path of the found file. Use this method with **splitFullFileName** to analyze the components of a file name.

Example

This example calls **fullName** to get the full name of the first form listed in the current directory. Then it calls **splitFullFileName** to split the name into its component parts and store them in a DynArray. Then it calls **view** to display the DynArray.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  splitName DynArray[] String
  fullFileName String
endVar

; if the customer.db file is in the sample directory
if fs.findFirst("c:\\pdoxwin\\sample\\customer.db") then

  ; store the full file name to a variable
  fullFileName = fs.fullName()

  ; split file name into parts and store them in a DynArray
  splitFullFileName(fullFileName, splitName)

  ; display the component parts
  splitName.view("Split name")
endIf

endmethod
```

See also

- findFirst, findNext, name, splitFullFileName

getDir

FileSystem

Method

Returns the directory path that the FileSystem variable is pointing to.

Syntax

getDir ( ) String

Description

Returns a string representing the path of the directory that the FileSystem variable is pointing to. The string does not include the drive letter—use **getDrive** for that.

Example

This example copies the form DIVEPLAN.FSL from the directory specified in *sourcePath* to the directory specified in *destPath*. This example uses **getDir** to extract the directory from *destPath*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fsl, fs2 FileSystem
  sourceFile, destPath String
endvar

sourceFile = "c:\\pdoxwin\\diveplan\\diveplan.fsl"
destPath   = "c:\\pdoxwin\\myforms\\*.*"
```

```
if fs1.findFirst(sourceFile) then      ; if the source file exists
  if fs2.findFirst(destPath) then      ; if the destination dir exists
    fs1.copy(sourceFile, fs2.getDir()) ; copy file to destination directory
  else
    msgStop(destPath, "Directory not found.")
  endIf
else
  msgStop(sourceFile, "File not found")
endIf

endmethod
```

See also

- getDrive, setDir

---

getDrive

FileSystem

Method

Returns the drive letter pointed to by the FileSystem variable.

Syntax

getDrive ( ) String

Description

Returns a string representing the drive letter pointed to by the FileSystem variable.

Example

This example calls **getDrive** to return the alias of the working directory. Then the example sets the default drive to H and calls **getDrive** again to confirm the change.

```
; setH::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  newDrive String
endvar

msgInfo("Default drive", fs.getDrive())      ; Displays :WORK:

newDrive = "H"

if fs.existDrive(newDrive) then
  if fs.setDrive(newDrive) then
    msgInfo("Default drive", fs.getDrive())   ; Displays H:
  else
    msgStop(newDrive, "Could not set drive.")
  endIf
else
  msgStop(newDrive, "Drive is not attached.")
endIf

endmethod
```

See also

- existDrive, getDir, setDrive

getFileAccessRights

FileSystem

Procedure

Reports access rights (also called file attributes) of a file.

Syntax

getFileAccessRights (const ***fileName*** String) **String**

Description

Returns a string describing access rights of a file. Return values can be one or more of the following: A, D, H, R, S, V (for archive, directory, hidden, read only, system, and volume, respectively). If the returned value is an empty string, the file has no attributes set. This procedure is similar to the **accessRights** method. However, **getFileAccessRights** does not require you to call the **findFirst** method first.

Example

This example displays in a dialog box the file attributes for the C:\CONFIG.SYS file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fileName String
endvar

fileName = "C:\\CONFIG.SYS"

msgInfo(fileName, getFileAccessRights(fileName))

endmethod
```

See also

- accessRights, setFileAccessRights

getValidFileExtensions

FileSystem

Procedure

Returns the valid file extensions for a specified object.

Syntax

getValidFileExtensions (const ***objectType*** String) String

Description

Returns a string containing the valid file extensions for the object specified in *objectType*, where *objectType* is one of: Form, Library, Report, or Script.

Example

This example displays a dialog box listing the valid file extensions for forms:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
```

```
  fx String
endVar

fx = getValidFileExtensions("Form")
msgInfo("Form file extensions:", fx)    ; displays fsl fdl

endmethod
```

See also

- getDir, getDrive, splitFullFilename

---

isDir

FileSystem

Procedure

Reports whether a specified string represents the name of a directory.

Syntax

isDir (const ***dirName*** String) Logical

Description

Returns True if *dirName* is a valid directory name; otherwise, it returns False.

Example

This example calls **isDir** to make sure that the directory specified by the variable *newDir* is valid. If it is, the call to **setDir** makes *newDir* the default directory. In this example, the value of *newDir* is hard coded, but in practice, it could be supplied by the user, read from a table, and so on.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  newDir String
endvar

newDir = "C:\\pdoxwin\\diveplan"
if isDir(newDir) then
  fs.setDir(newDir)
  msgInfo("Current directory", fs.getDir())
else
  msgStop(newDir, "Directory does not exist.")
endIf

endmethod
```

See also

- deleteDir, getDir, makeDir, setDir

---

isFile

FileSystem

Procedure

Reports whether a specified string is the name of a file in the current file system.

Syntax

isFile (const ***fileName*** String) Logical

Description

Returns True if *fileName* is a file in the current file system; otherwise, it returns False.

Example

The first example calls **isFile** and displays messages reporting whether the file specifications represent actual files.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endvar

message(isFile("c:\\dos\\chkdsk.exe")) ; displays True
sleep(1500)
message(isFile("c:\\dos\\xxxx.xxx"))   ; displays False
sleep(1500)

endmethod
```

The second example prompts the user to enter the full path and file name of a file to delete. A call to **isFile** tests if the file exists, and if it does, a call to **delete** deletes it:

```
; buttonOne::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  fileName String
endvar

fileName = "Enter full path and filename here."
fileName.view("Delete a file")

if isFile(fileName) then        ; if the specified file exists
  fs.delete(fileName)           ; delete the file
  message("File deleted.")
else
  msgStop(fileName, "File not found.")
endIf

endmethod
```

See also

- isDir

isFixed

FileSystem

Method

Reports whether a drive is fixed.

Syntax

isFixed (const ***driveLetter*** String) Logical

Description

Returns True if *driveLetter* represents a fixed (not removable or network) drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example

In the following example, drive C is the user's local hard disk, and drive H is a network drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

msgInfo("Is drive C fixed?", fs.isFixed("C"))  ; displays True
msgInfo("Is drive H fixed?", fs.isFixed("H"))  ; displays False

endmethod
```

See also

- isRemote, isRemovable

isRemote

FileSystem

Method

Reports whether a drive is remote (a network drive).

Syntax

isRemote (const ***driveLetter*** String) Logical

Description

Returns True if *driveLetter* represents a remote (network) drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example

In this example, drive H is a network (remote) drive. This code calls **existDrive** to make sure drive H is attached, then calls **isRemote** to find out if drive H is a network drive.

```
var
   h FileSystem
endVar
if h.existDrive("h") then ; if drive H is attached
   if h.isRemote() then
      msgInfo("Drive H: ", "Remote Drive")
   else
      msgInfo("Drive H:", "Not a Remote Drive.")
   endIf
else
   msgStop("Drive H", "Drive is not attached.")
endIf
```

See also

- isFixed, isRemovable

isRemovable

FileSystem

Method

Reports whether a drive is removable.

Syntax

isRemovable (const ***driveLetter*** String) Logical

Description

Returns True if *driveLetter* represents a removable drive; otherwise, it returns False. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example

In this example, drive D is a removable drive. This code calls **existDrive** to make sure drive D is attached, then calls **isRemovable** to find out if drive D is a removable drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  s  String
endVar

if fs.existDrive("D:") then ; if drive D is attached
  if fs.isRemovable("D") then
    msgInfo("Drive D: ", "Removable Drive")
  else
    msgInfo("Drive D:", "Not a Removable Drive.")
  endIf
endIf

endmethod
```

See also

- isFixed, isRemote

makeDir

FileSystem

Method

Creates a new directory.

Syntax

makeDir (const ***name*** String) Logical

Description

Creates all directories and subdirectories specified in *name.* This method returns True if successful in creating *name;* otherwise it returns False. This method also returns True if the directory already exists.

Example

The code in this example tries to create a new directory on drive M. The example displays a dialog box to report whether the method succeeded or failed.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

; this creates \New and \New\Directory etc...
l = fs.makeDir("c:\\New\\Directory\\Tree")

msgInfo("Status", iif(l, "New directory created", "makeDir Failure"))

endmethod
```

See also

- deleteDir, isDir

name

FileSystem

Method

Returns the name of a file.

Syntax

name ( ) String

Description

After a successful **findFirst** or **findNext**, **name** returns the name of the file whose name matches the pattern.

Example

This example calls **findFirst** and **findNext** to find the tables in the current directory, then calls **name** to create a pop-up menu listing the file names.

```
; showName::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  p PopUpMenu
  tv TableView
  choice, path String
endvar

if fs.findFirst("*.db") then    ; if a *.db file exists
  p.addStaticText("Tables")     ; create a pop-up menu
  p.addSeparator()
  p.addText(fs.name())          ; use file names in pop-up
  while fs.findNext()
    p.addText(fs.name())
  endWhile
  choice = p.show()             ; show the menu
  if not choice.isBlank() then  ; if user selected a table
    tv.open(choice)             ; display the selected table
  endif
endIf

endmethod
```

See also

- findFirst, findNext, fullName

privDir

FileSystem

Procedure

Returns the name of the user's private directory.

Syntax

privDir ( ) String

Description

Returns a string containing the full DOS path (including the drive ID letter) of the user's private directory.

Each user must have a private directory where temporary tables are stored. It can be on a network or a local drive.

Example

This example calls **privDir** to display the path to :PRIV: in the status bar.

```
method pushButton(var eventInfo Event)
   message("Your private directory is: ", privDir())
endmethod
```

See also

- getDir, startUpDir, workingDir

rename

FileSystem

Method

Renames a file.

Syntax

rename (const ***oldName***, String ***newName*** String) Logical

Description

Changes the name of file *oldName* to *newName*. If *newName* is used by another file, the method fails; it does not overwrite the existing file. **rename** returns True if it succeeeds; otherwise, it returns False. This method is independent of **findFirst** and **findLast**.

Example

The following example searches the current directory for the file specified in the variable *oldName*. If it exists, the call to **rename** tries to rename it. A dialog box appears to report errors, if any.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  oldName, newName String
endvar

oldName = "memo14.txt"
newName = "memo14.bak"

if fs.findFirst(oldName) then
```

```
  if not fs.rename(oldName, newName) then
    msgStop("Could not rename file", newName + " already exists.")
  endIf
else
  msgStop(oldName, "File not found.")
endIf

endmethod
```

See also

- copy, delete, findFirst, findNext

setDir

FileSystem

Method

Sets the directory path for a FileSystem variable.

Syntax

setDir (const ***name*** String) Logical

Description

Sets the directory path to *name* for a FileSystem variable. Compare this method to **setDrive**, which sets the default drive.

Example

This example calls **isDir** to find if the directory specified in the variable *newDir* is valid. If it is, the code calls **setDir** to make *newDir* the default directory.

```
method pushButton(var eventInfo Event)
var
   fs FileSystem
   newDir String
endvar

newDir = "c:\\pdoxwin\\mine\\zap"

if isDir(newDir) then
   fs.setDir(newDir)
else
   msgStop(newDir, "Not a valid directory.")
endIf

message(fs.getDir()) ; displays \pdoxwin\mine\zap
endmethod
```

See also

- getDir, setDrive

setDrive

FileSystem

Method

Makes a specified drive the default drive.

Syntax

setDrive (const ***name*** String) Logical

Description

Returns True if successful in setting the default drive to *name*; otherwise, it returns False. You can specify *name* using a letter ("C") or a letter and a colon ("C:"), or an alias (":MAST:").

Example

The first example calls **view**, defined for the String type, to display a dialog box and prompt the user for input. If the user enters the ID letter of a valid drive, the call to **setDrive** makes the ID letter the default drive.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  newDrive String
endvar

newDrive = "Enter drive ID or alias here."
newDrive.view("Change default drive.") ; prompt user for input

if fs.existDrive(newDrive) then
  fs.setDrive(newDrive)
else
  msgStop(newDrive, "Drive not available.")
endIf

endmethod
```

The next example shows how to use an alias with **setDrive**. It assumes that the alias :MAST: has already been defined.

```
; setDrive::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
endVar

fs.setDrive(":MAST:")

endmethod
```

See also

- getDrive, setDir

setFileAccessRights

FileSystem

Procedure

Sets access rights (also called attributes) of a file.

Syntax

setFileAccessRights **(** const ***fileName*** String **,** const ***rights*** String **)** Logical

Description

Sets the attributes (access rights) of *fileName* to the attributes specified in *rights*. *rights* is a string that evaluates to one or more or the following: A, D, H, R, S, V (for archive, directory, hidden, read only,

system, and volume, respectively). If you specify an empty string ("") for *rights*, all attributes for *fileName* are removed. You don't have to declare a FileSystem variable (or use the **findFirst** method) before calling **setFileAccessRights**.

Example

This example sets file attributes for C:\CONFIG.SYS to read only ("R") and hidden ("H").

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fileName String
endvar

fileName = "C:\\CONFIG.SYS"

; set file attribute for CONFIG.SYS to read only and hidden
if setFileAccessRights(fileName, "RH") then
  ; if successful, display a message with the current attributes
  message (fileName + " attributes set to " +
           getFileAccessRights(fileName))
else
  ; otherwise, the procedure failed
  message("Can't set file attributes for " + fileName)
endif

endmethod
```

See also

- accessRights, getFileAccessRights

---

size

FileSystem

Method

Returns the size of a file.

Syntax

size ( ) LongInt

Description

After a successful **findFirst** or **findNext**, **size** returns the number of bytes in a found file.

Example

This example creates a DynArray containing the file names and sizes of the Paradox tables in the current directory. The call to **view**, defined for the DynArray type, displays the information in a dialog box.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  da DynArray[] LongInt
endvar

if fs.findFirst("*.db") then
```

```
  da[fs.name()] = fs.size()
  while fs.findNext()
    da[fs.name()] = fs.size()
  endWhile
  da.view("Names and sizes")
else
  msgStop("*.db", "file not found.")
endIf

endmethod
```

See also

- freeDiskSpace, totalDiskSpace

splitFullFileName

FileSystem

Procedure

Breaks a full path name into its component parts.

Syntax

1. **splitFullFileName** (const ***fullFileName*** String, var ***components*** DynArray[] String)
2. **splitFullFileName** (const ***fullFileName*** String, var ***driveName*** String, var ***pathName*** String, var ***fileName*** String, var ***extensionName*** String)

Description

Divides a full file path (obtained using **fullName**) into its component parts. You can use syntax 1 to store the results in a DynArray, or use syntax 2 to store the results in separate variables.

If you use syntax 1, you must declare the DynArray before you call **splitFullFileName**. The DynArray has the following keys: DRIVE, EXT, NAME, and PATH.

Example

The first example calls **fullName** to get the full name of the first form listed in the current directory. Then it calls **splitFullFileName** to split the name into its component parts and store them in a DynArray, then it calls **view** to display the DynArray.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  splitName DynArray[] anytype
  fullFileName String
endVar

; if the customer.db file is in the sample directory
if fs.findFirst("c:\\pdoxwin\\sample\\customer.db") then

  ; store the full file name to a variable
  fullFileName = fs.fullName()

  ; split file name into parts and store them in a DynArray
```

```
  splitFullFileName(fullFileName, splitName)

  ; display the component parts
  splitName.view("Split name")
endIf

endmethod
```

The next example calls **splitFullFileName** to split the full name of a form into its component parts, then displays the path and the file name (without an extension) in dialog boxes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  driveName, pathName, fileName, extName String
endVar

if fs.findFirst("*.fsl") then
  splitFullFileName(fs.fullName, driveName, pathName, fileName, extName)
  pathName.view("Path name") ; displays the path
  fileName.view("File name") ; displays the filename (no extension)
endIf

endmethod
```

See also

❒ fullName

---

startUpDir

FileSystem

Procedure

Returns a string containing the path to the user's start-up directory.

Syntax

startUpDir ( ) String

Description

Returns a string containing the full path (including the drive ID letter) to the user's start-up directory, the directory from which Paradox was started.

Example

This example displays a dialog box listing the path to the directory from which the user started Paradox:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

msgInfo("Start-up directory", startUpDir())

endmethod
```

See also

❒ privDir, workingDir

time

FileSystem

Method

Returns the time and date a file was last modified.

Syntax

time ( ) DateTime

Description

Returns a DateTime value representing the time and date of the last modification to a file.

Example

This example calls **time** to get the time and date of the most recent changes to the *Customer* table. Then, statements compare the modification date with today's date and report the results.

```
method pushButton(var eventInfo Event)
var
   fs FileSystem
endvar

if fs.findFirst("customer.db") then
   if fs.time() < DateTime(today()) then
      message("old version")
   else
      message("new version")
   endif
endIf
endmethod
```

See also

- fullName, splitFullFilename

totalDiskSpace

FileSystem

Method

Returns the capacity of a drive.

Syntax

totalDiskSpace (const ***driveLetter*** String) LongInt

Description

Returns the total number of bytes drive *driveLetter* can hold. You can specify *driveLetter* using a letter ("C") or a letter and a colon ("C:").

Example

This example calls **totalDiskSpace** and **freeDiskSpace** to calculate the amount of space in use. It stores the information in a DynArray, then calls the **view** method defined for the DynArray type to display the information in a dialog box.

```
; spaceUsed::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  da DynArray[] LongInt
```

```
endvar

da["Total space"] = fs.totalDiskSpace("C")
da["Free space"] = fs.freeDiskSpace("C")
da["Space in use"] = da["Total space"] - da["Free space"]
da.view("Drive C")

endmethod
```

See also

- freeDiskSpace

windowsDir

FileSystem

Procedure

Returns the path to the WINDOWS directory.

Syntax

windowsDir ( ) String

Description

Returns the path to the WINDOWS directory.

Example

This example reads the file WIN.INI from drive B and copies it to the WINDOWS directory on the default drive:

```
; copyWinIni::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  fileName, destName String
endvar

fileName = "\\win.ini"

fs.setDrive("B")
if fs.findFirst(fileName) then
  destName = windowsDir() + fileName
  fs.copy(fileName, destName)
endIf

endmethod
```

See also

- privDir, startUpDir, windowsSystemDir, workingDir

windowsSystemDir

FileSystem

Procedure

Returns the path to the Windows system directory.

Syntax

windowsSystemDir ( ) String

Description

Returns the path to the Windows system directory.

Example

This example reads the file SPECIAL.DRV from drive B and copies it to the Windows system directory on the default drive:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  fileName, destName String
endvar

fileName = "\\special.drv"

fs.setDrive("B")
if fs.findFirst(fileName) then
  destName = windowsSystemDir() + fileName
  fs.copy(fileName, destName)
endIf

endmethod
```

See also

❒ windowsDir

workingDir

FileSystem

Procedure

Returns the name of the current working directory.

Syntax

workingDir ( ) String

Description

Returns the name of the current working directory (:WORK:).

Example

This example displays a dialog box containing the path to the current working directory:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

message("Working directory is: " + workingDir())

endmethod
```

See also

❒ privDir, windowsDir, windowsSystemDir

Form

Form

action	getPosition	mouseExit
attach	getTitle	mouseMove
bringToTop	hide	mouseRightDouble
close	hideSpeedBar	mouseRightDown
create	isMaximized	mouseRightUp
delayScreenUpdates	isMinimized	mouseUp
deliver	isSpeedBarShowing	moveTo
design	isVisible	moveToPage
disableBreakMessage	keyChar	open
dmAddTable	keyPhysical	openAsDialog
dmGet	load	postAction
dmHasTable	maximize	run
dmPut	menuAction	save
dmRemoveTable	methodDelete	setPosition
enumSource	methodGet	setTitle
enumSourceToFile	methodSet	show
enumTableLinks	minimize	showSpeedBar
enumUIOjbectNames	mouseDouble	wait
enumUIObjectProperties	mouseDown	windowClientHandle
formCaller	mouseEnter	windowHandle
formReturn		

A Form variable provides a handle for working with a Paradox form. Form type methods let you

- Load a form in a design window and save a design
- Open and close a form
- Attach to an open form
- Add and remove tables in a form's data model
- Enumerate object names, properties, and source code for methods
- Determine and change the position of a form, as well as maximize or minimize the form
- Send events to a form
- Get and set methods for a form

The Form type is the base type from which the other Display manager types are derived. Many of the methods listed in this section are used by the Application, Report, and TableView types, too. For more information and examples, refer to Chapter 8 in the *ObjectPAL Developer's Guide*.

action

Form

Method/Procedure

Performs an action command.

Syntax

action (const ***actionId*** SmallInt) Logical

Description

Performs the action represented by the constant *actionId*. This **action** method constructs and sends an ActionEvent to the built-in **action** method of the object you specify. ObjectPAL provides constants for *actionId;* see one of the categories beginning with Action in the Constants dialog.

You can also use **action** to send a user-defined action constant to a built-in **action** method. User-defined action constants are simply integers that don't interfere with any of ObjectPAL's constants. You can use them to signal other parts of an application. For instance, assume that the Const window for a form declares a constant named *myAction*. In the built-in **action** method for a page on the form, you might check the value of every incoming ActionEvent (with the **id** method); if the value is equal to *myAction*, you can respond to that action accordingly. Paradox's default response for user-defined action constants is simply to pass the action to the **action** method. For more information on defining constants, see Chapter 6 in the *ObjectPAL Developer's Guide.*

This **action** method is distinct from the built-in **action** method for a form or for any other UIObject. The built-in **action** method for an object *responds* to an action event; this method *causes* an action event.

Note When you call the **action** method as a procedure, ObjectPAL must make an educated guess about which object you want the action to affect. The event bubbles through the containership heirarchy until either the event reaches a container that can handle the action or the event reaches the form. If the event reaches the form, and the action is a data action, the form sends the event to the master table for the form.

Example

In this example, a form named *Sitenote* contains field objects bound to the *Sites* table. The current form contains a button named *openEditSites;* the **pushButton** method for *openEditSites* opens *Sitenote,* starts Edit mode, and waits for *Sitenote* to be closed:

```
; openEditSites::pushButton
method pushButton(var eventInfo Event)
var
  siteForm   Form
endVar
siteForm.open("Sitenote.fsl")  ; open Sitenote
siteForm.action(DataBeginEdit) ; start Edit mode on siteForm
```

```
message("To return, close Sitenote form.")
siteForm.wait()             ; this form will be inactive until
                            ; Sitenote returns
siteForm.close()            ; this form must close Sitenote
endmethod
```

See also

- The discussion of actions in Chapter 2
- action in the UIObject type
- User-defined constants in Chapter 6 in the *ObjectPAL Developer's Guide*

attach

Form

Method

Associates a Form variable with an open form.

Syntax

attach ([const ***formName*** String]) Logical

Description

Associates a Form variable with an open form. You can use *formName* to specify a form's current title, or you can omit *formName* to attach to the form where **attach** is executing.

Example

In this example, a form has two buttons: *openSites* and *attachToSites*. The **pushButton** method for *openSites* takes care of opening the *Sitenote* form. The **pushButton** method for *attachToSites* attaches the form variable *sitesForm* to the open form by way of the form's current title. In this case, the form title wasn't changed, so *attachToSites* can attach to *Sitenote* using the default title. Once attached, the **pushButton** method uses the *sitesForm* handle to minimize, maximize, and restore *Sitenote.*

The following code is attached to the **pushButton** method for *openSites*:

```
; openSites::pushButton
method pushButton(var eventInfo Event)
var
  sitesForm Form
endVar
sitesForm.open("Sitenote.fsl")   ; open Sitenote, default
                                 ; title will be "Form : SITENOTE.FSL"
endmethod
```

This code is attached to the **pushButton** method for *attachToSites*:

```
; attachToSites::pushButton
method pushButton(var eventInfo Event)
var
  sitesForm  Form
endVar
sitesForm.attach("Form : SITENOTE.FSL")  ; attach to Sitenote by its title
```

Form

```
; Note that this won't work:   sitesForm.attach("Sitenote")

; cycle through sizes
sitesForm.minimize()                ; minimize the form
sleep(2000)                         ; pause
sitesForm.maximize()                ; maximize the form
sleep(2000)                         ; pause
sitesForm.show()                    ; restore to original size
endmethod
```

See also

- getTitle, open, setTitle

bringToTop

Form

Beginner

Method/Procedure

Brings the window to the top of the display stack and makes it active.

Syntax

bringToTop ( )

Description

When several windows are displayed they seem to overlap, giving an appearance of layers. Use **bringToTop** to display a window at the top of the stack, not overlapped by any other windows. **bringToTop** makes a form the active window.

If a **hide** statement has made a form invisible, **bringToTop** makes it visible again.

Example

In the following example, the **pushButton** method for a button named *openSeveral* opens the *Sitenote* form, then opens a table window for the *Orders* table. The table window, *orderTV*, opens over the *Sitenote* form, *siteForm*. The method pauses for a few seconds, then makes *siteForm* the topmost layer:

```
; openSeveral::pushButton
method pushButton(var eventInfo Event)
var
   siteForm  Form
   orderTV   TableView
endVar
siteForm.open("Sitenote.fsl")     ; opens Sitenote form
orderTV.open("orders")         ; opens Orders over Sitenote
message("About to make the Sitenote form the highest layer.")
beep()
sleep(5000)                    ; pause
siteForm.bringToTop()          ; make Sitenote highest layer
endmethod
```

See also

- hide, isVisible, show

close

Beginner

Form

Method/Procedure

Closes a window.

Syntax

1. (Method) **close** ( )
2. (Procedure) **close** ([const ***returnValue*** AnyType])

Description

Closes a form as if the user had chosen Close from the Control menu. When you call **close** as a procedure, you can specify the value to be returned in *returnValue*. The value is returned to the calling form, if there is one.

Example

In this example, a form contains a button called *openAndClose*. The *Sitenote* form contains a button called *goBackButton*. The **pushButton** method for *goBackButton* uses **formReturn** to return control to the calling form:

```
; goBackButton::pushButton
method pushButton(var eventInfo Event)
formReturn()    ; return control to calling form or object
endmethod
```

The following code is attached to the **pushButton** method for a button on the current form named *openAndClose*. This method opens the *Sitenote* form to *siteForm*, then waits for it to return. Once *siteForm* returns (because the user clicked *goBackButton*), this method displays a message, pauses, then closes *siteForm*:

```
; openAndClose::pushButton
method pushButton(var eventInfo Event)
var
   siteForm Form
endVar
siteForm.open("Sitenote.fsl")
message("To return, press the Go Back button.")
siteForm.wait()             ; wait for the form to return
msgInfo("Status", "Closing the Sitenote form in three seconds.")
sleep(3000)                 ; pause
siteForm.close()            ; close Sitenote form
endmethod
```

See also

- open

create

Form

Method

Creates a blank form in a design window.

Form

Syntax

create ( ) Logical

Description

Creates a blank form and leaves it in the Form Design window. You can use the UIObject type methods **create** and **methodSet** to place objects in the new form and attach methods to them. Use **run** to run a form.

Example

In this example, the **pushButton** method for a button named *createAForm* creates a new form with the **create** method and sets the value of the new form's **mouseUp** method with **setMethod**. The **pushButton** method for *createAForm* then saves the new form to a file named NEWHELLO.FSL, runs the form, and calls the new form's **mouseUp** method (supplying the correct arguments). The **mouseUp** method for the *Newhello* form opens a dialog box that displays "Hello". Once the dialog box is closed (by the user), the **pushButton** method for *createAForm* closes the *Newhello* form.

```
; createAForm::pushButton
method pushButton(var eventInfo Event)
var
   newForm Form
endVar
newForm.create()              ; create a new blank form (a design window)
newForm.methodSet("mouseUp",  ; set the mouseUp method for the form
"method mouseUp(var eventInfo MouseEvent)
msgInfo(\"Greetings\", \"Hello\")
endMethod")                   ; backslashes delimit embedded quotes
newForm.save("newhello")      ; save the form
newForm.run()                 ; run the new form
                              ; call the mouseUp method for the form
newForm.mouseUp(100, 100, LeftButton )  ; dialog box displays "Hello"
newForm.close()               ; close the form
endmethod
```

See also

- design, load, open
- create, methodGet, methodSet in the UIObject type

delayScreenUpdates

Form

Procedure

Turns delayed screen updates on or off.

Syntax

delayScreenUpdates (const ***yesNo*** Logical)

Description

Postpones redrawing areas of the screen but doesn't lock the screen. You must specify Yes or No. Specifying Yes delays screen updates (redraws) until an operation is complete. Specifying No allows screen updates to occur normally.

For some operations, you won't notice a difference when **delayScreenUpdates** is set to Yes. This is especially true if the application is running on a fast machine.

Example

The following two methods override the **pushButton** methods for their respective buttons. The *drawOneByOne* button draws a number of boxes without changing **delayScreenUpdates**. The *drawAllAtOnce* button draws the same number of boxes, to a different location, but first sets **delayScreenUpdates** to Yes. If you run this code, you'll see the boxes created by *drawOneByOne* appear one at a time, but still rapidly. The boxes created by *drawAllAtOnce* are created behind the scenes—which causes a short pause—then appear all at the same time.

```
; drawOneByOne::pushButton
method pushButton(var eventInfo Event)
var
  ui UIObject
endVar

; delayScreenUpdates(No) is the default
; Create and display a set of boxes, showing them as
; they're created.
for i from 750 to 2550 step 300
  for j from 750 to 2550 step 300
    ui.create(boxTool, i, j, 150, 150)
    ui.Color = Blue
    ui.Visible = Yes
  endfor
endfor
endmethod
```

The *drawAllAtOnce* button on the same form creates the same number of boxes, but does so with **delayScreenUpdates** set to Yes. On very fast machines, you still may not be able to see the difference.

```
; drawAllAtOnce::pushButton
method pushButton(var eventInfo Event)
var
  ui UIObject
endVar

delayScreenUpdates(Yes)
; This code will create all boxes, then display
; them all at once.
for i from 4950 to 6750 step 300
  for j from 750 to 2550 step 300
    ui.create(boxTool, i, j, 150, 150)
    ui.Color = Red
    ui.Visible = Yes
  endfor
endfor
; reset to default
delayScreenUpdates(No)

endmethod
```

See also

- sleep in the System type

deliver

Form

Method

Delivers a form.

Syntax

deliver ( ) Logical

Description

Behaves like Form | Deliver. This method saves a copy of a form with an .FDL extension, which prevents users from editing the form in Form Design window. Users can open the form only in a Form window. Switching to the Form Design window on an open, delivered form is also prohibited. For more information about saving and delivering forms, refer to the *User's Guide*.

Example

In this example, the *createDeliver* button creates a new form, saves it to the name Newhello, then delivers it (which saves a version as NEWHELLO.FDL). When the method attempts to load the form in a Form Design window, **load** returns False.

```
; createDeliver::pushButton
method pushButton(var eventInfo Event)
var
   newForm Form
endVar
newForm.create()              ; create a new blank form (a design window)
newForm.save("newhello")      ; save the form
newForm.deliver()             ; deliver the newly created form
newForm.close()               ; close the form
if NOT newForm.load("newhello.fdl") then  ; load will return False
  msgStop("Sorry", "Can't load a delivered form.")
endif
endmethod
```

See also

- save

design

Form

Method

Switches a running form to the Form Design window.

Syntax

design ( ) Logical

Description

Switches a running form to the Form Design window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *ObjectPAL Developer's Guide*.

Use **run** to run a form from the Form Design window.

Note

Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

Example

See the example for create.

See also

- create, open, run

disableBreakMessage — Form

Procedure

Prevents program interruption by *Ctrl+Break*.

Syntax

disableBreakMessage (const ***yesNo*** Logical) Logical

Description

Lets you prevent or allow the user to interrupt a running program with *Ctrl+Break*.

Example

In this example, assume a form contains a table frame bound to the *Orders* table. The following code prevents the loop from being interrupted by a *Ctrl+Break*.

```
; throughTable::pushButton
method pushButton(var eventInfo Event)
; just a loop to test Ctrl-breaking out of
disableBreakMessage(Yes)     ; don't allow a Ctrl+Break
while NOT ORDERS.atLast()
  ORDERS.action(DataNextRecord)
endwhile
endmethod
```

See also

- Enable Ctrl+Break to Debugger command in Chapter 4 in the *ObjectPAL Developer's Guide*

dmAddTable — Form

Method/Procedure

Adds a table to a form's data model.

Syntax

dmAddTable (const ***tableName*** String) Logical

Description

Adds the table *tableName* to a form's data model.

Example

In this example, a form contains a button named *toggleSites,* and a list field named *showSiteNames*. The list data for the *showSiteNames* field is set with the DataSource property of its list object, *ListNames.* The **pushButton** method for *toggleSites* checks to see if the *Sites* table is in the data model for the form. If so, the reference to *Sites* is removed from the DataSource property of *ListNames,* then *Sites* is removed from the data model. Otherwise, the *Sites* table is added to the data model, and the DataSource property of *ListNames* is set to the *Site Name* field of *Sites.* (To reference a table's field in the DataSource property of a list, the table referenced must be part of the data model for the form.)

This is the code for the **pushButton** method of *toggleSites*:

```
; toggleSites::pushButton
method pushButton(var eventInfo Event)
; toggle Sites.db in and out of the data model
if dmHasTable("Sites") then    ; is Sites in data model?
  ; if so, remove dependencies, then remove table
  ; remove Sites as source from showSiteNames.ListNames
  showSiteNames.ListNames.DataSource = ""
  showSiteNames.Visible = False
  ; remove Sites from the data model
  dmRemoveTable("Sites")
  whichTable = ""
else
  ; if not already in data model, then add Sites
  dmAddTable("Sites")
  ; set the data for the list from the Sites table
  showSiteNames.ListNames.DataSource = "[Sites.Site Name]"
  showSiteNames.Visible = True
  whichTable = "Sites"
endIf

endmethod
```

See also

- dmHasTable, dmRemoveTable

---

dmGet

Form

Method/Procedure

Retrieves a field value from a table in the data model.

Syntax

dmGet (const ***tableName*** String, const ***fieldName*** String, var ***datum*** AnyType) Logical

Description

Provides access to table data in a form's data model. **dmGet** writes to *datum* a field value from a specified table. The table specified by *tableName* must be one of the tables in the form's data model. *fieldName* must be a field in *tableName.*

Example

In the following example, a form contains a table frame bound to the *Sites* table. The table frame contains only two fields: Site No and Site Name. The **pushButton** method for a button named *getHighlight* uses **dmGet** to find the value of the Site Highlight field for the current record. The method then displays the Site Highlight value in a dialog box and asks the user whether to change the value. If the user answers "Yes" in the dialog box, the method shows the original value for Site Highlight in a dialog box and prompts the user for a new value. The method then uses **dmPut** to write the changed value back to the *Sites* table:

```
; getHighlight::pushButton
method pushButton(var eventInfo Event)
var
  siteHighlight AnyType
  qAnswer       String
endVar
; get the value in the Site Highlight field for the current record
if dmGet("Sites", "Site Highlight", siteHighlight) then
 ; show the highlight and ask the user whether to change it
 qAnswer = msgQuestion("Change Highlight?",
              "At site " + SITES.Site_Name +
              " the highlight is " +
              String(siteHighlight) + ". Change highlight?")
 if qAnswer = "Yes" then
   ; check for Edit mode
   if thisForm.Editing  True then
     action(DataBeginEdit)
   endif
   ; ask user to replace existing highlight value in View dialog box
   siteHighlight.view("Enter a new highlight:")
   ; write the changed highlight back to the Site Highlight field
   dmPut("Sites", "Site Highlight", siteHighlight)
 endif
else
  msgStop("Sorry", "Couldn't find the highlight for this site.")
endif
endmethod
```

See also

- dmPut

dmHasTable

Form

Method/Procedure

Reports whether a table is part of a form's data model.

Syntax

dmHasTable (const ***tableName*** String) Logical

Description

Reports whether *tableName* is a table associated with a form.

Example

See the example for **dmAddTable** for an illustration of how to use **dmHasTable** as a procedure.

Form

This example shows how **dmHasTable** is used as a method. The **pushButton** method for a button named *isStockInDM* works with the form specified by the variable *thatForm*. This method opens the *Ordentry* form, then checks to see if the *Stock* table is in *thatForm's* data model. If not, the *Stock* table is added to the data model for *thatForm*:

```
; isStockInDM::pushButton
method pushButton(var eventInfo Event)
var
  thatForm Form
endVar
thatForm.load("Ordentry")                    ; open ORDENTRY form
if not thatForm.dmHasTable("stock") then    ; is Stock in data model
  msgInfo("Status", "Adding Stock to data model for form.")
  thatForm.dmAddTable("stock")               ; if not, add it
  thatForm.save()
else
  msgInfo("Status", "Stock is already in data model for form.")
endif
thatForm.close()
endmethod
```

See also

❒ dmAddTable, dmRemoveTable

dmPut

Form

Method/Procedure

Writes data to a table in a form's data model.

Syntax

dmPut (const ***tableName*** String, const ***fieldName*** String, const ***datum*** AnyType) Logical

Description

Provides access to table data in a form's data model. **dmPut** writes *datum* to a field in a specified table. The table specified by *tableName* must be one of the tables in the form's data model. *fieldName* must be a field in *tableName*.

Example

See the example for dmGet.

See also

❒ dmGet

dmRemoveTable

Form

Method/Procedure

Removes a table from a form's data model.

Syntax

dmRemoveTable (const ***tableName*** String) Logical

Description

Removes *tableName* from the list of tables associated with a form. Any objects on the form that depend on the table will be undefined when the table is removed.

Example

See the example for dmAddTable.

See also

- dmAddTable, dmHasTable

enumSource

Form

Method/Procedure

Creates a table listing the methods for each object in a form.

Syntax

enumSource (const ***tableName*** String
[, const ***recurse*** Logical]) Logical

Description

Creates a Paradox table listing every object you've written a method for, along with the ObjectPAL source code for the method. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the created table is

Field Name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

The Object field contains the full path name of the object.

If *recurse* is False, this method returns the method definitions for the form only. To include source code for methods on all objects contained by the form, *recurse* should be True.

Example

In this example, a form contains a button named *getSource*. The **pushButton** method for *getSource* uses **enumSource** as a procedure to enumerate the source code for the current form to a table named TEMPSORC.DB. The method then opens a table window for the *Tempsorc* table and waits for the user to close it. Then the method opens the *Sitenote* form to *siteForm*, uses **enumSource** as a method to write the source code for *siteForm* to a table named SITESORC.DB, and views the table:

```
; getSource::pushButton
method pushButton(var eventInfo Event)
var
  siteForm   Form
  tempTable  TableView
endVar
;enumSource("tempsorc.db", True) ; writes all source for the
                                ; current form to TEMPSORC.DB
;tempTable.open("tempsorc.db")
;tempTable.wait()
siteForm.open("Sitenote.fsl")       ; open another form
; write source for siteForm to SITESORC.DB
siteForm.enumSource("sitesorc.db", True)
siteForm.close()                ; close the form
tempTable.open("sitesorc.db")   ; view the new table
tempTable.wait()                ; wait for the user to close
                                ; the table
endmethod
```

See also

- enumSourceToFile, enumUIObjectNames, enumUIObjectProperties

---

enumSourceToFile

Form

Method/Procedure

Creates a file listing the methods for each object in a form.

Syntax

enumSourceToFile (const ***fileName*** String [, const ***recurse*** Logical]) Logical

Description

Creates a text file listing every object you've written a method for, along with the ObjectPAL source code for the method. Use the argument *fileName* to specify a name for the file. If *fileName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *fileName*; if no alias or path is specified, Paradox creates *fileName* in the working directory (:WORK:).

If *recurse* is False, this method returns the method definitions for the form only. To include source code for methods on all objects contained by the form, *recurse* should be True.

Example

The following code is attached to the **pushButton** method for a button named *getSourceToFile*. This method writes all the source code for the current form to TEMPSORC.TXT. The method then opens the *Sitenote* form and writes all the code for that form to a file named SITESORC.TXT:

```
; getSourceToFile::pushButton
method pushButton(var eventInfo Event)
var
  siteForm   Form
```

```
endVar
enumSourceToFile("tempsorc.txt", True) ; writes all source for the
                                       ; current form to TEMPSORC.TXT

siteForm.open("Sitenote.fsl")            ; open another form
; write source for siteForm to SITESORC.TXT
siteForm.enumSourceToFile("sitesorc.txt", True)
siteForm.close()                       ; close the form
endmethod
```

See also

❐ enumSource, enumUIObjectNames, enumUIObjectProperties

enumTableLinks

Form

Method/Procedure

Creates a table listing the tables linked to a form.

Syntax

enumTableLinks (const ***tableName*** String) Logical

Description

Creates a Paradox table listing the names of tables linked to a form and the types of links. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of tableName is:

Field Name	Type	Size
Name	A	81*
Link	A	81*
LinkType	A	24*

Example

In this example, the **pushButton** method for a button named *showTableLinks* writes table links for the current form to a table named TEMPLINK.DB. The method then opens the *Sitenote* form, and writes the table links for that form to a table named SITENOTE.DB:

```
; showTableLinks::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
enumTableLinks("templink.db")         ; lists links to current form
tempTable.open("templink")
tempTable.wait()
tempTable.close()
siteForm.open("Sitenote.fsl")
siteForm.enumTableLinks("Sitenote.db") ; lists links to siteForm
siteForm.close()
```

```
tempTable.open("Sitenote.fsl")
tempTable.wait()
tempTable.close()
endmethod
```

See also

- enumUIObjectNames, enumUIObjectProperties
- enumRefIntStruct in the TCursor type

enumUIObjectNames

Form

Method

Creates a table listing the UIObjects contained in a form.

Syntax

enumUIObjectNames (const ***tableName*** String) Logical

Description

Creates a Paradox table listing the name and type of each object contained in a form. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of tableName is:

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

The ObjectName field includes the entire path name of the object.

Example

In this example, the **pushButton** method for a button named *getObjectNames* opens the *Sitenote* form and enumerates all the object names on the form to a table named *Siteobjs*. The method then opens the *Siteobjs* table and waits for the user to close it:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
if siteForm.open("Sitenote.fsl") then         ; open the form
  siteForm.enumUIObjectNames("siteobjs.db") ; write object names
                                            ; SITEOBJS.DB
  siteForm.close()                          ; close the form
  tempTable.open("siteobjs")                ; open the new table
  tempTable.wait()                          ; wait for return
  tempTable.close()                         ; close after return
endIf
endmethod
```

See also

- enumSource, enumSourceToFile, enumUIObjectProperties

enumUIObjectProperties

Form

Method

Creates a table listing the properties of each UIObject contained in a form.

Syntax

enumUIObjectProperties (const ***tableName*** String) Logical

Description

Creates a Paradox table listing the name, property name, and property value of each object contained in a form. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails.

The structure of *tableName* is

Field Name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyType	A	48
PropertyValue	A	255

Example

In this example, the **pushButton** method for a button named *getProps* writes the properties for all objects contained by the current form to a table named *Tempprop*:

```
; getProperties::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
  tempTable TableView
endVar
if siteForm.open("Sitenote.fsl") then
  message("Enumerating properties to Siteprop table.")
  siteForm.enumUIObjectProperties("siteProp.db")
  tempTable.open("siteprop")
  message("Close the table to continue.")
  tempTable.wait()
  tempTable.close()
endIf
; to enumerate objects for current form, use the UIObject
; type method enumUIObjectProperties
; thisForm is the object ID for current form
message("Enumerating properties to Tempprop table.")
thisForm.enumUIObjectProperties("tempprop.db")
tempTable.open("tempprop")
message("Close the table to continue.")
tempTable.wait()
tempTable.close()
endmethod
```

See also

- enumUIObjectNames
- enumUIObjectProperties in the UIObject type

formCaller

Form

Procedure

Creates a handle to the calling form.

Syntax

formCaller (var ***caller*** Form) Logical

Description

Assigns the handle of the current form's calling form to *caller*, if the form is in a **wait**. If the current form was not opened by another form, and the form that opened the current form is not waiting upon the current form, the method returns False, and *caller* is unassigned.

Example

In this example, the **pushButton** method for *whoCalledMe* finds out which form called the current form:

```
; callOtherForm::pushButton  (calling form)
method pushButton(var eventInfo Event)
var
  siteForm  Form
endVar
siteForm.open("sitenote.fsl")  ; open siteForm
siteForm.wait()                ; wait for siteForm to return
siteForm.close()               ; close siteForm
endmethod
```

This is the code for *whoCalledMe* on the current form.

```
; whoCalledMe::pushButton
method pushButton(var eventInfo Event)
var
  myCaller     Form
  callerTitle  AnyType
endVar
if formCaller(myCaller) then       ; try to get a handle to
                                   ; the calling form
  callerTitle = myCaller.getTitle() ; get the form's title
  msgInfo("FYI", "I was called by: \n" + callerTitle)
endif
endmethod
```

See also

- formReturn, wait

formReturn

Form

Procedure

Returns control to a suspended method.

Syntax

formReturn ([const ***returnValue*** AnyType])

Description

When a form has been called by **wait**, the calling method suspends execution until **formReturn** returns control. You can choose to return a value to the calling form in *returnValue.*

If no other form is waiting for the current form, **formReturn** closes the current form.

Example

This example consists of three methods. The **pushButton** method for *openDialog* opens another form as a dialog box and waits for it to return a value. The other two methods are attached to buttons in the dialog box form. They use **formReturn** to return control and values to the calling form.

```
; openDialog::pushButton
method pushButton(var eventInfo Event)
var
  dlgForm     Form
  whichButton String
endVar
if dlgForm.openAsDialog("foforet2", WinStyleDefault,
                        1440, 1440, 7200, 5760) then
  ; waits until dlgForm calls formReturn or is closed
  ; returned value is stored to whichButton
  ; return value is cast to a String so that it will be correct
  ; type even if user closes dialog box from the Control menu
  whichButton = String(dlgForm.wait())
  dlgForm.close()
  msgInfo("Button pressed", whichButton)
else
  msgStop("Stop", "Couldn't open the form.")
endIf
endmethod
```

This method is attached to **pushButton** method for *OKButton* in *dlgForm.* It returns a value of "OK" when it returns control to the method that called **wait**:

```
; OKButton::pushButton
method pushButton(var eventInfo Event)
formReturn("OK")      ; return "OK" to calling form
endmethod
```

This method is attached to *cancelButton* in *otherForm.* It returns a value of "Cancel" when it returns control to the method that called **wait**.

```
; cancelButton::pushButton
method pushButton(var eventInfo Event)
formReturn("Cancel")  ; return "Cancel" to calling form
endmethod
```

See also

- formCaller, wait

getPosition

Form

Method/Procedure

Reports the position of a window onscreen.

Syntax

getPosition (var ***x*** LongInt, var ***y*** LongInt, var ***w*** LongInt, var ***h*** LongInt)

Description

Writes the position of a window onscreen to position and size arguments. The arguments *x* and *y* contain the horizontal and vertical coordinates of the upper left corner of the form (in twips), and *w* and *h* contain the width and height (in twips).

Example

In this example, the **pushButton** method for *moveOtherForm* opens a form and gets its position. The method then opens a second instance of the same form and sets its position so that no part of the second form overlaps the first:

```
; moveOtherForm::pushButton
method pushButton(var eventInfo Event)
var
  siteFormOne,
  siteFormTwo    Form
  x, y, w, h     LongInt
endVar
if siteFormOne.open("Sitenote") then
  siteFormOne.getPosition(x, y, w, h)
  siteFormTwo.open("Sitenote.fsl")    ; open another instance
  ; set position so that no part overlaps other instance
  siteFormTwo.setPosition(x + w, y + h, w, h)
endif
endmethod
```

See also

- setPosition

getTitle

Form

Method/Procedure

Returns the text of the window title bar.

Syntax

getTitle ( ) String

Description

Returns the text in the title bar of the window containing the object.

Example

In the following example, the **pushButton** method for *showTitle* opens a form, gets the new form's title and displays the title in a dialog box. This method then switches the open form to the Form Design window and retrieves its title again:

```
; showTitle::pushButton
method pushButton(var eventInfo Event)
var
   siteForm  Form
   titleText String
endVar
siteForm.open("Sitenote.fsl")
titleText = siteForm.getTitle() ; reads window title into titleText
msgInfo("Title:", titleText)    ; displays "Form : SITENOTE.FSL"
siteForm.design()               ; switch to the Form Design window
sleep()                         ; yield!
titleText = siteForm.getTitle() ; get the Form Design window title
msgInfo("Title:", titleText)    ; displays "Form Design: SITENOTE.FSL"
siteForm.close()
endmethod
```

See also

- attach, setTitle

hide

Beginner

Form

Method/Procedure

Makes a window invisible.

Syntax

hide ( )

Description

Makes a window invisible but doesn't close it.

Example

In this example, the **pushButton** method for *hideForm* opens a form, hides it, then shows it:

```
; hideForm::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
siteForm.open("Sitenote.fsl")        ; displays Sitenote form
siteForm.hide()                    ; makes form invisible
siteForm.action(DataEnd)           ; move to the end of the table
siteForm.action(DataBeginEdit)     ; start edit mode
siteForm.action(DataInsertRecord)  ; insert a new, blank record
if NOT siteForm.isVisible() then
  msgInfo("Status", "It's hidden.")
endif
message("Come out, come out, wherever you are!")
siteForm.show()                    ; make form visible again
if siteForm.isVisible() then
  msgInfo("Status", "It's visible.")
endif
endmethod
```

See also

- bringToTop, open, openAsDialog, show

hideSpeedBar

Form

Procedure

Makes the SpeedBar invisible.

Syntax

hideSpeedBar ( )

Description

Removes the SpeedBar from the Desktop. You must call **showSpeedBar** to restore it.

Example

In this example, the **pushButton** method for the *toggleSpeedBar* button checks whether the SpeedBar is showing. If the SpeedBar is visible, this method hides it; if the SpeedBar isn't visible, this method shows it:

```
; toggleSpeedBar::pushButton
method pushButton(var eventInfo Event)
if isSpeedBarShowing() then   ; if speedbar is off
   hideSpeedBar()             ; hide it
else                          ; otherwise
   showSpeedBar()             ; show it
endif
endmethod
```

See also

- isSpeedBarShowing, showSpeedBar

isMaximized

Form

Method/Procedure

Reports whether a window is displayed at its maximum size.

Syntax

isMaximized ( ) Logical

Description

Returns True if a form is displayed full screen; otherwise, it returns False.

Example

In this example, the **pushButton** method for the *cycleSize* button (on the current form) opens or attaches to the *Sitenote* form with the variable *siteForm*, and cycles from minimized to maximized to an intermediate size.

```
; cycleSize::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
; try attaching to form, since it might be open
if NOT siteForm.attach("Form : SITENOTE.FSL") then
  ; if attaching fails, try opening the form
  if NOT siteForm.open("sitenote.fsl") then
```

```
    msgStop("Failed", "Couldn't open Sitenote.")
    return     ; if open fails, give up
  endif
endif

; if we reach this point, we have a good form handle
switch
  case isMaximized()  :                   ; if forms are maximized
    msgInfo("Status", "Siteform is maximized.")
    siteForm.show()                       ; restore size
  case siteForm.isMinimized() :           ; if form is minimized
    msgInfo("Status", "Siteform is minimized.")
    siteForm.maximize()
  case NOT (siteForm.isMaximized() OR siteForm.isMinimized()):
    msgInfo("Status", "Siteform is neither minimized or maximized.")
    siteForm.minimize()                            ; minimize
  otherwise :
    msgStop("Stop", "Unable to change size of Siteform.")
endswitch
endmethod
```

See also

- isMinimized, maximize, minimize

isMinimized

Form

Method/Procedure

Reports whether a window is displayed as an icon.

Syntax

isMinimized ( ) Logical

Description

Returns True if a form is displayed as an icon; otherwise, it returns False.

Example

See the example for isMaximized.

See also

- isMaximized, maximize, minimize

isSpeedBarShowing

Form

Procedure

Reports whether the SpeedBar is visible.

Syntax

isSpeedBarShowing ( ) Logical

Description

Returns True if the SpeedBar is visible; otherwise, it returns False.

Example

See the example for hideSpeedBar.

See also

❐ hideSpeedBar, showSpeedBar

isVisible

Form

Method/Procedure

Reports whether any part of a window is displayed.

Syntax

isVisible ( ) Logical

Description

Returns True if any part of a window is displayed (not hidden); otherwise, it returns False.

Example

In the following example, the **pushButton** method for the *siteToTop* button attempts to attach to an open form. If the **attach** is successful, the method checks to see if the form is visible. If the form is visible, the method makes it the topmost window:

```
; siteToTop::pushButton
method pushButton(var eventInfo Event)
var
  siteForm Form
endVar
; if form is on desktop
if siteForm.attach("Form : SITENOTE.FSL") then
  if siteForm.isVisible() then     ; if form is visible
    siteForm.bringToTop()          ; make it the topmost layer
  else
    msgStop("Sorry", "Can't see Sitenote form.")
  endif
endif
endmethod
```

See also

❐ bringToTop, hide, show

keyChar

Form

Method

Sends an event to a form's **keyChar** method.

Syntax

1. **keyChar** (const ***aChar*** SmallInt, const ***vChar*** SmallInt, const ***state*** SmallInt) Logical
2. **keyChar** (const ***characters*** String [, const ***state*** SmallInt]) Logical

Description

Sends an event to a form's **keyChar** method. For syntax 1, you must specify the ANSI character code in *aChar,* the virtual key code in *vChar,* and the keyboard state constant in *state.* For syntax 2, you can

specify a string of one or more characters and, optionally, include a keyboard state constant.

ObjectPAL provides constants for *vChar* and *state*; see Keyboard and KeyBoardStates in the Constants dialog box.

Example

In this example, a form named *Otherfrm* is already open, and it contains one field named *fieldOne*. The form-level **keyChar** method for *Otherfrm* echoes characters to *fieldOne*. The **pushButton** method of a button named *callOtherKeyC* on the current form attaches to *Otherfrm* as *otherForm* and calls the **keyChar** method for *otherForm*, passing it a string. This is the code for the **pushButton** method for *callOtherKeyC* on the current form:

```
; callOtherKeyC::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; attach to the other form (assumes it's open)
if otherForm.attach("Form : OTHERFRM.FSL") then
  otherForm.keyChar("Hi! ")   ; send a string
else
  msgStop("Error", "The other form is not available.")
endif
endmethod
```

This code is attached to *Otherfrm*'s form-level **keyChar** method:

```
; thisForm::keyChar  (OTHERFRM.FSL)
method keyChar(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; send the key on to fieldOne
    msgInfo("Status", "Executing Otherfrm's keychar.")
    fieldOne.keyChar(eventInfo.char())
endif
endmethod
```

See also

- keyPhysical
- The description of the built-in keyChar method in Chapter 2

keyPhysical

Form

Method

Sends an event to a form's **keyPhysical** method.

Syntax

keyPhysical (const ***aChar*** SmallInt, const ***vChar*** SmallInt, const ***state*** SmallInt) Logical

Form

Description

Sends an event to a form's **keyPhysical** method. You must specify the ANSI character code in *aChar*, the virtual key code in *vChar*, and the keyboard state constant in *state*.

ObjectPAL provides constants for *vChar* and *state*; see Keyboard and KeyBoardStates in the Constants dialog box.

Example

In this example, a form named *OtherFr2* is already open, and it contains one field named *fieldOneThere*. The form-level **keyPhysical** method for *Otherfrm* echoes characters to *fieldOneThere*. The **keyPhysical** method of a field named *fieldOneHere* on the current form attaches to *Otherfrm* as *otherForm*. The method then calls the **keyPhysical** method for *otherForm*, passing it the ANSI code of the character or keypress, the virtual code of the character or keypress, and the keyboard state. This is the code for the **keyPhysical** method for *fieldOneHere* on the current form:

```
; fieldOneHere::keyPhysical    (current form)
method keyPhysical(var eventInfo KeyEvent)
var
  otherForm Form
endVar
; attach to the other form (assumes it's open)
if otherForm.attach("Form : OTHERFR2.FSL") then
  ; switch statement sorts out keyBoardState
  switch
    case eventInfo.isShiftKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(), Shift)
    case eventInfo.isAltKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(),
                            Alt)
    case eventInfo.isControlKeyDown() :
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(), Control)
    otherwise:
      otherForm.keyPhysical(eventInfo.charAnsiCode(),
                            eventInfo.vCharCode(), 0)

  endSwitch
else
  msgStop("Error", "The other form is not available.")
endif
endmethod
```

The following code is attached to the **keyPhysical** method for *otherForm*:

```
; thisForm::keyPhysical   (OTHERFRM)
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ; pass keyPhysical on to fieldOneThere
    ; switch statement sorts out keyBoardState
    switch
```

```
      case eventInfo.isShiftKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Shift)
      case eventInfo.isAltKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Alt)
      case eventInfo.isControlKeyDown() :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), Control)
      otherwise :
        fieldOneThere.keyPhysical(eventInfo.charAnsiCode(),
                                  eventInfo.vCharCode(), 0)
    endSwitch
endif
endmethod
```

See also

- keyChar
- The description of the built-in keyPhysical method in Chapter 2

load

Form

Method

Opens a form in the Form Design window.

Syntax

load (const ***formName*** String) Logical

Description

Opens *formName* in the Form Design window. This method works only with saved forms or reports (.FSL or .RSL); it does not work with delivered forms (.FDL or .RDL). For more information about saving and delivering forms, refer to the *ObjectPAL Developer's Guide.*

When designing or running a form, you can use UIObject type methods **create** and **methodSet** to place objects in the new form and attach methods to them. However, if you create objects while the form is running, the newly created objects will not automatically be saved when the form is closed.

Note Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

Example

In the following example, the **pushButton** method for a button named *drawABox* loads the *Sitenote* form in a design window. The method then sets the position of the form, creates a small box, names the box *newBox*, and sets its color to Blue. While the form is running, the box won't be visible; by default, the Visible property of objects created in this manner is False.

```
; drawABox::pushButton
method pushButton(var eventInfo Event)
```

Form

```
var
   myForm Form
   newObj UIObject
endVar
; open Sitenote in a design window
if myForm.load("Sitenote.fsl") then
   myForm.setPosition(720, 720, 1440*6, 1440*5)  ; 6" by 5"
   newObj.create(BoxTool, 1440, 1440*3, 360, 360, myForm)
   newObj.name = "newBox"
   newObj.color = Blue
else
   msgStop("Stop", "Couldn't load the form.")
endIf
endmethod
```

See also

- create, design, open, openAsDialog, run

maximize

Form

Beginner

Method/Procedure

Maximizes a window.

Syntax

maximize ( )

Description

Displays a window at its full size. Calling this method is equivalent to choosing Maximize from the Control menu.

Example

In this example, the **pushButton** method for the *goSites* button opens the *Sitenote* form (assumed to be in the current database), minimizes the current form, then waits for a response. If *Sitenote* returns "OK", this method maximizes the current form; otherwise, it restores the current form to its previous size:

```
; goSites::pushButton
method pushButton(var eventInfo Event)
var
  siteForm     Form
  returnString String
endVar
; open the Sitenote form, minimize self (this form), then wait
siteForm.open("Sitenote.fsl")
minimize()
returnString = String(siteForm.wait())
; if siteForm returned "OK", then maximize--otherwise, restore
if returnString = "OK" then
  maximize()
  siteForm.close()
else
  show()
  siteForm.close()
endif
endmethod
```

This code is attached to a button named *OKButton* on *Sitenote*:

```
; OKButton::pushButton
method pushButton(var eventInfo Event)
formReturn("OK")    ; return the string "OK" to the calling form
endmethod
```

See also

- isMaximized, isMinimized, minimize, show

menuAction

Form

Method/Procedure

Sends an event to a form's **menuAction** method.

Syntax

menuAction (const ***action*** SmallInt) Logical

Description

Constructs a MenuEvent and sends it to a form's built-in **menuAction** method. **action** is one of the MenuCommand constants, or a user-defined constant. ObjectPAL provides constants for **action**; see MenuCommands in the Constants dialog box. For more information on user-defined constants, see Chapter 6 in the *ObjectPAL Developer's Guide.*

Note You can't use **menuAction** to send a menu command constant that is equivalent to a command on the File menu. To simulate a File menu command, use one of the regular action constants, manipulate a property, or use one of the many System type methods that emulate File menu commands.

Example

In this example, the *sendATile* button on the current form opens the *Sitenote* form and sends it a MenuWindowTile action.

```
; sendATile::pushButton
method pushButton(var eventInfo Event)
var
  siteForm  Form
endVar
if siteForm.open("Sitenote.fsl") then
  siteForm.menuAction(MenuWindowTile)
endif
endmethod
```

See also

- action
- Procedures that begin with **dlg**... in the System type

methodDelete

Form

Method

Deletes a form-level method from a form.

Syntax

methodDelete (const ***methodName*** String) Logical

Description

Deletes a built-in or custom method specified in *methodName* from a form. You can also specify "Var", "Proc", "Uses", or "Const" in *methodName* to clear the Var, Proc, Uses, or Const window of a form. If *methodName* is a built-in method, the built-in behavior for that method is restored.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *User's Guide.*

Example

In this example, two forms are on the desktop in a design window: *Otherone* and *Othertwo*. The **pushButton** method for a button named *moveMethod* (on the current form) moves a method from *Otherone* to *Othertwo*:

```
; moveMethod::pushButton
method pushButton(var eventInfo Event)
var
  tempFormSrc,
  tempFormDest    Form
  transMethod String
endVar
; try to attach to both the source and the destination form
; assume source and destination are on the desktop in a design window
if tempFormSrc.attach("Form Design : OTHERONE.FSL") AND
   tempFormDest.attach("Form Design : OTHERTWO.FSL") then
  ; get definition for source form's mouseRightUp, then delete
  transMethod = tempFormSrc.methodGet("mouseRightUp")
  tempFormSrc.methodDelete("mouseRightUp")
  ; copy the method to the destination form mouseRightUp
  tempFormDest.methodSet("mouseRightUp", transMethod)
else
  msgStop("Error", "Couldn't attach to source and destination forms.")
endif
endmethod
```

See also

- methodGet, methodSet
- create, methodGet, methodSet in the UIObject type

---

methodGet

Form

Method

Gets a form-level method.

Syntax

methodGet (const ***methodName*** String) String

Description

Gets the text of the built-in or custom form-level method specified in *methodName* attached to a form. You can also specify "Var", "Const",

"Uses", or "Proc" to get the contents of the Var, Const, Uses, or Proc window of a form.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *User's Guide*.

Example

See the example for methodDelete.

See also

- methodDelete, methodSet
- create, methodGet, methodSet in the UIObject type

methodSet

Form

Method

Sets the definition of a form-level method.

Syntax

methodSet (const ***methodName*** String, const ***methodText*** String) Logical

Description

Writes the text in *methodText* to the built-in or custom form-level method *methodName*, overwriting any existing method definition. You can also specify "Var", "Const", "Uses", or "Proc" to set the contents of the Var, Const, Uses, or Proc window of a form.

This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *User's Guide*.

Example

See the example for methodDelete.

See also

- methodDelete, methodGet
- create, methodGet, methodSet in the UIObject type

minimize

Form

Method/Procedure

Minimizes a window.

Syntax

minimize ( )

Description

Displays a window as an icon. Calling this method is equivalent to choosing Minimize from the Control menu.

Example See the example for maximize.

See also

- isMaximized, isMinimized, maximize, show

mouseDouble

Form

Method Sends an event to a form's **mouseDouble** method.

Syntax

mouseDouble (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseDouble** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open and running. The **pushButton** method for a button named *sendMouseDouble* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseDouble** method for *otherForm*:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseDouble to target form at coordinates 1000, 1000
  otherForm.mouseDouble(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseDouble** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseDouble  (Othermse)
method mouseDouble(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseDouble"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
```

```
endif
endmethod
```

See also

- mouseDown, mouseRightDouble

mouseDown

Form

Method

Sends an event to a form's **mouseDown** method.

Syntax

mouseDown (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseDown** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseDown* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseDown** method for *otherForm*:

```
; sendMouseDown::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseDown to target form at coordinates 1000, 1000
  otherForm.mouseDown(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseDown** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseDown  (Othermse)
method mouseDown(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseDown"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
```

```
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseRightDown, mouseUp

mouseEnter

Form

Method

Sends an event to a form's **mouseEnter** method.

Syntax

mouseEnter (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseEnter** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseEnter* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseEnter** method for *otherForm*:

```
; sendMouseEnter::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseEnter to target form at coordinates 1000, 1000
  otherForm.mouseEnter (1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseEnter** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseEnter  (Othermse)
method mouseEnter(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseEnter"
    ; get the target and write name to lastTarget field
```

```
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseExit

mouseExit

Form

Method

Sends an event to a form's **mouseExit** method.

Syntax

mouseExit (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseExit** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state;* see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseExit* on the current form attaches to *Othermse* as *otherForm,* then calls the **mouseExit** method for *otherForm*:

```
; sendMouseExit::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseExit to target form at coordinates 1000, 1000
  otherForm.mouseExit(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseExit** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseExit  (Othermse)
method mouseExit(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseDown"
```

```
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseEnter

mouseMove

Form

Method

Sends an event to a form's **mouseMove** method.

Syntax

mouseMove (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseMove** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseMove* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseMove** method for *otherForm*:

```
; sendMouseMove::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseMove to target form at coordinates 1000, 1000
  otherForm.mouseMove(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseMove** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseMove  (Othermse)
method mouseMove(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
```

```
    lastMethod = "mouseMove"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseEnter, mouseExit

mouseRightDouble

Form

Method

Sends an event to a form's **mouseRightDouble** method.

Syntax

mouseRightDouble (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseRightDouble** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named send*MouseRightDouble* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightDouble** method for *otherForm*:

```
; mouseRightDouble::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightDouble to target form at coordinates 1000, 1000
  otherForm.mouseRightDouble(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseRightDouble** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightDouble  (Othermse)
method mouseRightDouble(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
```

```
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightDouble"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseDouble, mouseRightDown

---

mouseRightDown

Form

Method

Sends an event to a form's **mouseRightDown** method.

Syntax

mouseRightDown (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseRightDown** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseRightDown* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightDown** method for *otherForm*:

```
; mouseRightDown::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightDown to target form at coordinates 1000, 1000
  otherForm.mouseRightDown(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseRightDown** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightDown  (Othermse)
method mouseRightDown(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
```

```
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightDown"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseDown, mouseRightDouble

mouseRightUp

Form

Method

Sends an event to a form's **mouseRightUp** method.

Syntax

mouseRightUp (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseRightUp** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseRightUp* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseRightUp** method for *otherForm*:

```
; mouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseRightUp to target form at coordinates 1000, 1000
  otherForm.mouseRightUp(1000, 1000, RightButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseRightUp** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseRightUp  (Othermse)
method mouseRightUp(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
  then
```

```
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseRightUp"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

- mouseRightDown, mouseUp

mouseUp

Form

Method

Sends an event to a form's **mouseUp** method.

Syntax

mouseUp (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs a MouseEvent and sends it to a form's **mouseUp** method. The arguments *x* and *y* specify (in twips) the location of the event, and *state* specifies a key state. ObjectPAL provides constants for *state*; see KeyBoardStates in the Constants dialog box.

Example

In this example, assume the form *Othermse* is already open. The **pushButton** method for a button named *sendMouseUp* on the current form attaches to *Othermse* as *otherForm*, then calls the **mouseUp** method for *otherForm*:

```
; sendMouseUp::pushButton
method pushButton(var eventInfo Event)
var
  otherForm Form
endVar
; try to attach to target form
if otherForm.attach("Form : OTHERMSE.FSL") then
  ; send a mouseUp to target form at coordinates 1000, 1000
  otherForm.mouseUp(1000, 1000, LeftButton)
else
  msgStop("Quitting", "Could not find target form.")
endif
endmethod
```

This code is attached to the **mouseUp** method for *otherForm* (*Othermse*):

```
; otherMouse::mouseUp  (Othermse)
method mouseUp(var eventInfo MouseEvent)
var
  targObj  UIObject
endVar
if eventInfo.isPreFilter()
```

```
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; write method name to the lastMethod field
    lastMethod = "mouseUp"
    ; get the target and write name to lastTarget field
    eventInfo.getTarget(targObj)
    lastTarget = targObj.Name
endif
endmethod
```

See also

❐ mouseRightUp, mouseDown

moveTo

Form

Method

Moves to a form.

Syntax

moveTo ([const ***objectName*** String]) Logical

Description

Moves the focus to a form. Optionally, it moves to the object specified in *objectName*.

Example

In the following example, assume a form named *Sitenote* is already open. The **pushButton** method for the *goToSites* button in the current form attaches the variable *otherForm* to *Sitenote*, determines if *otherForm* is visible, and, if so, moves to *otherForm*. If *otherForm* is not visible, the method uses **show** to display the form at its default size (**show** also moves the focus to the target form):

```
; goToSites::pushButton
method pushButton(var eventInfo Event)
var
  otherForm  Form
endVar
; assume that Sitenote form is already open
if otherForm.attach("Form : SITENOTE.FSL") then
  if otherForm.isVisible() then
    otherForm.moveTo()         ; if form is visible, move to it
  else
    otherForm.show()           ; otherwise, make it visible
  endif
else
  msgStop("Stop", "Couldn't find form.")
endif
endmethod
```

See also

❐ moveToPage

moveToPage

Form

Method/Procedure

Displays a specified page of a form.

Syntax

moveToPage (const ***pageNumber*** SmallInt) Logical

Description

Displays the page of a form specified in *pageNumber*. *pageNumber* can be an integer variable or an integer constant, but it can't be an object ID. To move to a page by its object ID, use the **moveTo** method from the UIObject type.

Example

In this example, the current form has two pages. The *Sitenote* form exists in the working directory and has four pages. The **pushButton** method for *pageThruSites* (on the current form) first moves to the second page of the current form. Then the method opens the *Sitenote* form to the *otherForm* variable, and pages through *otherForm*:

```
; pageThruSites::pushButton
method pushButton(var eventInfo Event)
const
   BillingInfo = SmallInt(4)
endConst
var
   myForm, otherForm  Form
   somePage           SmallInt
endVar
moveToPage(2)                        ; moves to page 2 on this form
if otherForm.open("Sitenote.fsl") then  ; opens to first page
  sleep(2000)                        ; pause
  otherForm.moveToPage(2)            ; moves to page 2 of SiteNote
  sleep(2000)
  somePage = 3
  otherForm.moveToPage(somePage)     ; moves to page 3
  sleep(2000)
  otherForm.moveToPage(BillingInfo) ; moves to page 4
  sleep(2000)
endIf
endmethod
```

See also

- bringToTop

open

Form

Beginner

Method

Opens a window.

Syntax

1. **open** (const ***formName*** String
   [, const ***windowStyle*** LongInt]) Logical

2. **open** (const ***formName*** String, const ***windowStyle*** LongInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt) Logical
3. **open** (const ***openInfo*** FormOpenInfo) Logical

Description

Displays and runs the form specified in *formName*. Compare this method with **load**, which opens a form in a design window.

The optional arguments *x* and *y* specify the location of the upper left corner of the form (in twips), *w* and *h* specify the width and height (in twips), and *windowStyle* specifies display attributes. You can specify more than one window style element by adding the constants together. For example, the following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.open("sales", WinStyleVScroll + WinStyleHScroll)
```

ObjectPAL provides constants for *windowStyle*; see WindowStyles in the Constants dialog box.

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The FormOpenInfo record has been predeclared and has the following structure:

```
x, y, w, h      LongInt  ; position and size of the form
name            String   ; name of form to open
masterTable     String   ; new master table name
queryString     String   ; query to run (actual query string)
winStyle        LongInt  ; window style constant(s)
```

You can use the *masterTable* member to specify a different master table for the form (this is similar to choosing a different table for a form when you open the form from the Open Form dialog box). Alternatively, you can specify a query string in the *queryString* member. Paradox executes the query and opens the form; the result of the query is the master table.

Note Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

Example

In this example, the **keyPhysical** method for a field named *fieldOne* tests all key events. When the user presses *F1*, the form HELPFORM opens. The **keyPhysical** method opens a form from the current directory:

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  helpForm Form
endVar
message(eventInfo.vChar())
```

```
if eventInfo.vChar() = "VK_F1" then
  helpForm.open("helpform", WinStyleDefault,
                720, 720, 1440 * 2, 1440 * 4)
  disableDefault
endIf

endmethod
```

This example works like the previous example, except that it uses a FormOpenInfo record to set the characteristics of the form to be opened.

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  openHelpForm FormOpenInfo   ; a predeclared record type
  helpForm     Form
endVar
message(eventInfo.vChar())
if eventInfo.vChar() = "VK_F1" then
  openHelpForm.x = 720
  openHelpForm.y = 720
  openHelpForm.w = 2 * 1440
  openHelpForm.h = 4 * 1440
  openHelpForm.name = "helpform"
  helpForm.open(openHelpForm)
  disableDefault
endIf
endmethod
```

See also

- close, create, design, load, openAsDialog

openAsDialog

Form

Method

Opens a form window as a dialog box.

Syntax

1. **openAsDialog** (const ***formName*** [, const ***windowStyle*** LongInt]) Logical
2. **openAsDialog** (const ***formName*** String, const ***windowStyle*** LongInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt) Logical
3. **openAsDialog** (const ***openInfo*** FormOpenInfo) Logical

Description

Runs the form *formName* and displays it on top of any other open windows. *formName* is always on top, whether it's active or not. The optional arguments *x* and *y* specify the upper left corner of the window (in twips), *w* and *h* specify the width and height (in twips), and *windowStyle* specifies display attributes. ObjectPAL provides constants for *windowStyle*; see WindowStyles in the Constants dialog box.

You can specify more than one window style element by adding the constants. For example, the following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.open("sales", WinStyleVScroll + WinStyleHScroll)
```

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The FormOpenInfo record type is predeclared and has the following structure:

```
x, y, w, h       LongInt  ; position and size of the form
name             String   ; name of form to open
masterTable      String   ; new master table name
queryString      String   ; query to run (actual query string)
winStyle         LongInt  ; window style constant(s)
```

You can use the *masterTable* member to specify a different master table for the form (this is similar to choosing a different table for a form when you open the form from the Open Form dialog box). Alternatively, you can specify a query string in the *queryString* member. Paradox executes the query and opens the form; the result of the query is the master table.

Example

In this example, the **keyPhysical** method for a field named *fieldOne* tests all key events. When the user presses *F1*, the form HELPFORM opens. The **keyPhysical** method opens a form as a dialog box:

```
; fieldOne::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  helpForm Form
endVar
; if user presses F1, open a help dialog box
if eventInfo.vChar() = "VK_F1" then
  helpForm.openAsDialog("helpform", WinStyleDefault,
                       720, 720, 1440 * 4, 1440 * 3)
  helpForm.setTitle("Application Help") ; give the dialog box a new title
  helpForm.wait()
  helpForm.close()
  disableDefault                        ; don't call Help system
endIf
endmethod
```

See also

- formCaller, formReturn, open, wait

postAction

Form

Method

Posts an action to an action queue for delayed execution.

Syntax

postAction (const ***actionId*** SmallInt)

Description

Works like **action**, except that the action is not executed immediately. Instead, Paradox waits until the entire method has finished executing and Paradox is in a steady state. The action specified by *actionID* is posted to an action queue at the time of the method call; Paradox performs the action after the current method has finished executing.

Example

In this example, the **pushButton** method for *openSitesNew* opens the *Sitenote* form to the variable *otherForm*. The method then posts three actions to *otherForm*, and displays a message in a dialog box. The actions specified by **postAction** occur after the message dialog box appears and after this method ends:

```
; openSitesNew::pushButton
method pushButton(var eventInfo Event)
; otherForm variable is global to form--stays in scope after method ends
if otherForm.open("Sitenote.fsl") then
  ; these actions will not execute until after this method ends
  otherForm.postAction(DataEnd)          ; move to the last record
  otherForm.postAction(DataBeginEdit)    ; start Edit mode
  otherForm.postAction(DataInsertRecord) ; insert a new blank record
  msgInfo("Status", "About to perform posted actions. Watch closely.")
else
  msgStop("Stopped", "Could not open form.")
endif
endmethod
```

See also

- action
- The discussion of actions in Chapter 2

run

Form

Method

Runs a form that is currently open in the Form Design window.

Syntax

run ( ) Logical

Description

Switches a form from the Form Design window to a Form window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL). For more information about saving and delivering forms, refer to the *User's Guide*.

Use **design** to toggle a running form to the Form Design window.

Note Some form actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

Example

This example opens the *Sitenote* form in a design window, deletes the **pushButton** method from the form, then runs the form. Assume that the *Sitenote* form is in the current directory. This code is attached to the **pushButton** code for *delPushButton*:

```
; delPushButton::pushButton
method pushButton(var eventInfo Event)
var
  otherForm  Form
endVar
; load the Sitenote form, delete the pushButton
; method, then run the form
if otherForm.load("Sitenote") then
  otherForm.methodDelete("pushButton")
  otherForm.run()   ; since form is not saved, deletion
endif               ; won't be permanent
endmethod
```

See also

❐ design

save

Form

Method

Saves a form to disk.

Syntax

save ([const ***newFormName*** String]) Logical

Description

Writes a form to disk in the user's current working directory. This method works only when the form is in a design window.

You can use *newFormName* to specify a name for the form, or you can omit it. If you omit *newFormName* and the form doesn't have a name already, Paradox displays a dialog box to prompt the user to enter a name. If the form already has a name, Paradox saves it using that name.

Example

See the example for create.

See also

❐ create, design

setPosition

Form

Method/Procedure

Positions a window onscreen.

Syntax

setPosition (const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt)

Description

Positions a window onscreen. The arguments *x* and *y* specify the coordinates of the upper left corner of the form (in twips), and *w* and *h* specify the width and height (in twips).

For dialog boxes and for the Paradox Desktop, the position must be given relative to the entire screen; for forms, reports, and table windows, the position must be given relative to the Paradox Desktop.

Example

See the example for getPosition.

See also

- getPosition, open

setTitle

Form

Method/Procedure

Sets the text in the window title bar.

Syntax

setTitle (const ***text*** String)

Description

Changes the text of the window title bar to *text*. If you change a form's title, remember that you must use the new title when you want to attach to that form. (See the description of **attach** for more details.)

Example

See the example for openAsDialog.

See also

- attach, getTitle

show

Beginner

Form

Method/Procedure

Displays a minimized window at its previous size; makes a hidden form visible.

Syntax

show ( )

Description

Makes a hidden form visible. **show** also restores a minimized window to the size before it was minimized. This method is similar to the Restore command on the System/Control menu.

show doesn't make a form the topmost window; use **bringToTop** to make a form the top layer and give it focus.

Example See the example for hide.

See also ❐ hide, isVisible

showSpeedBar

Form

Procedure Makes the SpeedBar visible.

Syntax **showSpeedBar ( )**

Description Displays the SpeedBar.

Example See the example for hideSpeedBar.

See also ❐ hideSpeedBar, isSpeedBarShowing

wait

Beginner

Form

Method Suspends execution of a method.

Syntax **wait ( )** AnyType

Description Suspends execution of the current method until the form you're waiting for returns (see **formReturn**). This method is useful when you open a second form as a dialog box. Execution resumes in the first form when the second form (the one you're waiting for) calls **formReturn**, or when the second form closes. Once the called form returns, the calling form should close it with **close**. The called form does not automatically close, even if the user closes it; it stays open so that code on the calling form can examine it (for instance, to see settings on a dialog box).

Example See the example for formReturn.

See also ❐ formCaller, formReturn, openAsDialog

windowClientHandle

Form

Method/Procedure

Returns the handle of a window.

Syntax

windowClientHandle ( ) SmallInt

Description

A window handle is a unique integer identifier assigned to a window by Windows. **windowClientHandle** returns an integer value representing the window handle of the client area of a form. When called as a procedure, it returns the window handle of the client area of the current form. This method should be used only by advanced programmers.

This information is useful only if you're using functions from a Dynamic Link Library (DLL) written in C, C++, or Pascal. For more information about writing and using DLLs, refer to the **uses** entry in Chapter 3.

Example

In the following example, assume that a DLL called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle.

```
; someButton::pushButton
method pushButton(var eventInfo Event)
Uses MYTEST
  doSomething(const wHandle CHANDLE)
endUses
doSomething(windowClientHandle())  ; call doSomething and supply the
                                   ; handle of the client portion
                                   ; of the current form
endmethod
```

See also

- windowHandle
- the **uses** statement in Chapter 3

windowHandle

Form

Method/Procedure

Returns the handle of a window.

Syntax

windowHandle ( ) SmallInt

Description

A window handle is a unique integer identifier assigned to a window by Windows. **windowHandle** returns an integer value representing the window handle of a form. When called as a procedure, it returns

the window handle of the current form. This method should be used only by advanced programmers.

This information is useful only if you're using functions from a Dynamic Link Library (DLL) written in C, C++, or Pascal. For more information about writing and using DLLs, refer to the **uses** entry in Chapter 3.

Example

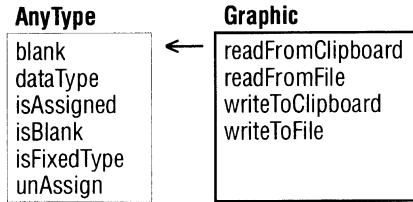
In the following example, assume that a DLL called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle.

```
; someButton::pushButton
method pushButton(var eventInfo Event)
Uses MYTEST
  doSomething(const wHandle CHANDLE)
endUses
doSomething(windowHandle())  ; call doSomething and supply the
                             ; window handle of the current form
endmethod
```

See also

- windowClientHandle
- the USES statement in Chapter 3

Graphic



A Graphic variable provides a handle for manipulating a graphic object. That is, you can use Graphic variables in ObjectPAL code to manipulate graphic objects. Graphic objects contain and display graphics in bitmap format (BMP). However, Paradox can import the following graphic formats: bitmap (BMP), encapsulated Postscript (EPS), graphic interchange format (GIF), Paintbrush (PCX), and tagged information file format (TIF).

Using Graphic type methods **readFromClipboard**, **writeToClipboard**, **readFromFile**, and **writeToFile**, you can use Graphic variables to transfer bitmaps between forms (and reports), tables, the Clipboard, and disk files.

For more information and examples, refer to the the *ObjectPAL Developer's Guide.*

The Graphic type also includes methods defined for the AnyType type.

readFromClipboard Graphic

Method

Reads a graphic from the Clipboard.

Syntax

readFromClipboard ( ) Logical

Description

Reads a graphic from the Clipboard to a variable of type Graphic. If the Clipboard contains a graphic that can be copied to the Graphic variable, **readFromClipboard** returns True. If the Clipboard is empty or does not contain a valid graphic, **readFromClipboard** returns False. **readFromClipboard** can read bitmap (BMP) and device independent bitmap (DIB) formats.

Example

In this example, a form contains a multi-record object named *BIOLIFE* bound to the *Biolife* table, and a button named *getGraphic*. The **pushButton** method for *getGraphic* locates the record with a Common Name field value of "Firefish", then writes the contents of the Clipboard to that record's Graphic field. If the Clipboard is empty or does not contain a graphic, the **readFromClipboard** method returns False, and the value of the Graphic field is not changed.

```
; getGraphic::pushButton
method pushButton(var eventInfo Event)

var
  myGraphic Graphic
endVar

if BIOLIFE.locate("Common Name", "Firefish") then

  if myGraphic.readFromClipboard() then
    ; get the current clipboard contents to myGraphic
    BIOLIFE.edit()                  ; start Edit mode on the table
    BIOLIFE.Graphic = myGraphic     ; write the bitmap to the field
    BIOLIFE.endEdit()               ; end Edit mode
  endIf
endIf
endmethod
```

See also

- readFromFile, writeToClipboard

readFromFile

Graphic

Method

Reads a graphic from a file.

Syntax

readFromFile (const ***fileName*** String) Logical

Description

Reads a graphic from a disk file specified in *fileName*. **readFromFile** returns True if the *fileName* name exists and contains a graphic format that can be imported; otherwise, it returns False. Paradox can import the following graphic formats: bitmap (BMP), encapsulated Postscript (EPS), graphic interchange format (GIF), Paintbrush (PCX), and tagged information file format (TIF).

Example

The following example assumes that a form contains a button named *getChess*, and an unbound graphic field named *bitmapField*. The **pushButton** method for *getChess* attempts to read the bitmap file CHESS.BMP from the C:\WINDOWS directory and stores CHESS.BMP in the *chessBmp* variable. If **readFromFile** is successful, *chessBmp* is written to the *bitmapField* object.

```
; getChess::pushButton
method pushButton(var eventInfo Event)
```

```
var
  chessBmp Graphic
endVar
; get the bitmap chess.bmp from the C:\Windows directory,
; and write it to the bitmapField graphic
if chessBmp.readFromFile("c:\\windows\\chess.bmp") then
  bitmapField = chessBmp
endIf
endmethod
```

See also

- readFromClipboard, writeToFile

writeToClipboard

Graphic

Method

Writes a bitmap to the Clipboard.

Syntax

writeToClipboard ( ) Logical

Description

Writes a bitmap to the Clipboard. **writeToClipboard** returns True if successful and False if it fails. Formats copied to Clipboard can be bitmap (BMP) or device independent bitmap (DIB).

Example

The following example assumes that a form contains a button named *getChessToClip*, and a bitmap field named *bitmapField*. The **pushButton** method for *getChessToClip* stores the value of *bitmapField* to *chessBmp*, then writes *chessBmp* to the Clipboard:

```
; getChessToClip::pushButton
method pushButton(var eventInfo Event)
var
  chessBmp Graphic
endVar
; get the bitmap from the bitmapField,
; and write it to the Clipboard
if NOT bitmapField.isblank() then
  chessBmp = bitmapField
  chessBmp.writeToClipboard()
endif
endmethod
```

See also

- readFromClipboard, writeToFile

writeToFile

Graphic

Method

Writes a bitmap to a file.

Syntax

writeToFile (const ***fileName*** String) Logical

Description

Writes a bitmap to a disk file specified in *fileName*. **writeToFile** returns True if the file specified can be created; otherwise it returns False.

Example

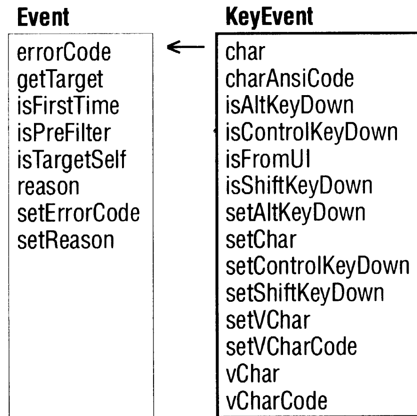
The following example assumes that a form contains a button named *writeChessToFile*, and a bitmap named *bitmapField*. The **pushButton** method for *writeChessToFile* stores the value of *bitmapField* to *chessBmp*, then writes *chessBmp* to a file in the current directory named CHESS1.BMP:

```
; writeChessToFile::pushButton
method pushButton(var eventInfo Event)
var
  chessBmp Graphic
endVar
; get the bitmap from the bitmapField,
; and write it to the Clipboard
if NOT bitmapField.isblank() then
  chessBmp = bitmapField
  chessBmp.writeToFile("chess1.bmp")
endif
endmethod
```

See also

- writeToClipboard, writetoFile

KeyEvent



A KeyEvent object gets and sets information about keystroke events.

The following built-in methods are triggered by KeyEvents: **keyChar**, and **keyPhysical**. These methods, along with the rest of the built-in methods, are discussed in Chapter 2. For information about the event model, see the *ObjectPAL Developer's Guide.*

The KeyEvent type includes methods defined for the Event type.

char

KeyEvent

Method

Returns the character associated with a keypress.

Syntax

char ( ) String

Description

Returns the character associated with a keypress. For example, if you type **a**, **char** returns "a." If you press *Shift+A*, **char** returns A. If a keypress results in an unprintable character, **char** returns an empty string ("").

char is the easiest way to check for an alphanumeric keypress when case matters. If case doesn't matter, use **vChar** to test against the string value of a virtual key code. For instance, if it matters whether the user presses a lowercase **a** or an uppercase **A**, use **char** to return the string value of the character pressed, and compare it to "a" or "A". If you want to find out if either **a** or **A** was pressed, use **vChar** and compare it to "A" (the virtual key code string for a keypress of a lowercase **a** or an uppercase **A**).

Example

This example displays, as a message at the bottom of the screen, the character typed into a field object. The code is attached to a field object's built-in **keyChar** method.

```
; thisField::keyChar
method keyChar(var eventInfo KeyEvent)
doDefault                  ; put character in the field
message(eventInfo.char()) ; then display character as a message
endmethod
```

See also

- charAnsiCode, vChar, vCharCode
- ansiCode, chrToKeyName, toANSI, and toOEM in the String type

charAnsiCode

KeyEvent

Method

Returns the ANSI value associated with a keypress.

Syntax

charAnsiCode ( ) SmallInt

Description

Returns an integer representing the ANSI value associated with a keypress. For example, if you type **a**, **charAnsiCode** returns 97. If you press *Shift+A*, **charAnsiCode** returns 65. **charAnsiCode** works with unprintable characters as well. For example, if you press *Enter*, **charAnsiCode** returns 13.

ANSI characters are listed in Appendix B.

Example

This example beeps when a user presses *Backspace* or *Ctrl+H*. This code is attached to a field object's built-in **keyPhysical** method.

```
; thisField::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.charAnsiCode() = 8 then  ; if user presses Ctrl+H or Backspace
  beep()                              ; make a sound
endif
endmethod
```

See also

- char, vChar, vCharCode
- ansiCode, chrToKeyName, toANSI, and toOEM in the String type

isAltKeyDown

KeyEvent

Method

Reports whether *Alt* was held down during a KeyEvent.

Syntax **isAltKeyDown ( )** Logical

Description Returns True if *Alt* was held down at the time a KeyEvent occurred; otherwise, it returns False.

Example The following example assumes a form has a box named *boxOne*. When the user presses *Alt+C*, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.isAltKeyDown() AND    ; if user presses Alt+C
       eventInfo.vChar() = "C" then
       disableDefault                  ; block normal processing
       ; alternate a boxOne's color between red and blue
       boxOne.color = iif(boxOne.color = Red, Blue, Red)
    endif
  else
    ; code here executes just for form itself
endif
endmethod
```

See also

- isControlKeyDown, isShiftKeyDown, setAltKeyDown

isControlKeyDown

KeyEvent

Method Reports whether *Ctrl* was held down during a KeyEvent.

Syntax **isControlKeyDown ( )** Logical

Description Returns True if *Ctrl* was held down at the time a KeyEvent occurred; otherwise, it returns False.

Example The following example assumes a form has a box named *boxOne*. When the user presses *Ctrl+C*, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter() then
  ; code here executes for each object in form
  if eventInfo.isControlKeyDown() and   ; if user presses Ctrl+C
    eventInfo.vChar() = "C" then
    disableDefault                      ; block normal processing
    ; alternate color of boxOne between red and blue
    boxOne.color = iif(boxOne.color = Red, Blue, Red)
```

```
  endif
else
  ; code here executes just for form itself
endif
endmethod
```

See also

- isAltKeyDown, isShiftKeyDown, setControlKeyDown

isFromUI

KeyEvent

Method

Reports whether a KeyEvent was generated by the user interacting with Paradox.

Syntax

isFromUI ( ) logical

Description

Reports whether a KeyEvent was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True if the KeyEvent was generated by the user; otherwise, it returns False.

See also

- isPreFilter in the Event type

isShiftKeyDown

KeyEvent

Method

Reports whether *Shift* was held down during a KeyEvent.

Syntax

isShiftKeyDown ( ) Logical

Description

Returns True if *Shift* was held down at the time a KeyEvent occurred; otherwise, it returns False.

Example

See the example for vChar.

See also

- isAltKeyDown, isControlKeyDown, setShiftKeyDown

setAltKeyDown

KeyEvent

Method

Simulates pressing and holding *Alt* during a KeyEvent.

Syntax

setAltKeyDown (const ***yesNo*** Logical)

Description

Adds information about the state of *Alt* to a KeyEvent. You must specify Yes or No. Yes means *Alt* was pressed during a KeyEvent; No means *Alt* was not pressed.

Example

The following example assumes a form has a box named *boxOne*. When the user presses *Alt+C*, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.isAltKeyDown() and    ; if user presses Alt+C
       eventInfo.vChar() = "C" then
       disableDefault                  ; block normal processing
       ; alternate a boxOne's color between red and blue
       boxOne.color = iif(boxOne.color = Red, Blue, Red)
    endif
  else
    ; code here executes just for form itself
endif
endmethod
```

To simulate pressing *Alt+C*, the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the *Alt* key down.

```
; sendAltC::pushButton
method pushButton(var eventInfo Event)
var
  ke KeyEvent
endVar
ke.setVChar("C")             ; set the character to C
ke.setAltKeyDown(Yes)        ; set the Alt key state to pressed
thisForm.keyPhysical(ke)     ; send off the event
endmethod
```

See also

- isAltKeyDown, setControlKeyDown, setShiftKeyDown

---

setChar

KeyEvent

Method

Specifies an ANSI character for a KeyEvent.

Syntax

setChar (const ***char*** String)

Description

Sets a KeyEvent to have an ANSI character based on the value of *char*, where *char* evaluates to single character string (example, a). ANSI characters are listed in Appendix B.

Example

This code is attached to a field's built-in **keyChar** method. The **keyChar** method for *fieldOne* converts each space to an underscore as the user types characters into the field:

```
; thisField::keyChar
method keyChar(var eventInfo KeyEvent)
  if eventInfo.Char() = " " then  ; when user enters a space
    eventInfo.setChar("_")        ; convert it to underscore
  endif                           ; process other keystokes normally
endmethod
```


See also

- setVChar, setVCharCode
- chr, chrOEM, keyNameToChr, toANSI, toOEM in the String type

setControlKeyDown

KeyEvent

Method

Simulates pressing and holding *Ctrl* during a KeyEvent.

Syntax

setControlKeyDown (const ***yesNo*** Logical)

Description

Adds information about the state of *Ctrl* to a KeyEvent. You must specify Yes or No. Yes means *Ctrl* was pressed during a KeyEvent; No means *Ctrl* was not pressed.

Example

The following example assumes a form has a box named *boxOne.* When the user presses *Ctrl+C,* the **keyPhysical** method for the form changes the color of *boxOne.* This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter() then
  ; code here executes for each object in form
  if eventInfo.isControlKeyDown() and   ; if user presses Ctrl+C
    eventInfo.vChar() = "C" then
    disableDefault                      ; block normal processing
    ; alternate color of boxOne between red and blue
    boxOne.color = iif(boxOne.color = Red, Blue, Red)
  endif
else
  ; code here executes just for form itself
endif
endmethod
```

To simulate *Ctrl+C,* the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the *Ctrl* key down.

```
; sendCtrlC::pushButton
method pushButton(var eventInfo Event)
var
```

```
  ke KeyEvent
endVar
ke.setChar("C")               ; set the character to C
ke.setControlKeyDown(Yes)     ; set the Ctrl key state to pressed
thisForm.keyPhysical(ke)      ; send off the event
endmethod
```

See also

- isControlKeyDown, setAltKeyDown, setShiftKeyDown

setShiftKeyDown

KeyEvent

Method

Simulates pressing and holding *Shift* during a KeyEvent.

Syntax

setShiftKeyDown (const *yesNo* Logical)

Description

Adds information about the state of *Shift* to a KeyEvent. You must specify Yes or No. Yes means *Shift* was pressed and held; No means *Shift* wasn't pressed.

Example

The following example assumes a form has a box named *boxOne*. When the user presses *Shift+C*, the **keyPhysical** method for the form changes the color of *boxOne*. This code is attached to a form's **keyPhysical** method:

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter() then
  ; code here executes for each object in form
  if eventInfo.isShiftKeyDown() and    ; if user presses Ctrl+C
    eventInfo.vChar() = "C" then
    disableDefault                     ; block normal processing
    ; alternate color of boxOne between red and blue
    boxOne.color = iif(boxOne.color = Red, Blue, Red)
  endif
else
  ; code here executes just for form itself
endif
endmethod
```

To simulate pressing *Shift+C*, the code for this method creates a KeyEvent variable, then sets its virtual key character to "C" and sets the *Shift* key down.

```
; sendShiftC::pushButton
method pushButton(var eventInfo Event)
var
  ke KeyEvent
endVar
ke.setVChar("C")            ; set the character to C
ke.setShiftKeyDown(Yes)     ; set the Shift key state to pressed
thisForm.keyPhysical(ke)    ; send off the event
endmethod
```

See also

- isShiftKeyDown, setAltKeyDown, setControlKeyDown

setVChar

KeyEvent

Method

Specifies a Windows virtual character for a KeyEvent.

Syntax

setVChar (const ***char*** String)

Description

Specifies in *char* a one-character string for a KeyEvent. Use **setVChar** to set a virtual character code string for a single letter. The virtual character code string for any letter is the uppercase letter. For instance, the virtual character code string for the letter k is "K" (uppercase only).

Example

See the example for setAltKeyDown.

See also

- setChar, setVCharCode
- chr, chrOEM, chrToKeyName in the String type

setVCharCode

KeyEvent

Method

Specifies a Windows virtual character for a KeyEvent.

Syntax

setVCharCode (const ***VK_Constant*** SmallInt)

Description

Specifies in *VK_Constant* a Windows virtual character constant for a KeyEvent. ObjectPAL provides constants for *VK_Constant*; see Keyboard in the Constants dialog box.

Example

This code is attached to a form's built-in **keyPhysical** method. When the user types **?**, this code invokes the Paradox Help system.

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    if eventInfo.char() = "?" then   ; if user types ?
      eventInfo.setVCharCode(VK_HELP)  ; invoke built-in help system
    endif
  else
    ; code here executes just for form itself
endif
endmethod
```

See also

- setChar, setVChar
- keyNameToVKCode in the String type

vChar

KeyEvent

Method

Returns a Windows virtual character.

Syntax

vChar ( ) String

Description

Returns a Windows virtual key name as a string. In the case of letters, the virtual key name is the same as the letter, except that it is always uppercase. Thus, you can use **vChar** to test whether the user typed a lowercase **j** or uppercase **J**. ObjectPAL provides constants for virtual key names; see Keyboard in the Constants dialog box.

Example

In the following example, assume a form contains a box named *boxOne*. When the user presses one of the arrow keys, this code moves *boxOne* in increments of 100 twips. If *Shift* is held down in combination with a movement key, *boxOne* moves 1000 twips. Since **vChar** returns the virtual key name as a string, this code must compare key names against string values such as "VK_LEFT". This code is attached to a form's built-in **keyPhysical** method.

```
; thisForm::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  kp      String       ; key name of the keystroke
  posPt   Point        ; x and y position of the box object
  boxStep SmallInt     ; number of Points to move the box
  x, y    LongInt      ; coordinates of the box object
endVar

if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    disableDefault                  ; don't execute built-in code

    kp = eventInfo.vChar()          ; load kp with vChar string
    posPt = boxOne.position         ; posPt stores current position of box
    x = posPt.x()                   ; x stores the horizontal position
    y = posPt.y()                   ; y stores the vertical position

    ; if the Shift key was held down when the movement key was pressed,
    ; assign a large number to boxStep, else, a small number
    boxStep = iif(eventInfo.isShiftKeyDown(), 1000, 100)

    ; this block assigns x or y variables according to
    ; the key combination that the user presses
    switch
      case kp = "VK_LEFT"  : x = x - boxStep
      case kp = "VK_RIGHT" : x = x + boxStep
      case kp = "VK_UP"    : y = y - boxStep
```

```
      case kp = "VK_DOWN"   : y = y + boxStep
      otherwise             : enableDefault    ; let built-in code execute
    endswitch

    ; now move the box to location specified by x and y variables,
    ; and display the virtual key name associated with the keystoke
    boxOne.position = Point(x,y)
    message("Value of vChar() was " + kp)

  else
    ; code here executes just for form itself
endif
endmethod
```

See also

- char, charAnsiCode, vCharCode
- ansiCode, chrToKeyName, toANSI, and toOEM in the String type

vCharCode

KeyEvent

Method

Returns the integer value of a Windows virtual character.

Syntax

vCharCode ( ) SmallInt

Description

Returns the integer value of a Windows virtual character. Windows virtual characters are listed in Appendix C.

ObjectPAL provides constants for virtual character codes; see Keyboard in the Constants dialog box.

Example

In this example, assume a form has a field named *thisField*. When the user types a value in *thisField* and presses *Enter*, the code creates and executes a query based on the value of the field. This code is attached to the built-in **keyPhysical** method for *thisField*.

```
; thisField::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  cName String       ; used as tilde var
  qs Query           ; the query statement
  tv TableView       ; tableView handle
endVar

if eventInfo.vCharCode() = VK_RETURN then      ; if user presses Enter
  cName = self.value                           ; store value of field
  qs = Query

      c:\pdoxwin\sample\biolife.db | Common Name  | Species Name |
                                   | check ~cName | check        |

    endQuery

  executeQBE(qs, "myFish.db")     ; run query, write contents to myFish table
  tv.open("myFish")               ; view myFish table view
```

```
endif
endmethod
```

See also

- char, charAnsiCode, vChar
- chr, chrOEM, VKCodeToKeyName in the String type

Library

Library

close
enumSource
enumSourceToFile
execMethod
open

A library is a Paradox object that stores custom methods, custom procedures, variables, constants, and user-defined data types. Libraries are useful for storing and maintaining frequently-used routines, and for sharing custom methods and variables among several forms.

In many ways, working with a library is like working with a form. For example, to create a form, choose File | New | Form; to create a library, choose File | New | Library. Like a form, a library has built-in methods. You add code to a library just as you do to a form, using the Methods dialog box and the ObjectPAL Editor. As with a form, you can open Editor windows to declare custom methods, procedures, variables, constants, data types, and external routines.

For more information and examples, refer to the *ObjectPAL Developer's Guide.*

close

Library

Method

Closes a library.

Syntax

close ( )

Description

Closes a library, and ends the association between a Library variable and the underlying library file.

Example

This example declares a Library variable named *lib,* and calls **open** to associate *lib* with the library TOOLS.LSL. The example executes a method from that library, then calls **close** to end the association between the variable and the library. Another call to **open** associates *lib* with the library KIT.LSL, making methods in that library available.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  lib Library          ; declare a Library variable
endVar
```

```
lib.open("TOOLS.LSL")  ; associate lib with the library TOOLS.LSL
lib.doThis()           ; execute a method from the library
lib.close()            ; end the association between lib and the library

lib.open("KIT.LSL")    ; associate lib with another library
lib.doThat()           ; execute a method from the library

endmethod
```

See also

- open

enumSource

Library

Method

Writes the code from a library to a Paradox table.

Syntax

enumSource (const ***tableName*** String [, const ***recurse*** Logical])

Description

Lists, in the Paradox table specified in *tableName,* all the custom code (methods, procedures, variables, etc.) stored in a library. If the table does not exist, Paradox creates it in the current working directory; if the table does exist, information is appended to it.

The structure of the table is:

Field name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

The Object field stores the UIObject name of the library, the MethodName field stores the name of the method, procedure, or window (Var, Const, Proc, Type, or Uses), and the Source field stores the corresponding source code.

This method also applies to the Form type. For forms, the optional argument *recurse* specifies whether to include overridden methods for all objects contained by the form. Because a Library does not contain objects, the *recurse* argument is not meaningful in the context of a Library.

You must call **open** to open the library before calling this method.

Example

This example declares a Library variable named *lib,* and calls **open** to associate *lib* with the library TOOLS.LSL. Then the example calls **enumSource** to list the code from the library to a Paradox table named LIBSRC.DB:

```
; srcToTable::pushButton
method pushButton(var eventInfo Event)
```

```
var
  lib Library
endVar

if lib.open("TOOLS.LSL", PrivateToForm) then

  ; write contents of TOOLS.LSL to LIBSRC.TXT--
  ; goes to :WORK: by default
  lib.enumSource("LIBSRC.DB")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endIf

endmethod
```

See also

❐ enumSourceToFile, open

Library

enumSourceToFile

Library

Method

Writes the code from a library to a text file.

Syntax

enumSourceToFile (const ***fileName*** String [, const ***recurse*** Logical])

Description

Lists all the custom code (methods, procedures, variables, and so on) stored in a library to the text file specified in *fileName*. If the file does not exist, Paradox creates it in the current working directory; if the file does exist, Paradox overwrites it without asking for confirmation.

In the text file, comment lines are used to identify and mark the beginning and end of each method, procedure, and so on. The following example shows the code for a library's Var window and built-in **open** method:

```
;|BeginMethod|#script1|Var|

Var
   myMsgCursor TCursor
endVar

;|EndMethod|#script1|Var|
;|BeginMethod|#script1|open|
method open(var eventInfo Event)

if not myMsgCursor.open("Msghelp.db")
   then msgStop("Error", "Couldn't open MsgHelp.db")
      fail()
endif

endmethod
;|EndMethod|#script1|open|
```

This method also applies to the Form type. For forms, the optional argument *recurse* specifies whether to include overridden methods for

all objects contained by the form. Because a Library does not contain objects, the *recurse* argument is not meaningful in the context of a Library.

You must call **open** to open the library before calling this method.

Example

This example declares a Library variable named *lib*, and calls **open** to associate *lib* with the library TOOLS.LSL. Then the example calls **enumSourceToFile** to list the code from the library to a text file named LIBSRC.TXT.

```
; getSource::pushButton
method pushButton(var eventInfo Event)
var
  lib Library
endVar

if lib.open("TOOLS.LSL", PrivateToForm) then

  ; write contents of TOOLS.LSL to LIBSRC.TXT--
  ; goes to :PRIV: by default
  lib.enumSourceToFile("LIBSRC.TXT")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endIf

endmethod
```

See also

❒ enumSource, open

execMethod

Library

Method

Calls a custom method that takes no arguments.

Syntax

execMethod (const ***methodName*** String)

Description

Calls the custom method indicated by the string *methodName*. The method named in *methodName* takes no arguments. **execMethod** allows you to call a library method based on the contents of a variable, which means the compiler does not know the method to call until run time. However, the method must be declared in the appropriate Uses window.

Example

This example creates an array of three items, where each item is the name of a custom method in a library. The code opens the library and calls **execMethod** for each item in the array:

```
var
   lib Library
   libMethods Array[3] String
```

```
   i SmallInt
endVar

libMethods[1] = "doThis"
libMethods[2] = "doThat"
libMethods[3] = "doOther"

if lib.open("tools.lsl", GlobalToDeskTop) then
   for i from 1 to libMethods.size()
      lib.execMethod(libMethods[i])
   endFor
else
   msgStop("TOOLS.LSL", "Could not open library.")
endIf
```

See also

- open

Library

open

Library

Method

Associates a Library variable with a library, and makes the library code available.

Syntax

open (const ***libraryName*** String [, const ***libScope*** SmallInt]) Logical

Description

Associates a Library variable with a library, and makes the library code available to one or more forms, depending on the value of *libScope*. ObjectPAL defines two constants for specifying the scope of a library: PrivateToForm and GlobalToDesktop (listed in the Constants dialog under LibraryScope):

- PrivateToForm: only the form that opened the library has access to its code.
- GlobalToDesktop: every form in the Desktop (Paradox session) has access to the library.

To open a library and make it available to every form in the current session of Paradox, use the argument GlobalToDesktop. For example, the following statement opens the library MYLIB.LSL:

```
lib.open("myLib.lsl", GlobalToDesktop)
```

For two or more forms to share the same library, each form must open the library global to the Desktop, and each form must have a Uses window that declares which library routines to use. This level of scope is useful in multiform applications, because it allows several forms access to the same custom methods and allows the forms to share the same global variables.

A library can be opened private to the form in one form and global to the Desktop in another form. Paradox will load a new instance of the library, if necessary.

By default, a library opens global to the Desktop. The following statements are equivalent:

```
lib.open("myLib.lsl") ; these statements are equivalent
lib.open("myLib.lsl", GlobalToDesktop)
```

Example

This example shows how two forms can open a library global to the Desktop and share the library. In the following code, attached to a form's built-in **open** method, *libOne* is opened private to the form. *libOne* cannot be shared. *libTwo* is opened global to the Desktop and can be shared.

```
; formOne::open
method open(var eventInfo Event)
var
  libOne, libTwo Library
endVar

if eventInfo.isPreFilter()
   then
   ; code here executes for each object in the form
else
   ; code here executes just for the form itself

   libOne.open("TOOLS.LSL", PrivateToForm)   ; no sharing with other forms
   libTwo.open("KIT.LSL", GlobalToDesktop)   ; can be shared with other forms

endIf

endmethod
```

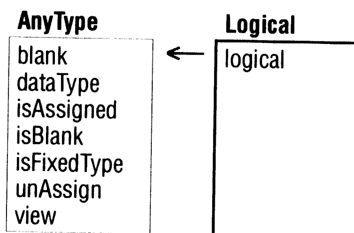
The following code, attached to another form's built-in **open** method, calls **open** to open the library KIT.LSL global to the Desktop. This form and the previous form can now share KIT.LSL.

```
; formTwo::open
method open(var eventInfo Event)
var
   kitLib Library
endVar

if eventInfo.isPreFilter()
   then
   ; code here executes for each object in the form
else
   ; code here executes just for the form itself
   kitLib.open("KIT.LSL", GlobalToDesktop) ; can be shared with other forms
endIf

endmethod
```

See also

- close

Logical



Logical

Logical variables have two possible values: True or False. You can use the ObjectPAL constants Yes or On in place of True, and use No or Off in place of False.

A Logical variable occupies 1 byte of storage. In order of precedence, the logical operators are NOT, AND, and OR.

Logical variables often answer questions about other objects and operations, for example,

- Is that table empty?
- Is that form displayed as an icon?
- Did that operation successfully create a text file?

For more information and examples, refer to the *ObjectPAL Developer's Guide*.

The Logical type also includes methods defined for the AnyType type.

logical

Beginner

Logical

Procedure

Casts a value as type Logical.

Syntax

logical (const ***value*** AnyType) Logical

Description

Casts (converts) the data type of *value* to Logical. If *value* is a numeric data type, non zero values evaluate to True and zero evaluates to True. If *value* is a string, it must evaluate to "True" or "False". (However, you can use True or False without the quotation marks.) ObjectPAL also provides Logical constants: On and Yes for True and Off and No for False; see General in the Constants dialog.

Example

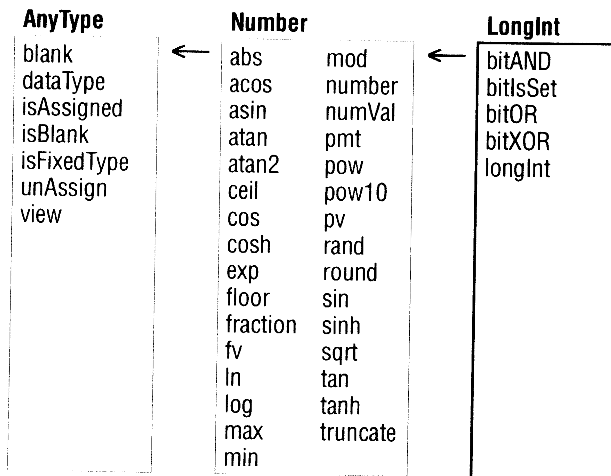
In this example, the **pushButton** method of a button named *showLogical* creates a string, casts it to a Logical type; then displays the result:

```
; showLogical::pushButton
method pushButton(var eventInfo Event)
var
  myVal     String
  theResult Logical
endVar
myVal = "True"              ; set a String of True
theResult = logical(myVal) ; and cast it to a Logical type
theResult.view()            ; show the result--Title displays Logical
endmethod
```

See also

- The description of the Logical type in the *ObjectPAL Developer's Guide*

LongInt



LongInt values are long integers; that is, they can be represented by a long series of digits. A LongInt variable occupies 4 bytes.

ObjectPAL converts LongInt values to range from -2,147,483,648 to 2,147,483,647. An attempt to assign a value outside of this range to a LongInt variable causes an error, for example,

```
var
   x, y, z LongInt
endVar

x = 2147483647 ; the upper limit value for a LongInt variable
y = 1
z = x + y      ; causes an error
```

To work with boundary values, store the result in a Number variable.

For more information and examples, refer to the *ObjectPAL Developer's Guide*.

Note Run-time library methods defined for the Number type also work with LongInt variables. The syntax is the same, and the returned value is a number. The LongInt type also includes methods defined for the AnyType type.

bitAND

LongInt

Method

Performs a bitwise AND operation on two values.

Syntax

bitAND (const ***value*** LongInt) LongInt

Description

Returns the result of a bitwise AND operation on value. **bitAND** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitAND** is:

a	b	a bitAND b
0	0	0
1	0	0
0	1	0
1	1	1

Example

In the following example, the **pushButton** method for a button named *andTwoNums* takes two integers and performs a bitwise AND calculation on them. The result of the calculation is displayed in a dialog box.

```
; andTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333    ; binary 00000000 00000000 10000010 00110101
b = -77777   ; binary 11111111 11111110 11010000 00101111
a.bitAND(b)  ; binary 00000000 00000000 10000000 00100101
msgInfo("The result of a bitAND b is:", a.bitAND(b))
; displays 32805
endmethod
```

See also

- bitOR, bitXOR

bitIsSet

LongInt

Method

Reports whether a bit is 1 or 0.

Syntax

bitIsSet (const ***value*** LongInt) Logical

Description

Examines the binary respresentation of an integer, reporting whether the **value** bit is 0 or 1. **bitIsSet** returns True if the bit specified is 1, and False if the bit is 0.

value is a number specified by 2^n, where *n* is an integer between 0 and 30. The exponent *n* corresponds to one less than the position of the bit to test, counting from the right. For example, to specify the third bit from the right, use 4 ($2^{(3-1)}$, which is 2^2).

Example

In the following example, the **pushButton** method for a button named *isABitSet*, examines the values in two unbound field objects: *whichBit* and *whatNum*. *whichBit* contains the bit position (counting from the right) of the bit to test. *whatNum* contains the long integer to test.

The **pushButton** method uses *whichBit* to calculate the value of the position, then assigns the result to *bitNum*. The method then checks *Num* to see if the *bitNum* bit is set, and displays the Logical result with a **msgInfo** dialog box.

```
; isABitSet::pushButton
method pushButton(var eventInfo Event)
var
  bitNum,
  Num      LongInt
endVar
; get the bit position number from the whichBit
; field and convert to multiple of 2
bitNum = LongInt(pow(2, whichBit - 1))
; get the number to test from the whatNum field
Num = whatNum
; is the bit for value bitNum 1 in Num?
msgInfo("Is Bit Set?", Num.bitIsSet(bitNum))
endmethod
```

The next example illustrates how you can use **bitIsSet** to display a long integer as a binary number. The **pushButton** method for *showBinary* constructs a string of zeros and ones by testing each bit of a four-byte long integer. For readability, a blank is added to the string every 8 digits.

```
; showBinary::pushButton
method pushButton(var eventInfo Event)
var
  binString  String    ; to construct the binary string
  Num        LongInt
  i          SmallInt  ; for loop index
endVar
if NOT whatNum.isBlank() then
  Num = whatNum                  ; get the number test from whatNum
  binString = ""                 ; initialize the string
  for i from 0 to 30
    if Num.bitIsSet(LongInt(pow(2, i))) then
      binString = "1" + binString    ; add a 1 to the front of the string
    else
      binString = "0" + binString    ; add a 0 to the front of the string
    endif
    if i = 7 OR i = 15 OR i = 23 then
      binString = " " + binString    ; add a space every 8 digits
    endif
  endfor
  if Num < 0 then
    binString = "1" + binString      ; set the sign bit
  else
    binString = "0" + binString
  endif
  ; show the number
  message("The binary equivalent is ", binString)
endif
endmethod
```

See also

- bitAND, bitOR, bitXOR

bitOR

LongInt

Method

Performs a bitwise OR operation on two values.

Syntax

bitOR (const ***value*** LongInt) LongInt

Description

Returns the result of a bitwise OR operation on *value*. **bitOR** operates on the binary representations of two integers, comparing them one bit at a time. Here is the truth table for **bitOR**:

a	b	a bitOR b
0	0	0
1	0	1
0	1	1
1	1	1

Example

For the following example, the **pushButton** method for a button named *orTwoNums* takes two integers and performs a bitwise OR calculation on them. The result of the calculation is displayed in a dialog box.

```
; orTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333  ; binary 00000000 00000000 10000010 00110101
b = -77777 ; binary 11111111 11111110 11010000 00101111
a.bitOR(b) ; binary 11111111 11111110 11010010 00111111
msgInfo("33333 OR -77777", a.bitOR(b)) ; displays -77249
endmethod
```

See also

- bitAND, bitXOR

bitXOR

LongInt

Method

Performs a bitwise XOR operation on two values.

Syntax

bitXOR (const ***value*** LongInt) LongInt

Description

Performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary respresentations of two integers, comparing them one bit at a time. Here is the truth table for **bitXOR**:

a	b	a.bitXOR(b)
0	0	0
1	0	1
0	1	1
1	1	0

Example

In the following example, the **pushButton** method for a button named *xorTwoNums* takes two integers and performs a bitwise XOR calculation on them. The result of the calculation is displayed in a dialog box.

```
; xorTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b LongInt
endVar
a = 33333   ; binary 00000000 00000000 10000010 00110101
b = -77777  ; binary 11111111 11111110 11010000 00101111
a.bitXOR(b) ; binary 11111111 11111110 01010010 00011010
msgInfo("33333 XOR -77777", a.bitXOR(b)) ; displays -110054
endmethod
```

See also

❒ bitAND, bitOR

---

longInt

Beginner

LongInt

Procedure

Casts a value as a LongInt.

Syntax

longInt (const ***value*** AnyType) LongInt

Description

Casts (converts) the data type of *value* to a long integer. If you convert from a more precise type (for example, Number), precision may be lost.

Example

The following example assigns a number to *x*, then casts *x* to LongInt and assigns the result to *l*. Notice that the decimal precision of *x* is lost when it is cast to a LongInt and assigned to *l*.

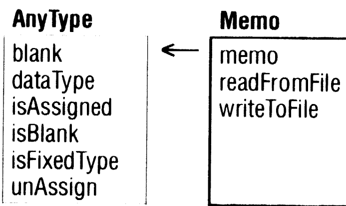
```
; convertToInt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
  l LongInt
endVar
```

```
x = 12.34             ; give x a value
x.view()              ; view x, title of dialog will be "Number"
l = longInt(x)        ; cast x as a LongInt and assign to l
l.view()              ; show l, note that decimal places are lost
                      ; displays 12
endmethod
```

See also

SmallInt type, Number type

Memo



Memos contain text and formatting data—up to 512MB in Paradox tables. Using Memo type methods **readFromFile** and **writeToFile,** you can transfer memos between forms (or reports), tables, and files.

You can also use the (=) operator to assign the value of a memo field to a Memo variable or a String variable.

Note There are no arithmetic or comparison operators for Memo variables.

If you assign a memo field to a String variable, you get only the memo text without any formatting. If you assign a memo field to a Memo variable, you get the text and the formatting.

The Memo type also includes methods defined for the AnyType type.

memo

Memo

Procedure

Casts a value as a Memo.

Syntax

memo (const ***value*** AnyType [, const ***value*** AnyType]*) String

Description

Casts (converts) the expression *value* to a Memo. If you specify multiple arguments, this method will cast all of them to Memos and concatentate them to one Memo.

Example

This example assumes that DOCFILES.DB exists and has an alpha field named Memo Name, a Date field named Memo Date, and a formatted memo field named Memo Data. For this example, a form has unbound fields named *stringObject* and *memoObject,* and a button named *getMemoData.* The code attached to *getMemoData's* **pushButton** method defines a TCursor to locate a particluar record in *DocFiles.* Then, the code casts and concatenates the contents of the three *DocFiles* fields to a String value, then to a Memo value. The value cast as a String is displayed in the *stringObject* object and the value cast as a Memo is displayed in the *memoObject* object. Note that when cast as

a String, formatting information is not displayed in *stringObject*. When cast as a Memo, *memoObject* displays all formatting information.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

if tc.open("DocFiles.db") then
  if tc.locate("Memo Name", "Project Notes") then
    ; this line casts data from three DOCFILES.DB fields as a String-
    ; because this is cast as a String, the data that appears in stringObject
    ; displays WITHOUT formatting
    stringObject.value = string(tc."Memo Name", "\t",
                                tc."Memo Date", "\n", tc."Memo Data")
    ; this line casts data from three DOCFILES.DB fields as a memo-
    ; because this is cast as a MEMO, the data that appears in memoObject
    ; displays with FORMATTED text
    memoObject.value = memo(tc."Memo Name", "\t",
                            tc."Memo Date", "\n", tc."Memo Data")
  else
    msgStop("Error", "Can't find Project Notes.")
  endif
else
  msgStop("Error", "Can't open DocFiles table.")
endif

endmethod
```

See also

- The Memo type in the *ObjectPAL Developer's Guide*

readFromFile

Memo

Method

Reads a memo from a file.

Syntax

readFromFile (const ***fileName*** String) Logical

Description

Reads a memo from a disk file specified in *fileName*. This method reads text only. It does not read the formatting of formatted memos.

Example

The following example reads the contents of a text file to a memo field in a table. Assume that a table named *PJNotes* exists in the current directory and has the following fields: ProjDate, a Date field, and ProjNotes, a Memo field. The **pushButton** method for a button named *getFile* opens, edits, and inserts a new record in the *PJNotes* table, then fills the ProjDate field with the current date, and fills the ProjNotes field with text from a file named NOTES.TXT.

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  MemoFile Memo
```

```
  pTC      TCursor
endVar

if pTC.open("pjNotes.db") then    ; open TCursor for PJNOTES.DB
  if MemoFile.readFromFile("notes.txt") then
    ; if memo file read was successful
    pTC.edit()                    ; edit project notes table
    pTC.insertRecord()            ; insert a new blank record
    pTC.ProjDate = today()        ; fill the ProjDate field
    pTC.ProjNotes = MemoFile      ; write memo to ProjNotes field
    pTC.endEdit()                 ; end Edit mode
  endif
  pTC.close()                     ; close the TCursor
endIf
endmethod
```

See also

- writeToFile

writeToFile

Memo

Method

Writes a memo to a file.

Syntax

writeToFile (const ***fileName*** String) Logical

Description

Writes a memo to a disk file specified in *fileName*. This method writes text only. It does not write the formatting of formatted memos.

Example

The following example writes the contents of a memo to a text file. Assume that a table named *PJNotes* exists in the current directory and has the following fields: *ProjDate*, a Date field, and *ProjNotes*, a Memo field. The **pushButton** method for a button named *writeFile* opens the *PJNotes* table, locates a record with the current date, then writes the contents of the *ProjNotes* field for that record to a file named NOTETDAY.TXT.

```
; getFile::pushButton
method pushButton(var eventInfo Event)
var
  MemoFile Memo
  pTC      TCursor
endVar
if pTC.open("pjNotes.db") then          ; open project notes table
  if pTC.locate("ProjDate", today()) then
    if NOT (pTC.ProjNotes = blank()) then  ; check if memo is blank
      MemoFile = pTC.ProjNotes             ; if not, write to MemoFile var
      MemoFile.writeToFile("notetday.txt") ; write MemoFile to text file
    endif
  endif
  pTC.close()                           ; close the TCursor
endIf
endmethod
```

See also

- readFromFile

Menu

Menu

addArray
addBreak
addPopUp
addStaticText
addText
contains
count
empty
getMenuChoiceAttribute
getMenuChoiceAttributeByID
hasMenuChoiceAttribute
remove
removeMenu
setMenuChoiceAttribute
setMenuChoiceAttributeByID
show

A Menu object is a list of items that appears in the application menu bar. When the user chooses an item from a menu, the text of that item is returned. Menus you build in ObjectPAL completely replace Paradox's built-in menus (but you can get them back using **removeMenu**).

By default, menus do not persist across forms; each form has its own menu system associated with it. If you create a menu for a form, the menu appears only when that form is active. If you then open a second form, the second form uses the built-in menus, not the menu you created for the first form. If you create a custom menu for each form, you can simulate context-sensitive menus in an application.

If you want two (or more) forms to display the same custom menu, set each form's StandardMenu property to Off. This instructs Paradox to retain the current menu when the user moves from one form to another. You can use the StandardMenu property to construct a single menu system for an entire application.

For more information about working with menus, refer to the *ObjectPAL Developer's Guide.*

Note A typical application uses both Menu objects and PopUpMenu objects. See also the PopUpMenu type later in this chapter.

addArray

Menu

Method

Appends elements of an array to a menu.

Syntax

addArray (const ***items*** Array[] String)

Description

Appends *items* from an array to a menu. The array *items* are displayed from left to right across the menu bar. To create a drop-down menu or a cascading menu, use **addPopUp**.

Example

This example constructs and displays an application menu bar when a form opens. This could be the application's main menu. Throughout the application, the menu displayed here can be changed by methods for other objects.

```
; thisForm::open
method open(var eventInfo Event)
var
  mMenu            Menu       ; main menu
  mmItems Array[3] String     ; main menu items
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ;menu appears when the form first opens
    mmItems[1] = "File"          ; fill the array
    mmItems[2] = "Edit"
    mmItems[3] = "Window"
    mMenu.addArray(mmItems)      ; same as mMenu.addText(...) 3 times
    mMenu.show()                 ; show the menu
endif
endmethod
```

See also

❐ addBreak, addPopUp, addStaticText, addText

addBreak

Menu

Method

Starts a new row in a menu.

Syntax

addBreak ( )

Description

Starts a new row in a menu. **addBreak** lets you explicitly "wrap" large menu constructs to two or more rows.

Example

This example constructs and displays an application menu bar when a form opens. It uses **addBreak** to add a second row on the menu bar.

```
; thisform::open
; method open(var eventInfo Event)
var
  mMenu Menu
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    ;menu appears when the form first opens
    mMenu.addText("File")
    mMenu.addText("Edit")
    mMenu.addBreak()
    mMenu.addText("About...")  ; this appears on the second row
    mMenu.show()               ; show the menu
endif
endmethod
```

See also

❒ addArray, addPopUp, addStaticText, addText

addPopUp

Menu

Method

Adds a pop-up menu to a menu bar item.

Syntax

addPopUp (const ***menuName*** String, const ***cascadedPopup*** PopUpMenu)

Description

Adds the heading *menuName* and a pop-up menu *cascadedPopup* to a menu. This method is useful for creating drop-down menus and cascading menus.

You must declare and create the pop-up menu before using **addPopUp**.

Example

The code in this example is attached to the built-in **arrive** method for each of two pages of a form. The **arrive** method for *pageOne* creates and displays a custom menu. The **arrive** method for *pageTwo* of the same form removes the custom menu. **addPopUp** is used to create a cascading pop-up menu and a drop-down menu.

Here is *pageOne*'s **arrive** method:

```
pageOne::arrive
method arrive(var eventInfo MoveEvent)
var
  p1, p2, p3  PopUpMenu
  m1          Menu
```

```
endVar

p1.addText("Passwords...")     ; add items to p1 popup
p1.addText("Attributes...")

p2.addText("Basic...")         ; add items to p2 popup
p2.addText("Scientific...")

p1.addPopUp("Calculator", p2) ; add another item to p1 popup,
                               ; and display p2 popup when the
                               ; item is selected

p3.addText("About...")         ; add an item to 3rd popup

m1.addPopUp("Utilities", p1)  ; add item to menu bar,
                               ; and drop-down p1 when selected
m1.addPopUp("Help", p3)       ; add item to menu bar,
                               ; and drop-down p3 when selected
m1.show()                      ; show the menu bar (not PopUpMenu)

endmethod
```

Here is *pageTwo*'s **arrive** method:

```
; pageTwo::arrive
method arrive(var eventInfo MoveEvent)
  removemenu()    ; remove the custom menu—the default menu
                  ; will appear instead
endmethod
```

See also

- addArray, addBreak

Menu

addStaticText

Method

Adds an unselectable text string to a menu.

Syntax

addStaticText (const ***item*** String)

Description

Appends *item* to a menu as unselectable text.

Example

In this example, code attached to a form's **open** method creates a menu bar. This example uses **addStaticText** to add a static menu item to the menu bar.

```
thisForm::open
method open(var eventInfo Event)
var
  mMenu Menu
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    mMenu.addStaticText("Main menu")    ; first item is static
```

```
    mMenu.addText("File")              ; add two more items
    mMenu.addText("Edit")
    mMenu.show()                       ; show the menu
endif
endmethod
```

See also

- addText

addText

Menu

Method

Adds a selectable text string to a menu.

Syntax

1. **addText** (const ***menuName*** String)
2. **addText** (const ***menuName*** String, const ***attrib*** SmallInt)
3. **addText** (const ***menuName*** String, const ***attrib*** SmallInt, const ***id*** SmallInt)

Description

Adds the item *menuName* to a menu. Menu items are displayed from left to right across the menu bar. You can use *attrib* to preset the display attribute of *menuName*. ObjectPAL provides constants (like MenuEnabled) for *attrib*; see MenuChoiceAttributes in the Constants dialog box.

In the third form of the syntax, you can specify an *id* number (a SmallInt) to identify the menu by number instead of by *menuName*. Then, in the built-in **menuAction** method, you use the *id* number to determine which menu the user chooses. When you specify a menu *id*, you should use the built-in constant UserMenu as a base constant, then add your own number to it. For example, the following line adds "File" to the *myMenu* menu and specifies an *id* number for that menu item:

```
myMenu.addText("File", MenuEnabled, UserMenu + 1)
```

For more information regarding user-defined constants, refer to Chapter 7 in the *ObjectPAL Developer's Guide*.

You can use an ampersand in an item to designate an accelerator key. For example, the item "&File" would appear as File, and the user could choose it by pressing *Alt+F*. If you rely on *menuName* to test for the user's choice, you need to include the ampersand in the comparison string. In this example, the return value is "&File", not "File".

If you want to right-align menu items, you can precede *menuName* with the string value "\008". Once you include "\008" in *menuName*, all subsequent menu items appear right-aligned; you don't have to

use "\008" again. For example, these lines display File on the left and Help and Utilities on the right:

```
myMenu.addText("File")
myMenu.addText("\008Help")
myMenu.addText("Utilities")
myMenu.show()
```

Example

The following examples demonstrate how **addText** syntax influences the way you test for the user's menu choice.

The first example uses the first form of **addText** syntax to create a simple menu. This example does not use *id* in the **addText** statements. The code attached to the built-in **menuAction** method must evaluate the string specified in *menuName* to determine the user's menu choice. The code that follows is attached to the **open** method for *pageOne*.

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu
utilPU.addText("&Time")
utilPU.addText("&Date")

 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu and right-align "Help" with \008
mainMenu.addText("\008&Help")

 ; now display the menu
mainMenu.show()

endmethod
```

The following code is attached to the **menuAction** method for *pageOne*. This code uses the **menuChoice** method to obtain the string value defined by *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

choice = eventInfo.menuChoice()  ; assign string value to choice

  ; now use choice value to determine which menu was selected
switch
  case choice = "&Time"  :
    msgInfo("Current Time", time())
  case choice = "&Date"  :
    msgInfo("Today's date", today())
  case choice = "\008&Help" :
    ; open the built-in help system
    action(EditHelp)
```

```
endSwitch

endmethod
```

The next example demonstrates how you can use the *id* clause with **addText** to refer to menu items by number instead of by name. This code establishes user-defined constants to make it easy to remember the menu *id* assignments. The following code goes in the Const window for *pageOne*:

```
; pageOne::Const
Const
  ; define constants for menu id's
  ; actual values (1, 2 and 3) are arbitrary
  TimeMenu = 1
  DateMenu = 2
  HelpMenu = 3
endConst
```

The following code is attached to the **open** method for *pageOne*. To control the menu display attributes, this code uses built-in constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (TimeMenu, DateMenu, and HelpMenu).

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
endVar

 ; build a pop-up menu and use constants (ie: TimeMenu)
 ; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, TimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, DateMenu + UserMenu)

 ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

 ; add "Help" to the menu bar and right-align "Help" with \008
mainMenu.addText("\008&Help", MenuEnabled, HelpMenu + UserMenu)

mainMenu.show()                ; display the menu

endmethod
```

The code that follows is attached to the **menuAction** method for *pageOne*. This method evaluates menu selections by *id* number rather than by the name specified in *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice SmallInt
endVar

choice = eventInfo.id()      ; assign constant value (ie: 900) to choice

  ; now use constants to determine which menu was selected
switch
```

```
  case choice = TimeMenu  + UserMenu :
    msgInfo("Current Time", time())
  case choice = DateMenu  + UserMenu :
    msgInfo("Today's Date", today())
  case choice = HelpMenu  + UserMenu :
    ; open the built-in help system
    action(EditHelp)
endSwitch

endmethod
```

See also

- addArray, addStaticText

contains

Menu

Method

Reports whether an item is in a menu.

Syntax

contains (const ***item*** AnyType) Logical

Description

Returns True if *item* is in the list of items in a menu; otherwise, it returns False. **contains** is case sensitive.

Example

This example assumes that a multi-record object is on the form. When the user changes the value in a field contained in the multi-record object, an Undo menu item is added to the existing custom menu bar. When the user moves to another record, Undo is removed. The example uses **contains** to determine if Undo is present before adding or removing the item. The menu variable is defined in the form's Var window. The menu bar is created by the form's **open** method.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
endVar
```

The following code is for the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    m1.addText("&Insert")
    m1.addText("&Delete")
    m1.show()              ; show two item menu
endif
endmethod
```

The following code is for the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)
if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  switch
      ; when user locks a record (starts to change a field value)
    case eventInfo.id() = DataLockRecord :
      ; add Undo and redisplay the menu
      m1.addText("&Undo")
      m1.show()

      ; when user posts the record (moves to another record)
    case eventInfo.id() = DataUnlockRecord
      ; remove Undo redisplay the menu
      m1.remove("&Undo")
      m1.show()
  endswitch

endif
endmethod
```

The following code is for the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  choice = eventInfo.menuChoice()

  switch
    case choice = "&Insert" :
      active.action(DataInsertRecord) ; insert new record
    case choice = "&Delete" :
      active.action(DataDeleteRecord) ; delete current record
    case choice = "&Undo"   :
      active.action(DataCancelRecord) ; revert record to original state
      m1.remove("&Undo")              ; remove Undo menu item
      m1.show()                       ; redisplay menu without Undo
  endswitch

endif
endmethod
```

See also

- count

---

count

Menu

Method

Returns the number of items in a menu.

Syntax

count ( ) SmallInt

Description

Returns the number of items in a menu, including separators, bars, and breaks.

count returns the number of items in a single menu. If you attach a pop-up menu to a menu bar item with **addPopUp**, **count** returns the number of items in the pop-up menu or the number of items in the menu bar, but not the total number of items in both menus.

Example

The following example constructs a menu and a pop-up menu, then displays the number of items in each menu. Note that **count** returns the number of items in a menu whether or not the menu is displayed.

```
; countMenus::pushButton
method pushButton(var eventInfo Event)
var
  m Menu
  p PopUpMenu
endVar

p.addText("&One")
p.addBar()
p.addText("T&wo")
p.addText("Th&ree")        3 items + 1 bar   4 elements

m.addText("&First")
m.addText("&Second")
m.addPopUp("&Third", p)      3 items in menu bar

msgInfo("Menu bar items", m.count()) ; displays 3  counts menu bar only
msgInfo("Pop-up items", p.count())   ; displays 4  counts pop up only

endmethod
```

See also

- contains

empty

Menu

Method

Removes all items from a menu

Syntax

empty ( )

Description

Removes all items from a custom menu. Use **empty** when you need to clear an existing menu before rebuilding it.

Example

The following example uses two buttons to display alternate menus. Both methods affect the same menu, declared with the variable *mainMenu* in the form's Var window.

The following code goes in the form's Var window:

```
; thisForm::Var
Var
  mainMenu Menu  ; custom menu bar
endVar
```

Following is the code for *showMenuOne*'s **pushButton** method:

```
; showMenuOne::pushButton
method pushButton(var eventInfo Event)
  mainMenu.empty()          ; clear the menu
  mainMenu.addText("&One")  ; reconstruct it
  mainMenu.addText("&Two")
  mainMenu.show()           ; display the changed menu
endmethod
```

Following is the code for *showMenuTwo*'s **pushButton** method:

```
; showMenuTwo::pushButton
method pushButton(var eventInfo Event)
  mainMenu.empty()              ; clear the menu
  mainMenu.addText("File")      ; reconstruct it
  mainMenu.addText("Edit")
  mainMenu.show()               ; show it again
endmethod
```

See also

- remove

getMenuChoiceAttribute

Menu

Procedure

Reports the display attributes of a menu item.

Syntax

getMenuChoiceAttribute (const ***menuChoice*** String) SmallInt

Description

Returns an integer representing the display attributes of the menu item specified in *menuChoice*. The integer value represents the combination of attributes that apply. Use **getMenuChoiceAttribute** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

ObjectPAL provides constants (like MenuEnabled) for display attributes; see MenuChoiceAttributes in the Constants dialog box.

This procedure returns the attribute of the currently displayed menu; if you have not created a custom menu, **getMenuChoiceAttribute** operates on the built-in menu.

Example

In this example, the **open** method for *pageOne* constructs and displays a simple menu. The *getMenuState* button reports whether or not the Time menu item is enabled.

The following code is attached to the **open** method for *pageOne*:

```
; pageOne::open
method open(var eventInfo Event)
var
  mainMenu Menu
  utilPU PopUpMenu
  attrib SmallInt
endVar

 ; build a pop-up menu, disable Time option
utilPU.addText("&Time", MenuDisabled + MenuGrayed)
utilPU.addText("&Date")
 ; attach pop-up and show the menu bar
mainMenu.addPopUp("&Utilities", utilPU)
mainMenu.addText("&Help")
mainMenu.show()

endmethod
```

The following code is for *getMenuState's* **pushButton** method:

```
; getMenuState::pushButton
method pushButton(var eventInfo Event)
var
  attrib SmallInt
endVar

  ; store attributes of Time in attrib
attrib = getMenuChoiceAttribute("&Time")
  ; this displays False because Time is enabled
msgInfo("Time enabled?", HasMenuChoiceAttribute(attrib, MenuEnabled))
  ; this displays True because Time is grayed
msgInfo("Time grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))

endmethod
```

See also

- getMenuChoiceAttributeById, hasMenuChoiceAttribute, setMenuChoiceAttribute, setMenuChoiceAttributeById

getMenuChoiceAttributeById — Menu

Procedure

Reports the display attribute of a menu item specified by its menu ID.

Syntax

getMenuChoiceAttributeById (const ***menuId*** SmallInt) SmallInt

Description

Returns an integer representing the display attributes of the menu item specified in *menuId.* The integer value represents the combination of attributes that apply. Use **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

ObjectPAL provides constants (like MenuEnabled) for menu attributes; see MenuChoiceAttributes in the Constants dialog box.

This procedure returns the attribute of the currently displayed menu; if you have not created a custom menu, **getMenuChoiceAttributeById** operates on the built-in menu.

This procedure is similar to **getMenuChoiceAttribute** in that both report the display attributes for a specified menu item. The difference is that you specify the actual menu ID (a SmallInt value) for **getMenuChoiceAttributeById**, and the menu name (a String value) for **getMenuChoiceAttribute**. **getMenuChoiceAttributeById** is especially useful when you specify a menu ID as part of **addText** syntax.

Example

The following example demonstrates how you can use **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a menu item is disabled. In this example, the **open** method for *pageOne* constructs a small menu. The **pushButton** method for the *getMenuState* button reports on the state of the Undo menu item.

The following code goes in the form's Var window:

```
; thisForm::Var
Var
  m1     Menu
  p1, p2 PopUpMenu
endVar
```

The following code goes in the form's Const window:

```
; thisForm::Const
Const
  UndoMenu  = 1
  InsMenu   = 2
  DelMenu   = 3
  IndexMenu = 4
  AboutMenu = 5
endConst
```

The following code is for the page's **open** method:

```
; pageOne::open
method open(var eventInfo Event)

p1.addText("Undo",   MenuDisabled + MenuGrayed, UndoMenu + UserMenu)
p1.addText("Insert", MenuEnabled, InsMenu + UserMenu)
p1.addText("Delete", MenuEnabled, DelMenu + UserMenu)
p2.addText("Index",  MenuEnabled, IndexMenu + UserMenu)
p2.addText("About",  MenuEnabled, AboutMenu + UserMenu)

m1.addPopUp("&Record", p1)
m1.addPopUp("&Help", p2)
m1.show()

endmethod
```

The following code is attached to the *getMenuState's* **pushButton** method:

```
; getMenuState::pushButton
method pushButton(var eventInfo Event)

  ; store attributes of Undo menu in attrib
attrib = getMenuChoiceAttributeById(UndoMenu + UserMenu)

  ; this displays False because Undo is disabled
msgInfo("Undo enabled?", hasMenuChoiceAttribute(attrib, MenuEnabled))

  ; this displays True because Undo is grayed
msgInfo("Undo grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))

endmethod
```

See also

- getMenuChoiceAttribute, getMenuChoiceAttributeById, hasMenuChoiceAttribute, setMenuChoiceAttribute

hasMenuChoiceAttribute

Menu

Procedure

Reports whether a menu item contains a given display attribute.

Syntax

hasMenuChoiceAttribute (const ***attrib*** SmallInt , const ***attribSet*** SmallInt) Logical

Description

Returns True if *attribSet* contains the attribute specified in *attrib*; otherwise, it returns False.

Use **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** or **getMenuChoiceAttributeById** to determine whether a particular display attribute for a menu item is represented in *attribSet*.

ObjectPAL provides constants (like MenuEnabled) for *attrib*; see MenuChoiceAttributes in the Constants dialog box.

Example

The following code demonstrates how you can use **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** to determine whether a particular attribute applies to the currently displayed menu. The following code is attached to the **open** method for *pageOne*.

```
; pageOne::open
method open(var eventInfo Event)
var
  m1 Menu
  p1 PopUpMenu
endVar

p1.addText("&Insert")    ; create a simple menu
p1.addText("&Delete")
p1.addText("&Undo")
m1.addPopUp("&Record", p1)
```

```
m1.show()
endmethod
```

The following code is attached to the **pushButton** method for the *toggleMenuState* button:

```
; toggleMenuState::pushButton
method pushButton(var eventInfo Event)
var
  attribSet SmallInt
endVar

 ; store composite menu attributes in attribSet
attribSet = getMenuChoiceAttribute("&Undo")

 ; this is True if Undo is enabled
if hasMenuChoiceAttribute(attribSet, MenuEnabled) then
   setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
else
  setMenuChoiceAttribute("&Undo", MenuEnabled)
endif

endmethod
```

See also

- getMenuChoiceAttribute, getMenuChoiceAttributeById

remove

Menu

Method

Removes an item from a menu.

Syntax

remove (const ***item*** String)

Description

Deletes the first occurrence of *item* from a menu. This method is useful for changing one item in a menu without having to rebuild the entire menu.

Example

The code shown in this example changes a menu immediately by removing an item and adding another item in its place.

```
; changeMenu::pushButton
method pushButton(var eventInfo Event)
var
   mainMenu Menu
endVar

; First, assume the user is working with a form.
; You could display a menu like this:
mainMenu.addText("File")
mainMenu.addText("Edit")
mainMenu.addText("Form")
mainMenu.show()
msgInfo("Status", "About to change menus. Watch closely.")

; Then, suppose the user switches to work on a report.
; You could change the menu like this:
```

```
mainMenu.remove("Form")
mainMenu.addText("Report")
mainMenu.show()

endmethod
```

See also

- contains, empty

removeMenu

Menu

Procedure

Removes a custom menu and restores the default menu.

Syntax

removeMenu ( )

Description

Replaces a menu built using ObjectPAL with Paradox's default menu.

Example

In the following example, the form's **open** method constructs a menu (but does not display it). The **arrive** method for *pageOne* displays the menu with **show**. The **arrive** method for *pageTwo* removes the menu and reveals the built-in Paradox menu.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
endVar
```

The following code is attached to the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

     m1.addText("&File")    ; construct a menu
     m1.addText("&Edit")
     m1.addText("For&m")

endif

endmethod
```

The following code is attached to the **arrive** method for *pageOne*:

```
; pageOne::arrive
method arrive(var eventInfo MoveEvent)
m1.show()  ; display the application menu
endmethod
```

The following code is attached to the **arrive** method for *pageTwo*:

```
; pageTwo::arrive
method arrive(var eventInfo MoveEvent)
removeMenu()    ; remove application menu, reveal built-in menu
endmethod
```

See also

- empty, remove

setMenuChoiceAttribute

Menu

Procedure

Sets the display attribute of a menu item.

Syntax

setMenuChoiceAttribute (const ***menuChoice*** String, const ***menuAttribute*** SmallInt)

Description

Sets the display attribute of *menuChoice* to *menuAttribute*. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttribute** affects the built-in menu.

ObjectPAL provides constants (like MenuGrayed) for *menuAttribute*; see MenuChoiceAttributes in the Constants dialog box.

Example

In the following example, you change the attribute of the Undo option, depending on whether there is anything to undo. As the user makes changes to the record, the Undo item becomes selectable. After posting the changes, Undo is unavailable.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
  p1 PopUpMenu
endVar
```

The following code is for the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    ; create a menu and show it
    p1.addText("&Undo", MenuDisabled + MenuGrayed)
    p1.addText("&Insert")
    p1.addText("&Delete")
    m1.addPopUp("&Record", p1)
    m1.show()
```

```
endif
endmethod
```

The following code is for the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    switch
        ; when user locks a record (starts to change a field value)
      case eventInfo.id() = DataLockRecord :
        ; enable Undo menu item
        setMenuChoiceAttribute("&Undo", MenuEnabled)

        ; when user posts the record (moves to another record)
      case eventInfo.id() = DataUnlockRecord :
        ; disable and gray Undo menu item
        setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
    endswitch

  else
    ;code here executes just for form itself
  endif

endmethod
```

The following code is for the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form

    choice = eventInfo.menuChoice()
    switch
      case choice = "&Insert" :
        active.action(DataInsertRecord)  ; insert new record
      case choice = "&Delete" :
        active.action(DataDeleteRecord)  ; delete current record
      case choice = "&Undo"   :
        active.action(DataCancelRecord)  ; revert record to original state
        setMenuChoiceAttribute("&Undo", MenuDisabled + MenuGrayed)
    endswitch

  else
    ;code here executes just for form itself
endif

endmethod
```

See also

- getMenuChoiceAttribute, getMenuChoiceAttributeById, hasMenuChoiceAttribute, setMenuChoiceAttributeById

Menu

setMenuChoiceAttributeById

Menu

Procedure

Sets the display attribute of a menu item.

Syntax

setMenuChoiceAttributeById (const ***menuId*** SmallInt, const ***menuAttribute*** SmallInt)

Description

Sets the display attribute of *menuId* to *menuAttribute*. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttributeById** affects the built-in menu.

ObjectPAL provides constants (like MenuGrayed) for *menuAttribute*; see MenuChoiceAttributes in the Constants dialog box.

Example

In the following example, you change the attribute of the Undo option, depending on whether there is anything to undo. As the user makes changes to the record, the Undo item becomes selectable. After posting the changes, Undo is unavailable. This example uses the *menuId* clause in **addText** so that the code can refer to menu items by number rather than menu name.

The following code goes in the form's Var window:

```
; thisForm::var
Var
  m1 Menu
  p1 PopUpMenu
endVar
```

The following code goes in the form's Const Window:

```
; thisForm::const
Const
  InsMenu  = 1  ; use constants for menu IDs
  DelMenu  = 2
  UndoMenu = 3
endConst
```

The following code is attached to the form's **open** method:

```
; thisForm::open
method open(var eventInfo Event)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

     ; construct a menu and display it
     p1.addText("&Undo",   MenuDisabled + MenuGrayed, UndoMenu + UserMenu)
     p1.addText("&Delete", MenuEnabled, DelMenu + UserMenu)
     p1.addText("&Insert", MenuEnabled, InsMenu + UserMenu)
     m1.addPopUp("&Record", p1)
```

```
    m1.show()
  endif
endmethod
```

The following code is attached to the form's **action** method:

```
; thisForm::action
method action(var eventInfo ActionEvent)

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    switch
         ; when user locks a record (starts to change a field value)
      case eventInfo.id() = DataLockRecord :
         ; enable Undo menu item
        setMenuChoiceAttributeById(UndoMenu, MenuEnabled)
         ; when user posts the record (moves to another record)
      case eventInfo.id() = DataUnlockRecord :
         ; disable and dim Undo menu item
        setMenuChoiceAttributeById(UndoMenu, MenuGrayed + MenuDisabled)
    endswitch
  else
    ;code here executes just for form itself
endif
endmethod
```

The following code is attached to the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuItem SmallInt
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form
  menuItem = eventInfo.id()
  switch
    case menuItem = InsMenu :
      active.action(DataInsertRecord)    ; insert new record
    case menuItem = DelMenu :
      active.action(DataDeleteRecord)    ; delete current record
    case menuItem = UndoMenu :
      active.action(DataCancelRecord)    ; revert record to original state
      setMenuChoiceAttributeById(UndoMenu, MenuDisabled + MenuGrayed)
  endswitch
else
  ;code here executes just for form itself
endif

endmethod
```

See also

- getMenuChoiceAttribute, getMenuChoiceAttributeById, hasMenuChoiceAttribute, setMenuChoiceAttribute

show

Menu

Method

Displays a menu.

Syntax

show ( )

Description

Displays a menu.

The user's choice is handled using the built-in **menuAction** method and **menuChoice** from the MenuEvent type. Refer to Chapter 7 in the *ObjectPAL Developer's Guide* for more information about working with menus.

Example

In this example, a form's **open** method constructs a simple menu, then displays it with **show**. The **menuAction** method for the form handles the user's menu choice. Following is the code attached to the **open** method for *thisForm*.

```
; thisForm::open
method open(var eventInfo Event)
var
  p1 PopUpMenu
  m1 Menu
endVar

if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself

    p1.addText("&Time")           ; construct a pop-up
    p1.addText("&Date")
    m1.addPopUp("&Utilities", p1) ; attach pop-up to menu item
    m1.show()                     ; display the m1 menu

endif

endmethod
```

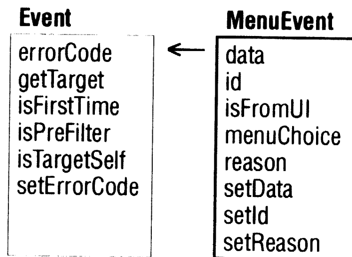
The following code is attached to the form's **menuAction** method:

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuName String
endVar

if eventInfo.isPreFilter() then
  ;code here executes for each object in form

  menuName = eventInfo.menuChoice()
  switch
    case menuName = "&Time" : msgInfo("Current Time", time())
    case menuName = "&Date" : msgInfo("Today's Date", date())
  endSwitch

else
  ;code here executes just for form itself
endif

endmethod
```

See also

- addText

MenuEvent



MenuEvent variables contain data related to menu selections in the application menu bar. When the user chooses an item from a menu, it triggers the **menuAction** method. By modifying an object's built-in **menuAction** method, you can define how the object responds. Built-in methods are discussed in Chapter 2. For information about the Event model, see the *ObjectPAL Developer's Guide*.

The MenuEvent type includes several methods defined for the Event type.

For more information and examples, refer to Chapter 6 of the *ObjectPAL Developer's Guide*.

data

MenuEvent

Method

Returns information about a MenuEvent.

Syntax

data **( )** LongInt

Description

This method should be used by Windows programmers only. **data** returns the *lParam* argument (usually zero) of specific Windows messages, such as WM_SYSCOMMAND and WM_COMMAND. See your Windows programming documentation for more information.

See also

❒ id, setData, setId

id

Beginner

MenuEvent

Method

Returns the ID of a MenuEvent.

Syntax

id ( ) SmallInt

Description

Returns the ID number of a MenuEvent. ObjectPAL provides constants (like MenuFileOpen) for many common menu choices; see MenuCommands in the Constants dialog box.

Example

This is attached to a form's built-in **menuAction** method. When the user selects Close from the Control menu, attempts to toggle to a design window, or chooses File | Exit, the method asks the user to confirm whether or not to leave the form.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  theFile String
  tv TableView
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    if eventInfo.id() = MenuControlClose OR
      eventInfo.id() = MenuFileExit OR
      eventInfo.id() = MenuFormDesign then
      disableDefault                   ; block departure
      ans = msgQuestion("Please confirm", "Do you really want to leave?")
      if ans = "Yes" then
        dodefault
      endif
    endif
endif
endmethod
```

The next example demonstrates how you can use the menu ID argument with **addText** to refer to menu items by number (ideally, user-defined constants) instead of by name. This code establishes user-defined constants to make it easy to remember the menu ID assignments. The following code defines constants global to *pageOne*.

```
; pageOne::Const
Const
  ; define constants for menu IDs
  ; actual values (1, 2 and 3) are arbitrary
  TimeMenu = 1
  DateMenu = 2
  HelpMenu = 3
endConst
```

The following code is attached to the **open** method for *pageOne*. To control the menu display attributes, this code uses built-in constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (TimeMenu, DateMenu, and HelpMenu).

```
; pageOne::open
method open(var eventInfo Event)
var
```

```
  mainMenu Menu
  utilPU PopUpMenu
endVar

  ; build a pop-up menu and use constants (ie: TimeMenu)
  ; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, TimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, DateMenu + UserMenu)
  ;UserMenu is an ObjectPAL constant
  ; attach pop-up to the Utilities main menu item
mainMenu.addPopUp("&Utilities", utilPU)

  ; add "Help" to the menu bar and right-justify "Help" with \008
mainMenu.addText("\008&Help", MenuEnabled, HelpMenu + UserMenu)

mainMenu.show()                    ; display the menu

endmethod
```

The following code is attached to the **menuAction** method for *pageOne*. This method evaluates menu selections by ID number rather than by the name specified in *menuName*.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice SmallInt
endVar

choice = eventInfo.id()      ; assign constant value to choice

  ; now use constants to determine which menu was selected
switch
  case choice = TimeMenu  + UserMenu :
    msgInfo("Current Time", time())
  case choice = DateMenu  + UserMenu :
    msgInfo("Today's Date", today())
  case choice = HelpMenu  + UserMenu :
    ; change menu ID to built-in constant (MenuHelpContents)--
    ; this effectively opens the built-in help system.
    eventInfo.setId(MenuHelpContents)
    eventInfo.setReason(MenuDesktop)
endSwitch

endmethod
```

See also

- setId

isFromUI

MenuEvent

Method

Reports whether a MenuEvent was generated by the user interacting with Paradox.

Syntax

isFromUI ( ) logical

Description

Reports whether a MenuEvent was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True if the MenuEvent was generated by the user; otherwise, it returns False.

See also

- isPreFilter in the Event type

---

menuChoice

MenuEvent

Beginner

Method

Returns a string containing an item chosen from a menu.

Syntax

menuChoice ( ) String

Description

Returns a string containing an item chosen from a menu. Use **menuChoice** to modify an object's built-in **menuAction** method to specify how that object responds to menu choices.

Note If a menu item's definition includes an accelerator key (for example "&Print"), remember to include the ampersand in the comparison string, for instance, the following code compares the return value of **menuChoice** with the string "&Print":

```
if eventInfo.menuChoice()   "&Print" then
   ; print the report
endif
```

Example

This example assumes a form contains at least one memo field, named *thisMemoField*. When the user arrives on *thisMemoField*, the built-in **arrive** method displays a menu that lets the user perform basic cut and paste operations. The built-in **menuAction** method attached to *thisMemoField* uses **menuChoice** to evaluate the user's selection, and take appropriate action. Although this example mimics the behavior of the default menus, this technique is necessary when the default menus are replaced by custom menus.

This code is attached to the built-in **arrive** method for *thisMemoField*:

```
; thisMemoField::arrive
method arrive(var eventInfo MoveEvent)
Var
  EditPopUp PopUpMenu
  EditMenu  Menu
endVar

EditPopUp.addText("&Cut")               ; create a pop-up menu
EditPopUp.addText("&Copy")
EditPopUp.addText("&Paste")

EditMenu.addPopUp("&Edit", EditPopUp)   ; add pop-up menu bar item
```

```
EditMenu.show()                        ; display the menu
endmethod
```

This code is attached to the the built-in **menuAction** method for *thisMemoField*. Note that comparisons in the **switch...endSwitch** statement must include the ampersand, such as "&Cut".

```
thisMemoField::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar
choice = eventInfo.menuChoice()       ; store the menu selection to choice

; now respond to the selection appropriately
switch
  case choice = "&Cut"   : self.action(EditCutSelection)
  case choice = "&Copy"  : self.action(EditCopySelection)
  case choice = "&Paste" : self.action(EditPaste)
endSwitch
endmethod
```

This code is attached to the built-in **depart** method for *thisMemoField*. When the user leaves *thisMemoField*, this code removes the menu. In this example, the default menus reappear when the user moves off the field. In a similar situation, you might want to display another custom menu structure.

```
; thisMemoField::depart
method depart(var eventInfo MoveEvent)
removeMenu()          ; remove the Edit menu
endmethod
```

MenuEvent

See also

- id, setId

reason

MenuEvent

Method

Reports the type of menu chosen.

Syntax

reason ( ) SmallInt

Description

Returns an integer value to report why a MenuEvent occurred. MenuEvent Reason constants occur when a built-in **menuAction** method is called. ObjectPAL provides the following constants for testing the value returned by **reason** (also listed in the Constants dialog box under MenuReason):

- MenuNormal means the MenuEvent was caused by a menu command or SpeedBar button that changes depending on which window you're using.

- MenuControl means the MenuEvent was caused by a System menu command or by the maximize or minimize button.
- MenuDesktop means the means the MenuEvent was caused by one of the basic desktop menu commands.

Example

In this example, the form's **menuAction** method examines every MenuEvent to determine the Reason constant for the MenuEvent. The Reason constant is then displayed in the *menuReasonField* field object.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
var
  reasonStr String
endVar
  if eventInfo.isPreFilter() then
    ; sort out the reason, and assign equivalent string to reasonStr
    reasonStr = iif(eventInfo.reason() = MenuNormal, "MenuNormal",
                iif(eventInfo.reason() = MenuControl, "MenuControl",
                 "MenuDesktop"))
    reasonId = eventInfo.reason()
    menuReasonField = String(reasonId) + " " + reasonStr
      ; Code here executes before each object
  else
      ; Code here executes afterwards (or for form)

  endif
endmethod
```

See also

- setReason

---

setData

MenuEvent

Method

Specifies information about a MenuEvent.

Syntax

setData (const ***menuData*** LongInt)

Description

This method should be used by Windows programmers only. **setData** specifies the *lParam* argument (usually zero) of specific Windows messages, such as WM_SYSCOMMAND and WM_COMMAND. See your Windows programming documentation for more information.

See also

- data, id, setId

setId

MenuEvent

Method

Specifies the ID of a MenuEvent.

Syntax

setId (const ***actionId*** SmallInt)

Description

Specifies in *actionId* an action to take as the result of a menu choice. ObjectPAL defines constants for *actionId*; see MenuCommands in the Constants dialog box.

If you change the ID for a MenuEvent with **setId**, you may also need to change the reason for that MenuEvent with **setReason**.

Note In many circumstances, you should use **menuAction** from the Form type or UIObject type to invoke a menu command. Although it is possible to change the Reason and ID for an existing MenuEvent (*eventInfo*), and it is also possible to create a new MenuEvent and set the Reason and ID for that event (only advanced users should try this), this technique is not always advisable.

Example

See the example for id.

See also

- id
- errorCode, setErrorCode in the Event type
- menuAction in the Form type
- menuAction in the UIObject type

setReason

MenuEvent

Method

Specifies a reason for generating a MenuEvent.

Syntax

setReason (const ***reasonId*** SmallInt)

Description

Specifies a reason for generating a MenuEvent. This method takes one of the following constants as an argument (listed in the Constants dialog box under MenuReasons):

- MenuNormal means the MenuEvent was caused by a menu command or SpeedBar button that changes depending on which window you're using.
- MenuControl means the MenuEvent was caused by a Control menu command or by the maximize or minimize button.

- MenuDesktop means the MenuEvent was caused by a one of the basic desktop menu commands.

Note In many circumstances, you should use **menuAction** from the Form type or UIObject type to invoke a menu command. Although it is possible to change the reason and ID for an existing MenuEvent (*eventInfo*), and it is also possible to create a new MenuEvent and set the reason and ID for that event (only advanced users should try this), this technique is not always advisable.

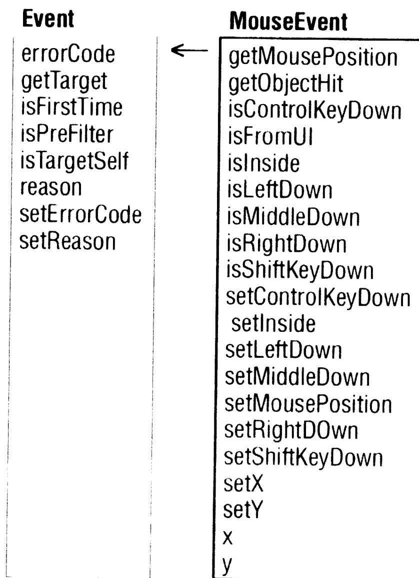
Example

See the example for id.

See also

- reason
- menuAction in the Form type
- menuAction in the UIObject type

MouseEvent



A MouseEvent object answers questions about the mouse, including

- Where is the mouse?
- Was a mouse button clicked?
- Which mouse button was clicked or held down during an operation?

The MouseEvent type includes several methods defined for the Event type.

The following built-in methods are triggered by MouseEvents: **mouseClick**, **mouseDown**, **mouseUp**, **mouseDouble**, **mouseRightUp**, **mouseRightDown**, **mouseRightDouble**, **mouseMove**, **mouseEnter**, and **mouseExit**. These methods, along with the rest of the built-in methods, are discussed in Chapter 2. For information about the event model, see the *ObjectPAL Developer's Guide*.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide*.

getMousePosition

MouseEvent

Method

Reports on the mouse position.

Syntax

1. **getMousePosition** (var ***p*** Point)
2. **getMousePosition** (var ***xPosition*** LongInt, var ***yPosition*** LongInt)

Description

Writes the mouse position to a variable. Syntax 1 stores the value in a Point variable, *p*. Syntax 2 stores the value in *xPosition* and *yPosition,* two LongInt variables representing the x- and y-coordinates of the mouse pointer. When you use Syntax 1, you can use Point type methods (for example, **isLeft** and **isRight**) to get more information.

This method gets the mouse position at the time of the MouseEvent. The current mouse position may be different.

Example

This example gets the position of the last **mouseUp** event and draws a small circle at that position. The method first checks if the source of the event was from the UI (in this case, from the user), and if the target of the event is the page itself (as opposed to whether it was bubbled up to the page from some other object). This method draws the circle only when the user clicks on the page.

```
; pageOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  crObj UIObject
  x, y  LongInt    ; point coordinates
endVar
if eventInfo.isFromUI() AND eventInfo.isTargetSelf() then
  ; create a small blue circle at the mouse position
  eventInfo.getMousePosition(x, y)
  crObj.create(ellipseTool, x, y, 100, 100)
  crObj.Color = DarkBlue
  crObj.Visible = True
endif
endmethod
```

See also

- getObjectHit, setMousePosition
- getTarget in the Event type

getObjectHit

MouseEvent

Method

Creates a handle to the UIObject that received the event.

Syntax

getObjectHit (var ***target*** UIObject) Logical

Description

Returns in *target* a handle to the target of the event. This method is useful for the internal MouseEvents that call **mouseExit** and **mouseEnter**. **getObjectHit** can return a different object than **getTarget** during a **mouseExit** or **mouseEnter** method.

Example

The following method is attached to the **mouseExit** method of a form. When the mouse exits an object, a message appears in the status window showing the name of the target object (**getTarget**) vs. the name of the object hit (**getObjectHit**).

```
; thisForm::mouseExit
method mouseExit(var eventInfo MouseEvent)
var
  targObj,
  hitObj   UIObject
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    eventInfo.getTarget(targObj)
    eventInfo.getObjectHit(hitObj)
    message(targObj.Name + " vs. " + hitObj.Name)
  else
    ;code here executes just for form itself

endif
endmethod
```

See also

- getMousePosition
- getTarget in the Event type

isControlKeyDown

MouseEvent

Method

Reports whether the user has held (or is holding) down *Ctrl* during a MouseEvent.

Syntax

isControlKeyDown ( ) Logical

Description

Returns True if *Ctrl* is held down during a MouseEvent; otherwise, it returns False.

Example

This example examines the keyboard state during a mouse click to determine whether to automatically insert the highest value in the range, the lowest value in a range, or the default value. The following constants are declared in the Const window for *fieldOne*:

```
; fieldOne::Const
Const
  HighRangeVal = Number(10000)
  LowRangeVal = Number(100000)
```

```
  DefaultVal = Number(50000)
endConst
```

This is the method for **mouseUp** for *fieldOne*:

```
; fieldOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
; insert high, low, or default value depending on how mouse was clicked
switch
  case eventInfo.isControlKeyDown() : self.Value = LowRangeVal
                                      message("Ctrl-click")
  case eventInfo.isShiftKeyDown()   : self.Value = HighRangeVal
                                      message("Shift-click")
  otherwise                         : self.Value = LowRangeVal
                                      message("Click")
endswitch
endmethod
```

See also

- isShiftKeyDown, setControlKeyDown

---

isFromUI

MouseEvent

Method

Reports whether a MouseEvent was generated by the user interacting with Paradox.

Syntax

isFromUI ( ) logical

Description

Reports whether an event was generated by the user interacting with Paradox, or internally (for example, by an ObjectPAL statement). This method returns True if the event was generated by the user; otherwise, it returns False.

See also

- isPreFilter in the Event type

---

isInside

MouseEvent

Method

Reports whether the mouse is inside the border of the target object.

Syntax

isInside ( ) Logical

Description

Reports whether the mouse is inside the border of the target object at the time of the event.

Example

In this example, the **mouseUp** method for *buttonOne* reports whether the last event is inside the borders of the target object. If you click *buttonOne*, the **mouseUp** MouseEvent is delivered to *buttonOne* and

isInside returns True. If you drag from inside the button to outside the button, so that the **mouseUp** occurs outside of the borders of *buttonOne*, the MouseEvent occurs for *buttonOne*, and triggers the **mouseUp** method, but **isInside** returns False for that MouseEvent.

```
; buttonOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
msgInfo("Is the last event inside ?", eventInfo.isInside())
endmethod
```

See also

- getObjectHit, setInside

isLeftDown

MouseEvent

Method

Reports whether the left (or primary) mouse button is held down during a MouseEvent.

Syntax

isLeftDown ( ) Logical

Description

Returns True if the left mouse button is held down during a MouseEvent, for instance, while dragging the mouse; otherwise, it returns False.

Example

In the following example, assume that the *Site Notes* field from the *Sites* table is placed on a form. This method, attached to the **mouseMove** method for *Site Notes*, checks whether the left or right button is down at the time of the move. If the left button is down, the field is selected from the point of the click to the beginning of the field. If the right button is down, the field is selected from the point of the click to the end of the field.

```
; Site Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)          ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)     ; select from point to end
  endif
endif
endmethod
```

See also

- isMiddleDown, isRightDown, setLeftDown

MouseEvent

isMiddleDown

MouseEvent

Method

Reports whether the middle mouse button is held down during a MouseEvent.

Syntax

isMiddleDown ( ) Logical

Description

Returns True if the middle mouse button is held down during a MouseEvent; otherwise (even if there is no middle mouse button), it returns False.

Example

This example assumes that a form contains a button called *sendMove*, and a field from the Sites table called *Site Notes*. The **pushButton** method for *sendMove* constructs a MouseEvent with the middle button down, then sends the MouseEvent off to the *Site Notes* field.

```
; sendMove::pushButton
method pushButton(var eventInfo Event)
var
  mo  MouseEvent          ; declare a MouseEvent to send
  ui  UIObject
endVar
ui.attach("Site Notes")  ; attach to Site Notes
mo.setMiddleDown(Yes)    ; set middle button down on MouseEvent
ui.mouseMove(mo)         ; dispatch event to mouseMove for Site Notes
endmethod
```

This method is attached to the **mouseMove** method for *Site Notes*. If the middle button is down for the MouseEvent, the method moves to the beginning of the current word, then selects the entire word.

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isMiddleDown() then
  self.action(MoveLeftWord)    ; go to the beginning of the word
  self.action(SelectRightWord) ; select the entire word
endif
endmethod
```

See also

- isLeftDown, isRightDown, setMiddleDown

isRightDown

MouseEvent

Method

Reports whether the right mouse button is pressed during a MouseEvent.

Syntax

isRightDown ( ) Logical

Description

Returns True if the right (or alternate) mouse button is held down during a MouseEvent, for instance, while right-dragging; otherwise, it returns False.

Example

In the following example, assume that the *Site Notes* field from the *Sites* table is placed on a form. The **mouseMove** method for *Site Notes* checks whether the left or right mouse button is down at the time of the move. If the left button is down, the field is selected from the point of the click to the beginning of the field; if the right button is down, the field is selected from the point of the click to the end of the field.

```
; Site Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)          ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)     ; select from point to end
  endif
endif
endmethod
```

See also

❒ isLeftDown, isMiddleDown, setRightDown

isShiftKeyDown

MouseEvent

Method

Reports whether *Shift* is held down during a MouseEvent.

Syntax

isShiftKeyDown ( ) Logical

Description

Returns True if *Shift* is held down during a MouseEvent; otherwise, it returns False.

Example

The following example is attached to the **mouseUp** method for the *Site Notes* field. When the user presses *Shift* while clicking, the word to the right of the insertion point is selected.

```
; Site Notes::mouseUp
method mouseUp(var eventInfo MouseEvent)
;if Shift is down, select the word to the right
if eventInfo.isShiftKeyDown() then
   self.action(SelectRightWord)
endif
endmethod
```

See also

❒ isControlKeyDown, setShiftKeyDown

setControlKeyDown

MouseEvent

Method

Simulates pressing and holding *Ctrl* during a MouseEvent.

Syntax

setControlKeyDown (const ***yesNo*** Logical)

Description

Adds information about the state of *Ctrl* for a MouseEvent. You must specify Yes or No. Yes means *Ctrl* was pressed and held during a MouseEvent; No means *Ctrl* was not pressed.

Example

The following example creates a MouseEvent and sets *Ctrl* to Yes. The event is then sent to the **mouseUp** built-in method for a field called *lcField.* This method is attached to the **pushButton** method for a button named *sendCtrl.*

```
; sendCtrl::pushButton
method pushButton(var eventInfo Event)
var
  ctrlMsEvent MouseEvent          ; declare the event
endVar

ctrlMsEvent.setControlKeyDown(Yes)   ; set the Control key
lcField.mouseUp(ctrlMsEvent)         ; send the event
endmethod
```

This code is attached to the **mouseUp** method for *lcField.* This method checks whether *Ctrl* is pressed when the mouse is clicked. If so, the value in the field is changed to all lowercase.

```
; lcField::mouseUp
method mouseUp(var eventInfo MouseEvent)
if eventInfo.isControlKeyDown() then    ; check for Control key
  self.Value = lower(self.Value)        ; change to lowercase
endif
endmethod
```

See also

- isControlKeyDown, setShiftKeyDown

setInside

MouseEvent

Method

Sets the mouse to be inside the current object.

Syntax

setInside (const ***trueFalse*** Logical) Logical

Description

Sets the MouseEvent to be inside the current object.

Example

In this example, the **mouseUp** method for *sendAnEvent* uses **setInside** to change the *eventInfo* variable, then sends the event to *buttonOne*.

```
; sendAnEvent::mouseUp
method mouseUp(var eventInfo MouseEvent)
eventInfo.setInside(Yes)
buttonOne.mouseUp(eventInfo)
endmethod
```

See also

❐ getObjectHit, isInside

setLeftDown

MouseEvent

Method

Simulates pressing the left mouse button.

Syntax

setLeftDown (const ***yesNo*** Logical)

Description

Adds information about the state of the left mouse button for a MouseEvent. You must specify Yes or No. Yes means the left button was clicked; No means the left button was not clicked.

Example

This example constructs a MouseEvent with the left button set down. The MouseEvent is then sent to the **mouseMove** method for *Site_Notes*. This code is attached to the **pushButton** method for *sendLeftButton*:

```
; sendLeftButton::pushButton
method pushButton(var eventInfo Event)
var
  leftMoveMouse MouseEvent     ; create the mouse event
  ui            UIObject
endVar
leftMoveMouse.setLeftDown(Yes) ; set Left button to Yes
ui.attach("Site_Notes")
ui.mouseMove(leftMoveMouse)    ; send the event to Site_Notes
endmethod
```

This code is attached to the **mouseMove** method for *Site_Notes*:

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)          ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)     ; select from point to end
  endif
endif
endmethod
```

See also

❐ isLeftDown, setMiddleDown, setRightDown

setMiddleDown

MouseEvent

Method

Simulates pressing the middle mouse button.

Syntax

setMiddleDown (const ***yesNo*** Logical)

Description

Adds information about the state of the middle mouse button for a MouseEvent. You must specify Yes or No. Yes means the middle button was clicked; No means the middle button was not clicked.

Example

This example assumes that a form contains a button called *sendMove* and a field object from the *Sites* table called *Site_Notes*. The **pushButton** method for *sendMove* constructs a MouseEvent with the middle button down, then sends MouseEvent to the *Site_Notes* field object.

```
; sendMove::pushButton
method pushButton(var eventInfo Event)
var
  mo  MouseEvent         ; declare a MouseEvent to send
  ui  UIObject
endVar
ui.attach("Site_Notes")  ; attach to Site_Notes
mo.setMiddleDown(Yes)    ; set middle button down on MouseEvent
ui.mouseMove(mo)         ; dispatch event to mouseMove for Site Notes
endmethod
```

This method is attached to the **mouseMove** method for *Site_Notes*. If the middle button is down for the MouseEvent, the method moves to the beginning of the current word, then selects the entire word.

```
; Site_Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isMiddleDown() then
  self.action(MoveLeftWord)    ; go to the beginning of the word
  self.action(SelectRightWord) ; select the entire word
endif
endmethod
```

See also

- isMiddleDown, setLeftDown, setRightDown

setMousePosition

MouseEvent

Method

Sets the position of the mouse for an event.

Syntax

1. **setMousePosition** (const ***xPosition*** LongInt, const ***yPosition*** LongInt)
2. **setMousePosition** (const ***p*** Point)

Description

Adds information about the position of the mouse for a MouseEvent. *xPosition* and *yPosition* specify the x- and y-coordinates in twips, relative to the upper left corner of the target object's container.

Example

The following example creates a new event, sets the mouse position to 500 twips to the right and below the current mouse position, and sends the event to the **mouseRightUp** method for the same object. This code is attached to the **mouseUp** method for an object called *boxOne*:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  rightEvent MouseEvent
endVar
; set the new position to current plus 500, 500
rightEvent.setMousePosition(eventInfo.x() + 500,
                            eventInfo.y() + 500)
mouseRightUp(rightEvent)            ; send off the new event
endmethod
```

See also

- getMousePosition

setRightDown

MouseEvent

Method

Simulates pressing the right mouse button.

Syntax

setRightDown (const ***yesNo*** Logical)

Description

Adds information about the state of the right mouse button for a MouseEvent. You must specify Yes or No. Yes means the right button was clicked; No means the right button was not clicked.

Example

This example constructs a MouseEvent with the right button set down. The MouseEvent is then sent to the **mouseMove** method for *Site_Notes*. This code is attached to the **pushButton** method for *sendRightButton*:

```
; sendRightButton::pushButton
method pushButton(var eventInfo Event)
var
  rightMoveMouse MouseEvent      ; declare the event
  ui             UIObject
endVar
rightMoveMouse.setRightDown(Yes) ; set right button down
ui.attach("Site_Notes")
ui.mouseMove(rightMoveMouse)     ; send the event to Site_Notes
endmethod
```

This code is attached to the **mouseMove** method for *Site_Notes*:

```
; Site Notes::mouseMove
method mouseMove(var eventInfo MouseEvent)
if eventInfo.isLeftDown() then
  self.action(SelectTop)          ; select from point to beginning
else
  if eventInfo.isRightDown() then
    self.action(SelectBottom)     ; select from point to end
  endif
endif
endmethod
```

See also

- isRightDown, setLeftDown, setMiddleDown

setShiftKeyDown

MouseEvent

Method

Simulates pressing and holding *Shift.*

Syntax

setShiftKeyDown (const ***yesNo*** Logical)

Description

Adds information about the state of *Shift* for a MouseEvent. You must specify Yes or No. Yes means *Shift* was pressed and held; No means *Shift* wasn't pressed.

Example

The following example creates a MouseEvent and sets *Shift* to Yes. The event is then sent to the **mouseUp** built-in method for a field called *ucField.* This method is attached to the **pushButton** method for a button named *sendShift.*

```
; sendShift::pushButton
method pushButton(var eventInfo Event)
var
  shiftMsEvent MouseEvent            ; declare the event
endVar

shiftMsEvent.setShiftKeyDown(Yes)    ; set the Shift key
ucField.mouseUp(shiftMsEvent)        ; send the event

endmethod
```

This code is attached to the **mouseUp** method for *ucField.* This method checks whether *Shift* is pressed when the mouse is clicked. If so, the value in the field is changed to all uppercase.

```
; ucField::mouseUp
method mouseUp(var eventInfo MouseEvent)
if eventInfo.isShiftKeyDown() then    ; check for Shift key
  self.Value = upper(self.Value)      ; change to uppercase
endif
endmethod
```

See also

- isShiftKeyDown, setControlKeyDown

setX

MouseEvent

Method

Specifies the horizontal coordinate of the pointer position. Coordinates must be specified relative to the upper left corner of the current object.

Syntax

setX (const ***xPosition*** LongInt)

Description

Sets the horizontal coordinate (in twips) of the pointer position to *xPosition*.

Example

This example involves two methods for the same object, *boxOne*. The **mouseUp** method creates a MouseEvent, setting the coordinates to 500 twips greater than the point of the click. The **mouseUp** method then sends the event to **mouseRightUp**. The **mouseRightUp** method gets the coordinates, converts them so they are placed properly on *boxOne*, and draws a box at the point indicated by the MouseEvent. If the MouseEvent is the result of a user interaction (**isFromUI** returns True), the new box is painted Red. If the MouseEvent is not the result of a user interaction, as when the event is passed from the **mouseUp** method, the new box is painted Green. The **mouseUp** method for *boxOne* is:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  rightEvent MouseEvent
endVar
; set the new position to current plus 500, 500
rightEvent.setX(eventInfo.x() + 500)
rightEvent.setY(eventInfo.y() + 500)
mouseRightUp(rightEvent)            ; send off the new event
endmethod
```

This code is attached to the **mouseRightUp** method for *boxOne*:

```
; boxOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  ui     UIObject     ; to create object at point of click
  msPt   Point        ; the x, y point of click
endVar

; get the x and y coordinates of the click
msPt = Point(eventInfo.x(), eventInfo.y())

; convert the point from the page to the box
self.convertPointWithRespectTo(pageOne, msPt, msPt)

; create the box, color it, and set it to visible
ui.create(boxTool, msPt.x(), msPt.y(), 200, 200)
ui.Visible = True
if eventInfo.isFromUI() then
  ui.Color = Red          ; native event
else
```

MouseEvent

```
  ui.Color = Green        ; mouse event passed from mouseUp
endif
endmethod
```

See also

- setY, x, y
- convertPointWithRespectTo in the UIObject type

setY

MouseEvent

Method

Specifies the vertical coordinate of the pointer position. Coordinates must be specified relative to the upper left corner of the current object.

Syntax

setY (const ***yPosition*** LongInt)

Description

Sets the vertical coordinate (in twips) of the mouse pointer position to *yPosition*.

Example

See the example for setX.

See also

- setX, x, y
- convertPointWithRespectTo in the UIObject type

x

MouseEvent

Method

Returns the horizontal coordinate of the pointer position.

Syntax

x ( ) LongInt

Description

Returns (in twips) the horizontal coordinate of the mouse pointer position.

Example

See the example for setX.

See also

- setX, y
- convertPointWithRespectTo in the UIObject type

y

MouseEvent

Method

Returns the vertical coordinate of the mouse pointer position.

Syntax

y ( ) LongInt

Description

Returns (in twips) the vertical coordinate of the pointer position.

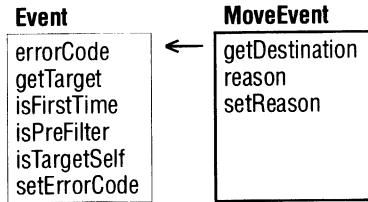
Example

See the example for setX.

See also

- setY, x
- convertPointWithRespectTo in the UIObject type

MoveEvent



Methods for the MoveEvent type enable you to get and set information about the events that occur as you navigate from one object to another in a form. The MoveEvent type includes several methods defined for the Event type. Refer to the "Event" section for more information.

The following built-in methods are triggered by MoveEvents: **arrive**, **canArrive**, **canDepart**, and **depart**. These methods, along with the rest of the built-in methods, are discussed in Chapter 2. For information about the event model, see the *ObjectPAL Developer's Guide*.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide*.

getDestination

MoveEvent

Method

Reports which object is the destination of a MoveEvent.

Syntax

getDestination (var ***dest*** UIObject)

Description

Returns in *dest* the object the user is trying to move to in a form.

Example

In this example, assume that the form contains a multi-record object bound to the *Orders* table. The **canDepart** method for the form is called whenever the user attempts to move off a field or other object in the form. The **canDepart** method shown in this example uses **getDestination** to find the intended destination of the MoveEvent. This method uses **getTarget** to find the source of the move and compare it with the destination.

If the containers of the two objects are the same, such as when the user is moving from one field to the next in a multi-record object, the method displays a dialog box asking for confirmation. When the user responds, the move occurs and the field the user moved from is set to yellow. If the target's container and the destination's container are different, such as when the user is attempting to leave the form altogether, the method doesn't display the dialog box. The following code is attached to the **canDepart** method for a form:

```
; thisForm::canDepart
method canDepart(var eventInfo MoveEvent)
var
  destObj UIObject
  targObj UIObject
  doMove  String
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    eventInfo.getTarget(targObj)
    eventInfo.getDestination(destObj)
    if targObj.ContainerName = destObj.ContainerName then
      ; handle only field-to-field moves within the MRO
      doMove = msgQuestion("Move?", "Move to " + destObj.name + " ?")
      if doMove = "No" then
        eventInfo.setErrorCode(CanNotDepart)
      else
        targObj.Color = Yellow    ; leave a trail of yellow fields
      endIf
    endif
  else
    ;code here executes just for form itself

endif
endmethod
```

See also

- reason
- getTarget in the Event type

---

reason

MoveEvent

Method

Reports why a move occurred.

Syntax

reason ( ) SmallInt

Description

Returns an integer value to report why a MoveEvent occurred. MoveEvent Reason constants occur when a built-in **arrive**, **depart**, **canArrive**, or **canDepart** method is called. ObjectPAL provides the following constants for testing the value returned by **reason** (listed in the Constants dialog box under MoveReasons):

- PalMove means the move was generated by an ObjectPAL statement.
- RefreshMove means the move was generated by table values being refreshed across a network.
- ShutDownMove means the move was generated as the form closed.
- StartupMove means the move was generated as the form opened.
- UserMove means the move was caused by the user interacting with the form.

Example

In this example, assume a form contains a field object named *fieldOne*, and a button named *moveToFieldOne*, as well as at least one other field object. A movement away from *fieldOne* is treated as normal; however, to return to *fieldOne*, the user must press the *moveToFieldOne* button. The **canArrive** method for *fieldOne* checks the reason for the move, and blocks field arrival if the reason is not UserMove. The following code is attached to the **canArrive** method for *fieldOne*:

```
; fieldOne::canArrive
method canArrive(var eventInfo MoveEvent)
; don't allow user to move to field by tabbing or clicking
if eventInfo.reason() = UserMove then
  eventInfo.setErrorCode(CanNotArrive)
  beep()
  message("Press the Move to Field One button to move to Field One.")
endif
endmethod
```

The following code is attached to the **pushButton** method for *moveToFieldOne*:

```
; moveToFieldOne::pushButton
method pushButton(var eventInfo Event)
; move to fieldOne if it does not currently have focus
if fieldOne.Focus = False then
  fieldOne.moveTo()
else
  fieldTwo.moveTo()
endif
endmethod
```

See also

- setReason

setReason

MoveEvent

Method

Specifies a Reason for a MoveEvent.

Syntax

setReason (const ***reasonId*** SmallInt)

Description

Specifies a Reason for generating a MoveEvent. This method takes one of the following constants as an argument (listed in the Constants dialog box under MoveReasons):

- PalMove means the move was generated by an ObjectPAL statement.
- RefreshMove means the move was generated by table values being refreshed across a network.
- ShutDownMove means the move was generated as the form closed.
- StartupMove means the move was generated as the form opened.
- UserMove means the move was caused by the user interacting with the form.

Example

In this example, the **canArrive** method for *fieldOne* blocks field arrival if the Reason for the move is UserMove. To temporarily circumvent this restriction, the form's **canArrive** method changes the Reason for UserMove events to PalMove events. This code is attached to the **canArrive** method for *fieldOne*:

```
; fieldOne::canArrive
method canArrive(var eventInfo MoveEvent)
; don't allow user to move to field by tabbing or clicking
if eventInfo.reason() = UserMove then
  eventInfo.setErrorCode(CanNotArrive)
  beep()
  message("Press the Move to Field One button to move to Field One.")
endif
endmethod
```

This code is attached to the **canArrive** method for the form:

```
; thisForm::canArrive
method canArrive(var eventInfo MoveEvent)
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
    ; change events with a reason of UserMove to PalMove
    if eventInfo.reason() = UserMove then
      eventInfo.setReason(PalMove)
    endif
  else
    ;code here executes just for form itself

endif
endmethod
```

See also

- reason

Number

AnyType		Number	
blank	←	abs	mod
dataType		acos	number
isAssigned		asin	numVal
isBlank		atan	pmt
isFixedType		atan2	pow
unAssign		ceil	pow10
view		cos	pv
		cosh	rand
		exp	round
		floor	sin
		fraction	sinh
		fv	sqrt
		ln	tan
		log	tanh
		max	truncate
		min	

Number variables represent floating-point values consisting of a significand (fractional portion, for example, 3.224) multiplied by a power of 10. The significand can contain up to 18 significant digits, and the power of 10 can range from $\pm 3.4 * 10^{-4930}$ to $\pm 1.1 * 10^{4930}$. An attempt to assign a value outside of this range to a Number variable causes an error.

Note The Number type includes methods defined for the AnyType type. Also, run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a Number. For example, the following code works, even though **sin** does not appear in the list of methods for the LongInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note ObjectPAL supports an alternate syntax:

methodName (*objVar*, *argument* [,*argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

This statement uses the alternate syntax:

```
sin(theNum)
```

It's best to use standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

Note The display formats of numeric method may vary depending on the Windows number format of the user's system, but ObjectPAL's internal representation is always the same.

abs

Beginner

Number

Method Returns the absolute value of a number.

Syntax **abs ( )** Number

Description Removes the sign from a numeric value.

Example For the following example, assume that a form contains three field objects: *forecastAmt*, *actualAmt*, and *diffPercent*. The **newValue** method for *actualAmt* finds the difference between *forecastAmt* and *actualAmt*, then calculates how far off the forecast was. Depending on the values in *forecastAmt* and *actualAmt*, the number by which they differ can be positive or negative. To find the percentage of error, **abs** is used to get the absolute value of the number, which is then multiplied by 100 to get a percentage. This code is attached to the **newValue** method for *actualAmt*:

```
; actualAmt::newValue
method newValue(var eventInfo Event)
var
  difference  Number
endVar
; don't execute if newValue is being called at startup, or
; if one of the fields involved is blank
if eventInfo.reason() <> StartupValue then
  if NOT self.isBlank() AND
     NOT forecastAmt.isBlank() then
    ; find out how much forecast differs from actual
    difference = (forecastAmt - Number(self.Value)) / forecastAmt
    diffPercent = difference.abs() * 100  ; get the variation as
                                          ; an absolute value
  else
    msgStop("Error", "The forecastAmt field can't be blank.")
  endif
endif
endmethod
```

See also

- number

Number

acos

Beginner

Number

Method

Returns the 2-quadrant arc cosine of a number.

Syntax

acos ( ) Number

Description

Given a number between -1 and 1, **acos** returns a numeric value between 0 and pi, expressed in radians. **acos** is called the 2-quadrant arc cosine because it returns values within quadrants 1 and 4 (that is, between –pi/2 and pi/2). **acos** is the inverse of **cos** (if **acos**(*x*) = *y*, then **cos**(*y*) = *x*).

Example

The **pushButton** method for the *findArcCos* button calculates and displays the arc cosine of a 30-degree angle.

```
; findArcCos::pushButton
method pushButton(var eventInfo Event)
var
  thirtyDegrees Number
endVar
thirtyDegrees = PI / 3.0
msgInfo("The arc cosine of 30 degrees", thirtyDegrees.acos())   ; displays 1.02
endmethod
```

See also

- acos, asin, atan

asin

Beginner

Number

Method

Returns the 2-quadrant arc sine of a number.

Syntax

asin ( ) Number

Description

Given a number between –1 and 1, **asin** returns a numeric value between –pi/2 and pi/2, expressed in radians.

Example

For this example, the **pushButton** method for the *findASin* button displays the arc sine of a number.

```
; findASin::pushButton
var
  x Number
endvar
x = .5
msgInfo("arc sine of .5", x.asin())   ; displays .52
endmethod
```

See also

- acos, atan, cos, sin

atan

Beginner

Number

Method

Returns the 2-quadrant arc tangent of a number.

Syntax

atan ( ) Number

Description

Given a tangent in radians, **atan** returns the angle in radians. **atan** is called the 2-quadrant arc tangent because it returns values within quadrants 1 and 4 (that is, between –pi/2 and pi/2). **atan** is the inverse of **tan** (if atan(x) = y, then tan(y) = x).

Example

In this example, the **pushButton** method for *getAtan* calculates the 2-quadrant arc tangent of x and y, then displays the result.

```
; getAtan::pushButton
method pushButton(var eventInfo Event)
var
  x   Number
  checkPi, fortyFiveDegrees Number
endvar
x = 1
fortyFiveDegrees = x.atan()
msgInfo("45 degrees in radians: ", fortyFiveDegrees)  ; 0.79
checkPi = fortyFiveDegrees * 4       ; pi radians = 180 degrees
msgInfo("pi: ", format("w12.10", checkPi))
endmethod
```

See also

- atan2, cos, sin, tan, tanh

atan2

Beginner

Number

Method

Returns the 4-quadrant arc tangent of a number.

Syntax

atan2 (const **x** Number) Number

Description

Given a tangent in radians, **atan2** returns an angle in radians. **atan2** is called the 4-quadrant arc tangent because it returns values in all four quadrants.

Example

For the following example, assume that a form contains a button named *getAtan2*. The **pushButton** method for *getAtan2* calculates the 4-quadrant arc tangent of x and y, then displays the results:

```
; getAtan2::pushButton
method pushButton(var eventInfo Event)
var
  x,
  y,
  checkpi,
  fortyFiveDegrees Number
endvar
x = 1                          ; The angle whose tangent is 1 / 1
y = 1                          ; is a 45 degree angle
fortyFiveDegrees = x.atan2(y)
msgInfo("45 degrees in radians: ", fortyFiveDegrees)  ; 0.79
checkpi = fortyFiveDegrees * 4.0      ; pi radians = 180 degrees
msgInfo("pi: ", format("w12.10", checkpi))
endmethod
```

See also

- cos, sin, tan, tanh

---

ceil

Number

Beginner

Method

Rounds a numeric expression up to the nearest whole number.

Syntax

ceil ( ) Number

Description

Rounds a numeric expression up (toward positive infinity) to the nearest whole number.

Example

In this example, the **pushButton** method for a button named *ceilVsRound* calculates the ceiling value of a number, then shows the rounded value of the same number:

```
; ceilVsRound::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 3.1
msgInfo("The ceil of " + string(x) + " is", ceil(x))     ; displays 4.0
msgInfo("The round of " + string(x) + " is", x.round(0)) ; displays 3
endmethod
```

See also

- floor

---

cos

Number

Beginner

Method

Returns the cosine of an angle.

Syntax

cos ( ) Number

Description

Returns a value between -1 and 1 for the cosine of a value or expression representing the size of the angle in radians.

Example

In this example, the **pushButton** method for the *findCosine* button calculates and displays the cosine of a 60-degree angle:

```
; findCosine::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The cosine of 60 degrees", sixtyDegrees.cos()) ; displays 0.50
endmethod
```

See also

- cosh, sin, tan

---

cosh

Beginner

Number

Method

Returns the hyperbolic cosine of an angle.

Syntax

cosh ( ) Number

Description

Returns the hyperbolic cosine of a value or expression representing the size of the angle in radians. The formula used is

cosh(angle) = (exp(angle) + exp(-angle)) / 2

Example

The **pushButton** method for the *findCosineH* button calculates and displays the *h* cosine of 60 degrees.

```
; findCosineH::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The h cosine of " + format("W8.6", sixtyDegrees) + " radians",
         format("W14.12", sixtyDegrees.cosh()))
; displays 1.600286857702
endmethod
```

See also

- cos, sin, tan

exp

Number

Beginner

Method

Returns the exponential (base *e*) of a number.

Syntax

exp ( ) Number

Description

Computes e^x, where *e* is the constant 2.7182845905. The inverse method is **ln**.

Example

In this example, the **pushButton** method for a button named *getExponent* button calculates and displays the base *e* of 1:

```
; getExponent::pushButton
method pushButton(var eventInfo Event)
msgInfo("The exp of 1.0", format("W14.12", exp(1.0)))
; exp(1) formatted to display full precision
endmethod
```

See also

- ln, log

floor

Number

Beginner

Method

Rounds a numeric expression down to the nearest whole number.

Syntax

floor ( ) Number

Description

Rounds a numeric expression down (toward negative infinity) to the nearest whole number.

Example

In the following example, the **pushButton** method for a button named *floorVsRound* uses **floor** to round *x* down to the nearest integer. By comparison, for the same number, **round** results in a higher number.

```
; floorVsRound::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 3.9
msgInfo("The floor of " + string(x) + " is", floor(x))   ; displays 3.0
msgInfo("The round of " + string(x) + " is", x.round(0)) ; displays 4.0
endmethod
```

See also

- ceil

fraction

Beginner

Number

Method

Returns the fractional part of a number.

Syntax

fraction ( ) Number

Description

Returns the fractional part of a number, the part to the right of the decimal.

Example

In this example, the **pushButton** method for *fractButton* displays the fraction portion of a numeric variable:

```
; fractButton::pushButton
method pushButton(var eventInfo Event)
var
  myNum Number
endVar
myNum = 12.23
msgInfo("Fractional part of " + string(myNum),
        myNum.fraction())                ; displays .23
endmethod
```

See also

- mod

fv

Beginner

Number

Method

Returns the future value of a series of equal payments.

Syntax

fv (const ***interestRate*** Number, ***periods*** Number) Number

Description

Returns the future value of a series of equal payment *periods*, invested at interest rate *interestRate*. *interestRate* is expressed as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the deposit period; that is, if the deposits are monthly, the interest rate should be monthly too.

The formula used is

FV = payment(pow(1 + *rate*, *periods*) - 1) / *rate*

fv is sometimes called the future or compound value of an annuity because you can use it to calculate the amount accumulated in an annuity fund when making regular, equal payments over time.

Number

Example

This example calculates how much a 14.5% Individual Retirement Account would be worth if $166.67 were deposited every month for 30 years.

```
; findFutureVal::pushButton
method pushButton(var eventInfo Event)
var
  depositAmt,
  intRate,
  numPayments,
  iraValue       Number
endVar
intRate = .145 / 12       ; convert yearly interest to monthly interest
numPayments = 360         ; monthly payments for 30 years
depositAmt = 166.67       ; monthly deposit amount ($2000 a year)
iraValue = depositAmt.fv(intRate, numPayments)
msgInfo("IRA Value", "Depositing " + string(depositAmt) +
        " a month for " + string(numPayments/12) + " years at " +
        string(intRate * 12 * 100) + "% yields " + string(iraValue) +
        ". You'll be old but you'll be rich!")
; displays "Depositing 166.67 a month for 30 years
;          at 14.50% yields 1,027,394.23 ..."
endmethod
```

See also

- pmt, pv

ln

Number

Beginner

Method

Returns the natural logarithm of a numeric expression.

Syntax

ln ( ) Number

Description

Calculates the natural logarithm to the base *e* of a positive value. The constant *e* is approximated by the value 2.7182845905. If the value is 0 or negative, **ln** fails.

The inverse method is **exp**. Use **log** to compute base 10 logarithms.

Example

In the following example, the **pushButton** method for the *findNatLog* button calculates and displays the natural logarithm of several numbers:

```
; findNatLog::pushButton
method pushButton(var eventInfo Event)
var
  x  Number
endVar
x = 2.71828
msgInfo("Natural log of " + format("W10.6", x), ln(x)) ; displays 1.00
x = 7.3891
msgInfo("Natural log of " + format("W10.6", x), ln(x)) ; displays 2.00
x = 20.0855
```

```
msgInfo("Natural log of " + format("W10.6", x), ln(x)) ; displays 3.00
endmethod
```

See also

- exp, log

log

Beginner

Number

Method

Returns the base 10 logarithm of a numeric expression.

Syntax

log ( ) Number

Description

Returns the base 10 logarithm of a value or numeric expression. If the value is 0 or negative, **log** fails.

Use **ln** to compute natural logarithms.

Example

```
; findLog::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = 10
msgInfo("The logarithm of " + string(x), log(x)) ; displays 1.00
x = 100
msgInfo("The logarithm of " + string(x), log(x)) ; displays 2.00
x = 1000
msgInfo("The logarithm of " + string(x), log(x)) ; displays 3.00
endmethod
```

See also

- exp, ln

max

Beginner

Number

Procedure

Returns the larger of two numbers.

Syntax

max (const ***x1*** AnyType, const ***x2*** AnyType) AnyType

Description

Returns the larger of the two values *x1* and *x2*.

Example

In the following example, you want to find the medical deduction you're allowed for tax purposes. The **pushButton** method for *findMedDeduct* finds the maximum of 7.5% of *AGI* or *medExpense*, then deducts 7.5% of *AGI* from the result. Finding the maximum number first ensures that the calculation won't return a negative number.

```
; findMedDeduct
method pushButton(var eventInfo Event)
var
  medExpense,
  AGI        Number
endVar
AGI = 32000.45
medExpense = 4035.24
msgInfo("Allowed Medical Deduction",
        max(medExpense, AGI * .075) - (AGI * .075))  ; displays 1,635.21
; assumes that you can deduct only that part of your medical and dental
; expenses greater than 7.5% of Adjusted Gross Income
endmethod
```

See also

- min

---

min

Number

Beginner

Procedure

Returns the smaller of two numbers.

Syntax

min (const ***x1*** AnyType, const ***x2*** AnyType) AnyType

Description

Returns the smaller of the two values, *x1* and *x2*.

Example

In this example, you want to calculate the maximum amount of tax-deductible charitable contributions, and no more than 30% of adjusted gross income can be deducted. The **pushButton** method for the *findCharityDeduct* button finds and displays the minimum of 30% of *AGI* and *charity*.

```
; findCharityDeduct::pushButton
method pushButton(var eventInfo Event)
var
  charity,
  AGI       Number
endVar
AGI = 32000.45     ; Adjusted Gross Income
charity = 12000    ; charitable contributions for the year
msgInfo("Allowed Charity Deduction", min(charity, AGI * .30))
; displays 9,600.13
; assumes charitable contributions up to 30% of AGI
; are allowed as deductions
endmethod
```

See also

- max

mod

Number

Beginner

Method

Returns the remainder when one number is divided by another.

Syntax

mod (const ***modulo*** Number) Number

Description

Returns the remainder (or modulus) when a number is divided by the value of *modulo*. If *modulo* is 0, **mod** returns 0.

Example

In the following example, the **pushButton** method for the *showRemainder* button calculates and displays the modulus for a series of division operations:

```
; showRemainder::pushButton
method pushButton(var eventInfo Event)
var
  x, m Number
endVar
x = 8
msgInfo("The remainder of " + string(x) + "/" + "3",
        x.mod(3))                       ; displays 2
msgInfo("The remainder of " + string(x) + "/" + "12",
        x.mod(12))                      ; displays 3
x = -2
msgInfo("The remainder of " + string(x) + "/" + "10",
        x.mod(10))                      ; displays -2
x = -10
msgInfo("The remainder of " + string(x) + "/" + "-100",
        x.mod(-100))                    ; displays -10
endmethod
```

See also

- fraction

number

Number

Beginner

Procedure

Casts a value as a Number.

Syntax

number (const ***value*** AnyType) Number

Description

Casts (converts) *value* to a Number. *value* must be in the form of a valid number that can be entered in a field. **number** is used to cast a non-numeric type to a Number when a numeric operand is required in an expression, or a numeric argument is required in a procedure or method. **number** behaves the same as **numVal**.

Example

In the following example, a variable *x* is declared as a String, then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it, then displays the result:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
  x  String
endVar
x = "1,123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", number(x) * 2) ; displays 2,247.08
endmethod
```

See also

- numVal

numVal

Number

Procedure

Casts a value as a Number.

Syntax

numVal (const ***value*** AnyType) Number

Description

Casts (converts) *value* to a Number. *value* must be in the form of a valid number that can be entered in a field. **numVal** is most often used to cast a non-numeric type to a Number when a numeric operand is required in an expression, or a numeric argument is required in a procedure or method. **numVal** behaves the same as **number**.

Example

In the following example, a variable *x* is declared as a String, then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it, then displays the result:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
  x  String
endVar
x = "1,123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", numVal(x) * 2) ; displays 2,247.08
endmethod
```

See also

- number

pmt

Beginner

Number

Method

Returns the periodic payment required to pay off a loan.

Syntax

pmt (const ***interestRate*** Number, const ***periods*** Number) Number

Description

Returns the constant, regular payment required to amortize (pay off) a loan. The formula used is:

$$PMT = \frac{p * i}{1 - (1 + i)^{-t}}$$

where p = principal amount, i = effective interest rate per period, and t = term of the loan (number of payment periods).

Payments are considered due at the end of each period.

pmt works for amortization-type loans (for example, conventional home mortgages), in which part of the payment consists of interest on the remaining principal, and the remainder pays off part of the principal of the loan. **pmt** does not work for consumer-type loans, such as repayments of credit accounts or automobile loans.

The interest rate used in **pmt** is expressed in *interestRate* as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the payment periods; that is, if the payments are monthly, the interest rate should also be monthly. Since the interest rate usually quoted for amortization loans (mortgages) is annual, divide it by 12 for monthly payments, by 4 for quarterly payments, and so on.

Start with the nominal annual interest rate quoted, not the accompanying annual percentage rate (APR).

Example

In the following example, the **pushButton** method for the *findPayment* button calculates the monthly payment for a 24-month loan of $1,000 at 12%:

```
; findPayment::pushButton
method pushButton(var eventInfo Event)
var
  monthlyPayment,
  loanAmt,
  intRate,
  numPayments Number
endVar
loanAmt = 1000          ; borrow $1000
intRate = .12 / 12      ; 12 percent annual interest
numPayments = 24        ; 1 payment per month for 2 years
monthlyPayment = loanAmt.pmt(intRate, numPayments)
msgInfo("Monthly payment", "The monthly payment for a loan of " +
```

Number

```
        String(loanAmt) + " at " + String(intRate * 12 * 100) +
        "% interest for " + String(SmallInt(numPayments)) +
        " months is " + String(monthlyPayment))    ; payment is $47.07
endmethod
```

See also

- fv, pv

pow

Beginner

Number

Method

Raises a number to a power.

Syntax

pow (const ***exponent*** Number) Number

Description

Returns the value of a number raised to the power specified in *exponent*. If the result is larger than 10^{308} or smaller than 10^{-307}, you'll get an error.

Example

In the following example, the **pushButton** method for the *raiseTwo* button calculates $root^{expn}$ and displays the result:

```
; raiseTwo::pushButton
method pushButton(var eventInfo Event)
var
  root,
  expn    Number
endVar
root = 2
expn = 8
msgInfo(string(root) + " raised to the power of " +
        string(expn), root.pow(expn)) ; displays 256
endmethod
```

See also

- ln, log, pow10

pow10

Beginner

Number

Method

Calculates 10 to a specified power.

Syntax

pow10 ( ) Number

Description

Returns the value of 10 raised to a specified power.

Example

In this example, the **pushButton** method for the *raiseTen* button calculates 10^{expn} and displays the result:

```
; raiseTen::pushButton
method pushButton(var eventInfo Event)
var
  expn,
  result Number
endVar
expn = 9
result = expn.pow10()
msgInfo("Ten raised by a power of " + String(expn),
        format("EC", result))               ; displays 1,000,000,000
endmethod
```

See also

- ln, log, pow

---

pv

Beginner

Number

Method

Returns the present value of a series of equal payments.

Syntax

pv (const ***interestRate*** Number, const ***periods*** Number) Number

Description

Calculates the present value of equal, regular payments on a loan (or withdrawals from an investment) at a rate specified in *interestRate* for a number of periods specified in *periods*. The payments reduce the principal, but the remaining balance continues to generate and compound interest.

The formula used is

$$PV = payment * \left(\frac{1-(1+rate)^{-n}}{rate} \right)$$

where n is the number of periods.

The interest rate used in **pv** is expressed as a decimal number (like .12), not as a percentage (12%). Make sure the rate period matches the payment period; that is, if the payments are monthly, the interest rate should also be monthly. You can use **pv** to calculate how large a mortgage you can afford. (Use **pmt** to work in reverse and find the monthly payment needed to amortize a given amount.) You can also use **pv** to calculate the amount you'll need to purchase an annuity that will make regular, equal payments to you over time. For this reason, **pv** is sometimes called the present value of an annuity.

Example

Suppose you can afford to pay $1,200 per month and can get a 30-year mortgage at a fixed annual rate of 9% (0.75% monthly). The **pushButton** method for *findPV* calculates and displays the loan amount for which you can qualify:

```
; findPV::pushButton
method pushButton(var eventInfo Event)
var
  payAmt,
  intRate,
  term,
  mortgage    Number
endVar
payAmt   = 1200
intRate  = .09 / 12             ; monthly interest for 9% a year
term     = 360                  ; 30 years (expressed in months)
mortgage = payAmt.pv(intRate, term)
msgInfo("Maximum Mortgage", "If you can pay " + string(payAmt) +
        " a month for " + string(term /12) + " years at " +
        string(intRate * 12 * 100) + "% you can qualify for " +
        format("E$C", mortgage))     ; displays $149,138
endmethod
```

Suppose when you retire you would like to withdraw $2,500 each month for 30 years from an annuity account that accumulates 7.5% annual interest. This **pushButton** method for the *findAnnuity* button calculates how much you'll need in the account:

```
; findAnnuity::pushButton
method pushButton(var eventInfo Event)
var
  monthlyAmt,
  term,
  intRate,
  investment  Number
endVar

monthlyAmt = 2500.00  ; monthly amount you want annuity to pay
term = 360            ; 30 years, converted to 360 months
intRate = .075/12     ; 7.5% a year, converted to monthly rate
investment = monthlyAmt.pv(intRate, term)  ; what you need to start with
msgInfo("Annuity Required", "For an annuity to return $" +
        string(monthlyAmt) + " a month at " +
        format("W4.2", intRate * 12 * 100) + "% for " +
        string(SmallInt(term / 12)) +
        " years, the original amount must be " +
        string(investment))             ; displays 357,544.07
endmethod
```

See also

- fv, pmt

rand

Number

Beginner

Procedure

Generates a random value ranging from 0 to 1.

Syntax

rand ( ) Number

Description

Generates a random value ranging from 0 to 1.

Example

In the following example, the **pushButton** method for the *getRand* button calculates and displays a random number *x* between 1 (*minNum*) and 10 (*maxNum*).

```
; getRand::pushButton
method pushButton(var eventInfo Event)
var
  x,
  minNum,
  maxNum  SmallInt
endVar
minNum = 1
maxNum = 10
; get a random integer between minNum and maxNum
x = SmallInt(rand() * (maxNum - minNum + 1) + minNum)
msgInfo("A number between " + String(minNum) + " and " +
        String(maxNum),  x)
endmethod
```

See also

- truncate

round

Beginner

Number

Method

Rounds a number to a specified number of decimal places.

Syntax

round (const ***places*** SmallInt) Number

Description

Returns a number rounded to the number of decimal places specified in *places*.

Example

In the following example, the **pushButton** method for the *showRound* button rounds a number to 4 decimal places and displays the result, then rounds and displays a number to the nearest 1000.

```
; showRound::pushButton
method pushButton(var eventInfo Event)
var
  roundMe Number
endVar
roundMe = 1.2356838
msgInfo(format("W9.7",roundMe) + " rounded to 4 decimal places",
         format("W6.4", roundMe.round(4))) ; displays 1.2357
roundMe = 678394
msgInfo(string(roundMe) + " rounded to -3 decimal places",
         roundMe.round(-3))               ; displays 678,000
endmethod
```

See also

- ceil, floor, truncate

sin

Number

Beginner

Method

Returns the sine of an angle.

Syntax

sin ( ) Number

Description

Returns a numeric value between -1 and 1 for the sine of a value representing the size of the angle in radians.

Example

The **pushButton** method for the *findSin* button finds the sine of a 45-degree angle:

```
; findSin::pushButton
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees  = PI / 4.0
msgInfo("The sine of 45 degrees",
        format("W14.12", fortyFiveDegrees.sin()))
; displays 0.707106781187
endMethod
```

See also

- asin, cos, tan

sinh

Number

Beginner

Method

Returns the hyperbolic sine of an angle.

Syntax

sinh ( ) Number

Description

Returns the hyperbolic sine of a value representing the size of the angle in radians. The formula used is

sinh (angle) = (exp(angle) - exp(-angle)) / 2

Example

In this example, the **pushButton** method for the *getHSine* button finds the hyperbolic sine of a 45-degree angle:

```
; getHSine
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees = PI / 4.0
msgInfo("The hyperbolic sine of 45 degrees",
       format("w14.12", fortyFiveDegrees.sinh()))
```

```
; displays 0.868670961486
endmethod
```

See also

❒ cos, sin, tan

sqrt

Beginner

Number

Method

Returns the square root of a number.

Syntax

sqrt ( ) Number

Description

Returns the square root of a positive value or numeric expression.

Example

In this example, the **pushButton** method for the *getSqrt* button assigns the value from *fieldOne* (an unbound field object) to *x*, checks to see if *x* is negative, and, if not, calculates and displays the square root of *x*:

```
; getSqrt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = fieldOne
if x < 0 then
  msgStop("Sorry",
          "Can't take the square root of a negative number.")
else
  msgInfo("The square root of " + string(x),
           format("w14.6", sqrt(x))) ; displays result
endif
endmethod
```

See also

❒ exp

tan

Beginner

Number

Method

Returns the tangent of an angle.

Syntax

tan ( ) Number

Description

Returns the tangent of a value or numeric expression representing the size of the angle in radians. **tan** diverges at –pi/2, pi/2, and every ±pi radians from those values.

Example

In this example, the **pushButton** method for the *getTan* button calculates the tangent of a 45-degree angle and displays the result:

```
; getTan::pushButton
method pushButton(var eventInfo Event)
var
  fortyFiveDegrees Number
endVar
fortyFiveDegrees = PI / 4.0
msgInfo("Tangent of 45 degrees", fortyFiveDegrees.tan())   ; displays 1.00
endmethod
```

See also

- atan, atan2, cos, sin

tanh

Number

Beginner

Method

Returns the hyperbolic tangent of an angle.

Syntax

tanh ( ) Number

Description

Returns the hyperbolic tangent of a value or numeric expression representing the size of the angle in radians. The formula is

tanh(angle) = sinh(angle) / cosh(angle)

Example

In this example, the **pushButton** method for a button named *getHTan* calculates the hyperbolic tangent of a 60-degree angle and displays the result:

```
; getHTan::pushButton
method pushButton(var eventInfo Event)
var
  sixtyDegrees Number
endVar
sixtyDegrees = PI / 3.0
msgInfo("The hyperbolic tangent of 60 degrees",
        format("W14.12", sixtyDegrees.tanh()))
; displays .780714435359
endmethod
```

See also

- atan, cos, sin

truncate

Number

Beginner

Method

Truncates a number to a specified number of decimal places.

Syntax

truncate (const ***places*** SmallInt) Number

Description

Returns a number truncated towards 0 to the number of decimal places in *places*. It does not round the value.

Example

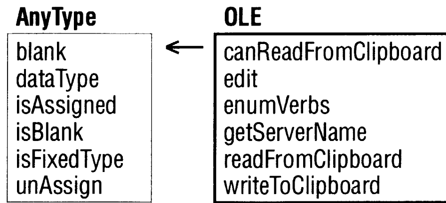
In this example, the **pushButton** method for the *chopAValue* button assigns the value from *fieldOne* (an unbound field object) to *x*, truncates *x* to 3 decimal places, and displays the truncated result:

```
; chopAValue::pushButton
method pushButton(var eventInfo Event)
var
  x Number
endVar
x = fieldOne
msgInfo("x truncated to 3 places",
        format("W14.6", x.truncate(3))) ; displays truncated version of x
endmethod
```

See also

- ceil, floor, round

OLE



OLE is an acronym for Object Linking and Embedding, a protocol that provides access to the function of another application without having to leave Paradox and open that application each time you want to make a change.

For example, suppose you have tables that contain bitmap graphics, and you want to create a Paradox application that enables users to edit those graphics. One approach is to create the graphics using a paint program that is an OLE server (defined below). Then, use ObjectPAL OLE type methods to make the function of the paint program available to your users (assuming, of course, that your users have the paint program installed on their systems).

The OLE type also includes methods defined for the AnyType type.

Note ObjectPAL and Paradox also support DDE (for Dynamic Data Exchange), another protocol for sharing data.

The following terms are used when discussing OLE operations:

OLE *server* is an application that can provide access to its documents via the OLE mechanism. Paradox is not an OLE server.

OLE *client* is an application that can use the OLE mechanism to access documents created by an OLE server. Paradox is an OLE client.

OLE *object* is a document created using an OLE server. It contains the data you want to use in your Paradox application.

OLE *variable* is an ObjectPAL variable declared to be of type OLE. An OLE variable provides a handle for manipulating an OLE object. In other words, you can use OLE variables in ObjectPAL code to manipulate OLE objects.

For more information about OLE objects and ObjectPAL, refer to the *ObjectPAL Developer's Guide*. See the *User's Guide* for information about using OLE objects and Paradox interactively.

canReadFromClipboard

OLE

Method

Reports whether an OLE object can be pasted from the Clipboard into an OLE variable.

Syntax

canReadFromClipboard ( ) Logical

Description

Returns True if an OLE object can be read (pasted) from the Clipboard into an OLE variable; otherwise, it returns False. This method is useful in a routine that informs the user whether an operation is possible. For example, you can make a menu item dimmed and inactive when this method returns False.

Example

In this example, a form has an OLE object named *oleObject* and a button named *updateOLE*. The **pushButton** method for *oleObject* tests the contents of the Clipboard to determine whether an OLE document can be read into an OLE variable. If the method **canReadFromClipboard** returns True, this example pastes the contents of the clipboard into *oleObject*; otherwise it displays an error message.

```
; updateOLE::pushButton
method pushButton(var eventInfo Event)
var
  oleVar   OLE
endVar

; if an OLE object can be read from the clipboard
if oleVar.canReadFromClipboard() then

  ; read OLE object from clipboard to OLE variable
  oleVar.readFromClipboard()

  ; update the form's OLE object with contents of clipboard
  oleObject = oleVar

else
  beep()
  msgStop("Error", "Can't read from clipboard.")
endif

endmethod
```

See also

- readFromClipboard

edit

OLE

Method

Launches the OLE server and lets the user edit the object or take some other action.

Syntax

edit (const ***oleText*** String, const ***verb*** SmallInt) Logical

Description

Launches the OLE server application and gives control to the user. The argument *oleText* is a string that Paradox passes to the server application. Many server applications can display *oleText* in the title bar. **edit** passes *verb* to the application server to specify an action to take.

verb is an integer that corresponds to one of the OLE server's action constants. The meaning of *verb* varies from application to application, so a *verb* that is appropriate for one application may not be appropriate for another. With the **enumVerbs** method, you can learn what verbs the server supports, then use one of them in the call to **edit**.

If you want to launch an OLE server without using **enumVerbs** first, use 0 (zero) for *verb*—this value is the primary *verb*, and should be supported by all OLE servers.

Example

Suppose the *DocFiles* table stores documents created with your favorite word processor in an OLE field. The table has two fields: File Name (A25) and File (O). This example locates a record in the table then uses **edit** to invoke the word processor, thereby enabling the user to edit the document in the OLE field.

The following example is the code attached to the **pushButton** method for *editFile.*

```
; editFile::pushButton
method pushButton(var eventInfo Event)
var
  myFile   OLE
  tc       TCursor
  fileName String
endVar

fileName = "ProjNote.doc"

if tc.open("DocFiles.db") then
  ; find "ProjNote.doc" in in the File Name field
  if tc.locate("File Name", fileName) then

    ; store in the myFile OLE variable the
    ; contents of the OLE field
    myFile = tc.File

    ; launch the OLE server (the word processor) for myFile
    myFile.edit("Document Manager", 0)

  else
    msgStop("Sorry", "Can't find " + fileName + ".")
  endif
else
  msgStop("Sorry", "Can't open table.")
endif

endmethod
```

See also

- enumVerbs

enumVerbs

OLE

Method

Creates a DynArray listing the actions supported by the OLE server.

Syntax

enumVerbs (var ***verbs*** DynArray[] SmallInt) Logical

Description

Creates a DynArray listing the action commands (called *verbs*) supported by the OLE server.

When you associate an OLE variable with an OLE object (sometimes called an OLE *document*), Paradox knows from what server application the object was generated. Through OLE methods such as **enumVerbs** and **getServerName**, you can ask questions of the server. **enumVerbs** asks the server for a list of supported action commands, then loads them into a dynamic array. Each DynArray index corresponds to the name that the server gives to a specific action; DynArray values correspond to the action constant used by the server. Because the meaning of *verb* varies from application to application, you need to know precisely what verb to pass to the server to instruct it to do what you want.

For example, Windows Paintbrush is an OLE server. Paintbrush has only one action command, named "Edit," with a value of 0. The following code reads from the Clipboard a graphic generated with Paintbrush, generates a dynamic array with **enumVerbs**, then displays the contents of the DynArray in a dialog box.

```
var
  oleVar OLE
  dy DynArray[] SmallInt
endVar

oleVar.readFromClipboard() ; read from the Clipboard into oleVar
oleVar.enumverbs(dy)       ; generate a DynArray of verbs
dy.view()                  ; display DynArray contents in a dialog
```

The preceding code assumes the Clipboard contains an OLE object (a graphic image in this case) that was generated in Paintbrush. The dynamic array contains one element whose index is "Edit" and whose value is 0. Some OLE servers use more than one verb, and would therefore generate a larger list. Other OLE servers use "Edit" but preface the name with an ampersand, such as "&Edit". The ampersand prefix is useful when you want to display action names in a menu. Paradox recognizes the ampersand as a special character and displays "&Edit" as Edit, and designates E as an accelerator key.

Refer to Menu type methods earlier in this chapter to learn more about menus and special characters.

Example

For this example, the *Sounds* table has an alpha field named SoundName and an OLE field named SoundData. Data in the OLE field were copied from the Windows Sound Recorder. The following example uses **enumVerbs** to create a pop-up menu that lists the verbs (actions) for Sound Recorder. Because Sound Recorder supports two actions (Edit and Play), this example lets the user choose to edit or play the sound contained in the OLE field.

```
; editSounds::pushButton
method pushButton(var eventInfo Event)
var
  oleVar  OLE
  p       PopUpMenu
  verbs   DynArray[] SmallInt
  tc      TCursor
  mChoice, tagName String
endvar
soundName = "tada.wav"
tblName = "Sounds.db"

if tc.open(tblName) then
  if tc.locate(1, soundName) then ; search in first field for tada.wav
    oleVar = tc.SoundData        ; assign OLE variable with OLE field value
    oleVar.enumVerbs(verbs)      ; load DynArray with Sound Recorder actions
    forEach tagName in verbs     ; create a pop-up menu from verb listing
      p.addText(tagName)         ; Sound Recorder's verbs are &Edit and &Play
    endForEach
    mChoice = p.show()           ; display "Edit" and "Play" in the pop-up menu

    ; If the user selects from the menu, pass the selected "verb" to the
    ; edit method. verbs[mChoice] evaluates to 0 or 1.
    ; "PdoxWin" appears in Sound Recorder's title bar when Play is selected
    if not mChoice.isBlank() then
      oleVar.edit("PdoxWin", verbs[mChoice])
    endif

  else
    msgStop("Sorry", "Can't find " + soundName + ".")
  endif
else
  msgStop("Sorry", "Can't open " + tblName + ".")
endif

endmethod
```

See also

- readFromClipboard

getServerName

OLE

Method

Reports the name of the OLE server for an OLE object.

Syntax

getServerName ( ) String

Description

Returns as a string the name of the OLE server for an OLE object. This method is useful when you want to inform the user of the OLE server's name.

Example

For this example, assume the *Media* table has an alpha field named MediaName, an alpha field named ServerName, and an OLE field named MediaData. The following code scans through *Media*'s records, filling the ServerName field with the name of the OLE server that generated data in the MediaData field.

```
; getServerName::pushButton
method pushButton(var eventInfo Event)
var
  oleVar  OLE
  tc      TCursor
endvar

if tc.open("Media") then
  tc.edit()
  scan tc for not isBlank(tc.SoundData) :
    oleVar = tc.SoundData
    tc.ServerName = oleVar.getServerName()
  endScan
  tc.close()
else
  msgStop("Error", "Can't open Media table.")
endif

endmethod
```

See also

- edit, enumVerbs

readFromClipboard

OLE

Method

Pastes an OLE object from the Clipboard into an OLE variable.

Syntax

readFromClipboard ( ) Logical

Description

Returns True if an OLE object is successfully read (pasted) from the Clipboard into an OLE variable; otherwise, it returns False.

After an OLE object is read from the Clipboard, changes made to the OLE object while in Paradox do not affect the underlying file.

Example

See the example for **canReadFromClipboard**.

See also

- canReadFromClipboard

writeToClipboard

OLE

Method

Copies an OLE variable to the Clipboard.

Syntax

writeToClipboard ( ) Logical

Description

Copies the original OLE object to the Clipboard as if the server had done the copy (because Paradox is not an OLE server).

This method returns True if an OLE object is successfully written (pasted) to the Clipboard; otherwise, it returns False.

Example

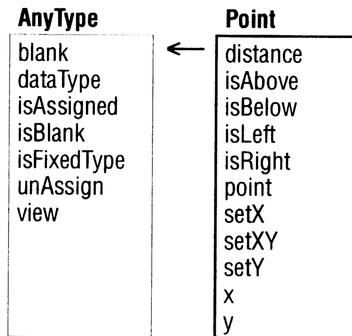
The following example reads an OLE field in a Paradox table and assigns its value to an OLE variable. Then it writes the variable to the Clipboard, where it can be used by Paradox, or by another application. The code assumes that EMPLOYEE.DB has an alpha field named Last Name and an OLE field named Picture.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  empTC TCursor
  oleImage OLE
endVar

empTC.open("Employee.db")     ; EMPLOYEE.DB has OLE images

if empTC.locate("Last Name", "Binkley") then

  oleImage = empTC.Picture    ; Picture is an OLE field
  oleImage.writeToClipboard() ; write contents of OLE field to variable

else
  msgStop("Error", "Can't find Binkley...")
endif
endmethod
```

See also

- readFromClipboard

Point



A Point variable holds information about a point on the screen. To ObjectPAL, the screen is a two-dimensional grid, with the origin at the upper left corner of the design object's container, positive x-values extending to the right, and positive y-values extending down. A Point has an x-value and a y-value, where *x* and *y* are measured in twips (a twip is 1/1440 of an inch; 1/20 of a printer's point.)

Methods defined for the Point type get and set information about screen coordinates and relative positions of points. For example, the size and position properties of a design object are specified in points.

Note ObjectPAL calculates point values relative to the container of the design object in question. For example, if a box contains a button, ObjectPAL calculates the button's position relative to the box. If the button sits in an empty page, ObjectPAL calculates the button's position relative to the page. Methods that take or return Point values as arguments use this relative framework.

The Point type also includes methods defined for the AnyType type. Refer to the "AnyType" section for more information.

distance

Point

Method Returns the distance between two points.

Syntax **distance** (const ***pt*** Point) Number

Description Returns the number of twips between a point and *pt*.

Example

Suppose a form contains 2 boxes: *redBox* and *brownBox*. The **pushButton** method for a button named *getDistance* finds the distance between the upper left corners of the boxes:

```
; brownBox::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = redBox.Position
p2 = brownBox.Position
msgInfo("Distance between boxes", p1.distance(p2))
; shows the distance between the top left corner of
; redBox and the top left corner of brownBox
endmethod
```

See also

❒ isAbove, is Below, isLeft, isRight, x, y

isAbove

Point

Method

Reports whether a point is above another point.

Syntax

isAbove (const ***pt*** Point) Logical

Description

Returns True if the y-coordinate of a point is less than the y-coordinate of *pt*; otherwise, it returns False.

Example

In this example, the **pushButton** method for *convergeBoxes* moves *boxOne* closer to *boxTwo*, until the two boxes converge. Assume that *boxOne* starts to the left of and above *boxTwo*. Each time the button is clicked, *boxOne* will move down until it is on the same vertical plane, then move to the right until it is covered by *boxTwo*.

```
; convergeBoxes::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = boxOne.position            ; get the position of boxOne
p2 = boxTwo.position            ; get the position of boxTwo
if p1.isAbove(p2) then          ; compare the two points
  ; if p1 is higher than p2, move boxOne down
  boxOne.position = point(p1.x(), p1.y() + 100)
else
  if p1.isLeft(p2) then
    ; if p1 is to the left of p2, move boxOne to the right
    boxOne.position = point(p1.x() + 100, p1.y())
  endif
endif
endmethod
```

See also

❒ isBelow, isLeft, isRight

isBelow

Point

Method

Reports whether a point is below another point.

Syntax

isBelow (const ***pt*** Point) Logical

Description

Returns True if the y-coordinate of a point is greater than the y-coordinate of *pt*; otherwise, it returns False.

Example

In this example, the **pushButton** method for *convergeBoxes* moves *boxTwo* closer to *boxOne*, until the two boxes converge. Assume that *boxTwo* starts to the right of and below *boxOne*. Each time the button is clicked, *boxTwo* will move up until it is on the same vertical plane, then move to the left until it is covered by *boxOne*.

```
; convergeBoxes::pushButton
method pushButton(var eventInfo Event)
var
  p1, p2 Point
endVar
p1 = boxOne.position             ; get the position of boxOne
p2 = boxTwo.position             ; get the position of boxTwo
if p2.isBelow(p1) then           ; compare the two points
  ; if p2 is lower than p1, move boxTwo up
  boxTwo.position = point(p2.x(), p2.y() - 100)
else
  if p2.isRight(p1) then
    ; if p2 is to the left of p1, move boxTwo to the left
    boxTwo.position = point(p2.x() - 100, p2.y())
  endif
endif
endmethod
```

See also

- isAbove, isLeft, isRight

isLeft

Point

Method

Reports whether a point is to the left of another point.

Syntax

isLeft (const ***pt*** Point) Logical

Description

Returns True if the x-coordinate of a point is less than the x-coordinate of *pt*; otherwise, it returns False.

Example

See the example for isAbove.

See also

- isAbove, isBelow, isRight

isRight

Point

Method

Reports whether a point is to the right of another point.

Syntax

isRight (const ***pt*** Point) Logical

Description

Returns True if the x-coordinate of a point is greater than the x-coordinate of *pt*; otherwise, it returns False.

Example

See the example for isBelow.

See also

❒ isAbove, isBelow, isLeft

point

Point

Procedure

Casts an expression as a Point.

Syntax

1. **point** ([const ***x*** LongInt, const ***y*** LongInt]) Point
2. **point** (const ***newPoint*** Point) Point

Description

Casts (converts) an expression as a Point.

Example

In this example, you want to vary the position of a box called *rateBox*. The values of an unbound field object named *rateField* range from 1 to 10. The position of *rateBox* is determined by the value in *rateField*. The following code is attached to the **changeValue** method for *rateField*:

```
; rateField::changeValue
method changeValue(var eventInfo ValueEvent)
Const
  baseXPosition = LongInt(3000)
  baseYPosition = LongInt(1000)
endConst
Var
  rateX   LongInt
endVar
try
  ; this if statement will fail if the field contents can't
  ; be compared to the integers 0 and 10 - for instance, if
  ; the user enters a string
  if eventInfo.newValue() = 0 AND eventInfo.newValue() = 10 then
    rateX = (eventInfo.newValue() * 400) + baseXPosition
    rateBox.Position = point(rateX, baseYPosition)
  else
    fail() ; if the value is a number but is out of range,
           ; call the fail block
  endif
onFail
```

```
  disableDefault
  eventInfo.setErrorCode(CanNotDepart)
  msgStop("Stop", "Rating should be a number between 0 and 10.")
endTry

endmethod
```

See also

- ❒ setX, setY

setX

Point

Method

Specifies the x-coordinate of a point.

Syntax

setX (const ***newXValue*** LongInt)

Description

Sets the x-coordinate of a point to *newXValue*. If *newXValue* is not a LongInt, it is converted to a LongInt, and precision may be lost.

Example

In this example, a form contains an ellipse called *circleOne* and a button named *moveRight*. The **pushButton** method for *moveRight* uses **setX** to change the horizontal coordinate of a point, then sets the position of *circleOne* to the changed point:

```
; moveRight::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position        ; get the position of the circle
p1.setX(p1.x() + 100)          ; add 100 twips to the x-coordinate
circleOne.Position = p1        ; set the new position
message(p1)                    ; display coordinates
endmethod
```

See also

- ❒ setY

setXY

Point

Point

Method

Specifies the x- and y-coordinates of a point.

Syntax

setXY (const ***newXValue*** LongInt, const ***newYValue*** LongInt)

Description

Sets the x- and y-coordinates of a point to *newXValue* and *newYValue*. This method combines the functions of **setX** and **setY**. If *newXValue* and *newYValue* are not LongInts, they are converted to LongInts, and precision may be lost.

Example

In this example, a form contains an ellipse called *circleOne* and a button named *moveDiagonal*. The **pushButton** method for *moveDiagonal* uses **setXY** to change the horizontal and vertical coordinates of a point, then sets the position of *circleOne* to the changed point:

```
; moveDiagonal::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position                 ; get the position of the circle
p1.setXY(p1.x() + 100, p1.y() + 100)  ; add 100 twips to each coordinate
circleOne.Position = p1                 ; set the new position
message(p1)                             ; display coordinates
endmethod
```

See also

- setX, setY

setY

Point

Method

Specifies the y-coordinate of a point.

Syntax

setY (const ***newYValue*** LongInt)

Description

Sets the y-coordinate of a point to *newYValue*. If *newYValue* is not a LongInt, it is converted to a LongInt, and precision may be lost.

Example

In this example, a form contains an ellipse called *circleOne* and a button named *moveDown*. The **pushButton** method for *moveDown* uses **setY** to change the vertical coordinate of a point, then sets the position of *circleOne* to the changed point:

```
; moveDown::pushButton
method pushButton(var eventInfo Event)
var
  p1 Point
endVar
p1 = circleOne.position   ; get the position of the circle
p1.setY(p1.y() + 100)     ; add 100 twips to y-coordinate
circleOne.Position = p1   ; set the new position
message(p1)               ; display coordinates
endmethod
```

See also

- setX, setXY

x

Point

Method Returns the x-coordinate of a point.

Syntax **x** ( ) LongInt

Description Returns the x-coordinate of a point.

Example See the example for setX.

See also
- setX, setY, y

y

Point

Method Returns the y-coordinate of a point.

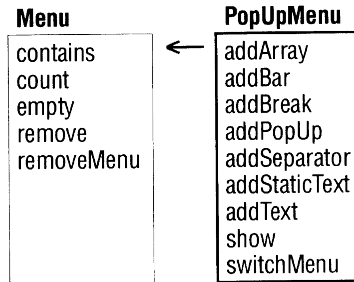
Syntax **y** ( ) LongInt

Description Returns the y-coordinate of a point.

Example See the example for setY.

See also
- setX, setY, x

PopUpMenu



A PopUpMenu is a list of items that appears vertically in response to an Event (usually a mouse click). When the user chooses an item from a pop-up menu, the text of that item is returned to the method.

A PopUpMenu is distinct from a Menu, a list of items that appears horizontally in the application menu bar. However, the PopUpMenu type includes methods defined for the Menu type.

Note Choosing an item from a pop-up menu *does not* trigger the built-in **menuAction** method unless the pop-up menu is attached to a custom menu.

Using PopUpMenu methods, you can

- Build a pop-up menu
- Display the pop-up menu and return the selected item
- Inspect the items in a pop-up menu
- Provide keyboard access

For more information about working with pop-up menus, refer to Chapter 7 in the *ObjectPAL Developer's Guide.*

Note A typical application uses both Menu objects and PopUpMenu objects. See also the "Menu" section earlier in this chapter.

addArray

PopUpMenu

Method Appends elements of an array to a pop-up menu.

Syntax **addArray** (const ***items*** Array[] String)

Description Appends *items* from an array to a pop-up menu.

Example

The following code is attached to a field object's built-in **mouseRightUp** method. When the user right-clicks the field, a list of available payment types appears in a pop-up menu. The following code is attached to the **mouseRightUp** method for *paymentField.*

```
; paymentType::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  items  Array[4] String
  pl     PopUpMenu           ;  addArray is called for this PopUpMenu
  choice String
endVar

disableDefault              ; don't show default Font menu

items[1] = "Visa"
items[2] = "MasterCharge"
items[3] = "Check"
items[4] = "Cash"

pl.addArray(items)          ; add items array to the PopUpMenu
choice = pl.show()          ; display menu, remember choice
if not choice.isBlank() then
  self.value = choice
endif

endmethod
```

See also

- addBar, addBreak, addSeparator, addStaticText, addText

addBar

PopUpMenu

Method

Adds a vertical bar to a pop-up menu.

Syntax

addBar ( )

Description

Adds a vertical bar to a pop-up menu. The **addBar** method marks the beginning of a new column of choices and inserts a vertical bar immediately before the new column. **addBar** is the vertical equivalent of **addSeparator**.

Example

The following code displays a pop-up menu with two columns of choices. The first two choices are displayed in the left column. The remaining choices are displayed in the right column. This code is attached to a field's **mouseRightUp** method.

```
; navField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  navPopUp   PopUpMenu     ; to show a navigate pop-up menu
  navChoice  String        ; store the menu choice
endVar
```

PopUpMenu

```
disableDefault                         ; don't show normal menu for field

navPopUp.addText("Previous record")    ; left menu
navPopUp.addText("First record")
navPopUp.addBar()                      ; add vertical bar
navPopUp.addText("Next record")        ; right menu
navPopUp.addText("Last record")

navChoice = navPopUp.show()            ; invoke menu
; ...
; process choice
; ...

endmethod
```

See also

- addBreak, addSeparator

addBreak

PopUpMenu

Method

Starts a new column in a pop-up menu.

Syntax

addBreak ( )

Description

Starts a new column in a pop-up menu. The first item added after the call to **addBreak** appears at the top of a column to the right of the previous column, and subsequent items appear below it. The **addBreak** method behaves like **addBar** in that it marks the beginning of a new column of choices. However, **addBreak** doesn't create a vertical bar between columns. **addBreak** doesn't create a cascading menu; use **addPopUp** instead.

Example

The following code displays a pop-up menu with nine choices in three vertical columns. This code is attached to *whereToButton*'s **pushButton** method.

```
; whereToButton::pushButton
method pushButton(var eventInfo Event)
var
  navPopUp     PopUpMenu      ; a pop-up of navigation choices
  navChoice    String         ; navigation chosen
endVar

navPopUp.addText("Home")       ; left menu
navPopUp.addText("Left")
navPopUp.addText("End")

navPopUp.addBreak()            ; start second column
navPopUp.addText("Up")
navPopUp.addText("Center")
navPopUp.addText("Down")

navPopUp.addBreak()            ; start third column
navPopUp.addText("PgUp")       ; right menu
navPopUp.addText("Right")
```

```
navPopUp.addText("PgDn")

navChoice = navPopUp.show()     ; invoke menu

; ... process choice

endmethod
```

See also

- addBar, addSeparator

---

addPopUp

PopUpMenu

Method

Adds a pop-up menu to the structure.

Syntax

addPopUp (const ***menuName*** String,
const ***cascadedPopup*** PopUpMenu)

Description

Adds *menuName* and *cascadedPopup* to the current pop-up menu structure, creating a cascading menu. *menuName* appears as an item in the original pop-up menu, and the first item in *cascadedPopup* appears next to it. Subsequent items in *cascadedPopUp* appear in a column below the first item.

Example

This example uses **addPopUp** to attach a cascading menu to a menu bar item (a menu from the Menu type). In this example, the code attached to the built-in **open** method for *thisPage* creates and displays the menu structure. The code attached to *thisPage*'s **menuAction** handles the user's selection because the pop-up menus are attached to a menu bar item.

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, subMenu2 PopUpMenu
endVar

  ; create 2nd level submenu
subMenu2.addText("&Time")
subMenu2.addText("&Date")

  ; add 2nd level to 1st level
subMenu1.addPopUp("&Utilities", subMenu2)

  ; add 1st level to menu bar
mainMenu.addPopUp("&File", subMenu1)

  ; display the menu bar
mainMenu.show()

endmethod
```

The following code is attached to *thisPage*'s **menuAction** method:

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar

choice = eventInfo.menuChoice()
switch
  case choice = "&Time" : msgInfo("Current Time", time())
  case choice = "&Date" : msgInfo("Today's Date", date())
endSwitch

endmethod
```

The next example uses **addPopUp** to create a cascading pop-up menu. This menu structure is not attached to a menu bar item. The code immediately following the call to **show** takes action based on the user's selection; the built-in **menuAction** method is not used.

The following code is attached to the **mouseRightUp** method for *pageTwo*:

```
; pageTwo::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  p1, p2, p3 PopUpMenu
  choice String
endVar

disableDefault                ; don't show normal pop-up menu

p2.addText("&Time")           ; build p2 and p3 submenus
p2.addText("&Date")
p3.addText("&Red")
p3.addText("&Green")
p3.addText("&Blue")

p1.addPopUp("&Utilities", p2) ; create Utilities item and attach p2 to it
p1.addPopUp("&Colors", p3)    ; create Colors item and attach p3 to it

choice = p1.show()            ; display menu and store selection to choice

switch                        ; now take action based on selection
  case choice = "&Red"   : self.color = Red
  case choice = "&Green" : self.color = Green
  case choice = "&Blue"  : self.color = Blue
  case choice = "&Time"  : msgInfo("Current Time", time())
  case choice = "&Date"  : msgInfo("Today's Date", date())
endSwitch

endmethod
```

See also

- addBreak

addSeparator

PopUpMenu

Method

Adds a horizontal bar to a pop-up menu.

Syntax

addSeparator ( )

Description

Appends a horizontal bar to separate item groups in a pop-up menu. **addSeparator** is used to group similar commands together within a menu.

Example

The following example uses **addSeparator** to group pop-up menu commands. The following code is attached to the built-in **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, clrMenu PopUpMenu
endVar

clrMenu.addText("&Red")
clrMenu.addText("&Blue")
clrMenu.addText("&White")

subMenu1.addText("&Time")
subMenu1.addText("&Date")
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About")

mainMenu.addPopUp("&Utilities", subMenu1)
mainMenu.show()
endmethod
```

The following code is attached to the built-in **menuAction** method for *thisPage*:

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  choice String
endVar
choice = eventInfo.menuChoice()
switch
  case choice = "&Red"   : self.color = Red
  case choice = "&Blue"  : self.color = Blue
  case choice = "&White" : self.color = White
  case choice = "&Time"  : msgInfo("Current Time", time())
  case choice = "&Date"  : msgInfo("Today's Date", date())
  case choice = "&About" : eventInfo.setId(MenuHelpAbout)
endSwitch
endmethod
```

See also

- addBar, addBreak

addStaticText

PopUpMenu

Method

Adds an unselectable text string to a pop-up menu.

Syntax

addStaticText (const ***item*** String)

Description

Appends an item to a pop-up menu as unselectable text. Static text is usually used as the title (first item) in a pop-up menu.

Example

The following code is attached to a field object's built-in **mouseRightUp** method. When the user right-clicks the field, a list of available payment types appears in a pop-up menu. This example displays the first item as static text. The following code is attached to the **mouseRightUp** method for *paymentField*.

```
; paymentType::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  items  Array[4] String
  p1     PopUpMenu          ;  addArray is called for this PopUpMenu
  choice String
endVar

disableDefault              ; don't show default Font menu

items[1] = "Visa"
items[2] = "MasterCharge"
items[3] = "Check"
items[4] = "Cash"

                            ; display first item as static text
p1.addStaticText("Payment Method")
p1.addSeparator()           ; add a horizontal separator
p1.addArray(items)          ; add items array to the PopUpMenu
choice = p1.show()          ; display menu, remember choice
if not choice.isBlank() then
  self.value = choice
endif

endmethod
```

See also

- addText

addText

PopUpMenu

Method

Adds a selectable text string to a pop-up menu.

Syntax

1. **addText** (const ***menuName*** String)
2. **addText** (const ***menuName*** String, const ***attrib*** SmallInt)

3. addText (const ***menuName*** String, const ***attrib*** SmallInt, const ***id*** SmallInt)

Description

Appends *menuName* to a pop-up menu as a selectable item. You can use *attrib* to preset the display attribute of *menuName*. ObjectPAL provides constants (like MenuDisabled) for *attrib*; see MenuChoiceAttributes in the Constants dialog box.

The third form of **addText** syntax is used only when the pop-up menu is attached to a Menu object. You can specify an *id* number (of type SmallInt) to identify the menu by number instead of by *menuName*. Then, in the built-in **menuAction** method, you use the *id* number to determine which menu the user chooses.

When you specify a menu *id*, you should use the built-in constant UserMenu as a base constant, then add your own number to it. For example, the following line adds "File" to the *myPopup* PopUpMenu and specifies an *id* number for that menu item:

```
myPopup.addText("File", MenuEnabled, UserMenu + 1)
```

For more information regarding user-defined constants, refer to Chapter 7 in the *ObjectPAL Developer's Guide*. See also the "Menu" section earlier in this chapter for more information on Menu objects and the *id* clause.

You can use an ampersand in an item so the user can select it using the keyboard. For example, the item "&File" would display as File, and the user could choose it by pressing *F*. When testing the user's choice, remember to include the ampersand. In this example, the returned value would be "&File", not "File".

You can also use "\t" to put a Tab between an item and its accelerator. For example, the item "&Edit Data\tF9" would display "Edit Data" left-aligned and "F9" right-aligned. The string value returned in this case would be "&Edit Data\tF9".

Example

The following examples demonstrate variations of **addText** syntax.

For the first example, assume a form has an unbound field named *payField*. When the user right-clicks the field, a list of available payment methods appears in a pop-up menu. The user can choose from the list to insert that value into the field or press *Esc* to cancel. The following code goes in the Var window for *payField*:

```
; payField::var
var
  payPopUp PopUpMenu
  mChoice  String
endVar
```

The following code is attached to the **open** method for *payField*. When the field opens for the first time, this code adds four items to the *payPopUp* PopUpMenu. This code does not display the pop-up menu; it just prepares the menu for later.

```
; payField::open
method open(var eventInfo Event)

payPopUp.addText("Visa")
payPopUp.addText("MasterCard")
payPopUp.addText("Check")
payPopUp.addText("Cash")

endmethod
```

The following code is attached to *payField*'s built-in **mouseRightUp** method. When the user right-clicks the field, this method displays the menu with **show**, then inserts the user's choice in the unbound field.

```
; payField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault              ; don't show default pop-up menu

mChoice = payPopUp.show()    ; display menu, store selection to mChoice
if not isBlank(mChoice) then ; if user does not press Esc
  self.value = mChoice       ; insert mChoice in unbound field
endif
endmethod
```

The next example demonstrates how you can use the *id* clause for pop-up menus attached to a Menu object. This code establishes user-defined constants to make it easy to remember the menu *id* assignments. The following code goes in the Const window for *thisPage*.

```
; thisPage::const
Const
  MenuRed    = 1  ; define constant values for menu ids
  MenuBlue   = 2
  MenuWhite  = 3
  MenuTime   = 4
  MenuDate   = 5
  MmenuAbout = 6
endConst
```

The following code is attached to the **open** method for *thisPage*. To control the menu display attributes, this code uses built-in constants such as MenuEnabled. To identify each menu item by number, the code uses the constants defined in the Const window for *thisPage* (*MenuRed*, *MenuBlue*, and so forth).

```
; thisPage::open
method open(var eventInfo Event)
var
  mainMenu Menu
  subMenu1, clrMenu PopUpMenu
endVar

  ; add text to pop-up menus and use user-defined constants
```

```
clrMenu.addText("&Red", MenuEnabled, MenuRed + UserMenu)
clrMenu.addText("&Blue", MenuEnabled, MenuBlue + UserMenu)
clrMenu.addText("&White", MenuEnabled, MenuWhite + UserMenu)

subMenu1.addText("&Time", MenuEnabled, MenuTime + UserMenu)
subMenu1.addText("&Date", MenuEnabled, MenuDate + UserMenu)
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About", MenuEnabled, MenuAbout + UserMenu)

  ; attach pop-up menus to mainMenu and display the menu bar
mainMenu.addPopUp("&Utilities", subMenu1)
mainMenu.show()
endmethod
```

The following code is attached to the **menuAction** method for *thisPage*. This example evaluates menu selections by ID number rather than by the name specified in *menuName*.

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuId SmallInt
endVar

menuId = eventInfo.id()   ; store menu id number in menuId

switch
  case menuId = MenuRed   + UserMenu  : self.color = Red
  case menuId = MenuBlue   + UserMenu : self.color = Blue
  case menuId = MenuWhite  + UserMenu: self.color = White
  case menuId = MenuTime   + UserMenu: msgInfo("Current Time", time())
  case menuId = MenuDate   + UserMenu: msgInfo("Today's Date", date())
  case menuId = MenuAbout  + UserMenu: eventInfo.setId(MenuHelpAbout)
endSwitch

endmethod
```

See also

- addStaticText

---

show

PopUpMenu

Method

Displays a pop-up menu and returns the item selected.

Syntax

show **(** [const ***xTwips*** SmallInt, const ***yTwips*** SmallInt] **)** String

Description

Displays a pop-up menu and returns the item selected. If the user presses *Esc* instead of making a selection, the returned value is a zero-length string. The optional arguments *xTwips* and *yTwips* specify the coordinates, in twips, of the upper left corner of the pop-up menu. If not specified, they are set to the x- and y-coordinates of the pointer.

Example

For the following example, assume a form has an unbound field named *payField.* When the user right-clicks the field, a list of payment types appears in a pop-up menu. The user can choose from the list to insert that value into the field or press *Esc* to cancel. The following code goes in the Var window for *payField*:

```
; payField::var
var
  payPopUp PopUpMenu
  mChoice  String
endVar
```

The following code is attached to the **open** method for *payField.* When the field opens for the first time, this code adds four items to the *payPopUp* PopUpMenu. This code does not display the pop-up menu; it just prepares the menu for later.

```
; payField::open
method open(var eventInfo Event)

payPopUp.addText("Visa")
payPopUp.addText("MasterCard")
payPopUp.addText("Check")
payPopUp.addText("Cash")

endmethod
```

The following code is attached to *payField*'s built-in **mouseRightUp** method. When the user right-clicks the field, this method displays the menu with **show**, then inserts the user's choice into the unbound field.

```
; payField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault              ; don't show default pop-up menu

mChoice = payPopUp.show()    ; display menu, store selection to mChoice
if not isBlank(mChoice) then ; if user does not press Esc
  self.value = mChoice       ; insert mChoice into unbound field
endif
endmethod
```

See also

- empty in the Menu type

switchMenu

PopUpMenu

Keyword

Builds and displays a pop-up menu and handles the menu choice.

Syntax

switchMenu
  CaseList
  [otherwise : *Statements* **]**
endSwitchMenu

CaseList is any number of statements in the following form:

CASE *menuItem* : *Statements*

Description

Uses the values of the *menuItem* argument in each *CaseList* to create and display a pop-up menu. The *Statements* following each *menuItem* specify how to handle each menu choice. The optional **otherwise** clause specifies what to do if the user closes the menu without making a choice (for example, by pressing *Esc*).

Example

The following example uses **switchMenu** to create, display, and process the choice from a pop-up menu. A string describing the selection is displayed in the message window of the status line.

```
; actionButton::pushButton
method pushButton(var eventInfo Event)
switchMenu
  case "Add"    : message("Add selected.")
  case "Edit"   : message("Edit selected.")
  case "Delete" : message("Delete selected.")
  otherwise     : message("No selection from menu.")
endSwitchMenu
endmethod
```

See also

❐ addText, show

Query

Query

executeQBE isAssigned Query writeQBE

An ObjectPAL Query variable represents a QBE query. You can use ObjectPAL to create and execute queries from methods just as if you were using Paradox interactively. You can execute a query from a query file, a query statement, or a string.

Most methods for working with queries are defined for the Database type, because in certain applications you will need to specify the database that contains the tables to query.

executeQBE

Query

Method

Executes a QBE query.

Syntax

1. **executeQBE** ( ) Logical
2. **executeQBE** (const ***ansTbl*** String) Logical
3. **executeQBE** (const ***ansTbl*** Table) Logical
4. **executeQBE** (var ***ansTbl*** TCursor) Logical

Description

Executes a query created in an ObjectPAL method and writes the results to *ansTbl*. In syntax 1, where *ansTbl* is not specified, **executeQBE** writes to ANSWER.DB in the user's private directory. In syntax 2, specify the answer table as a string; if you don't include a file extension, *ansTbl* is a Paradox table by default. In syntax 3, where *ansTbl* is a Table variable, *ansTbl* must be assigned and valid. If you use syntax 4 to write query results to a TCursor, the results are stored in system memory only; a table is not created on disk.

If **executeQBE** is successful—if *ansTbl* or ANSWER.DB is created—this method returns True (even if the resulting table is empty); otherwise it returns False.

A query in a method begins with a Query variable, the = sign, and the keyword **Query** followed by a blank line. Next comes the body of the query, and another blank line. The query ends with the keyword **EndQuery**.

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this method with **executeQBEString** in the Database type.) You can use absolute paths or aliases in the query definition.

Example

For this example, the **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable, then runs it with **executeQBE**. The query statement in this example is an insert query: it retrieves certain records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria for this example uses a tilde variable *myState* that designates Oregon customers to be included in the results. Since "OR" is the abbreviation for Oregon, the *myState* variable must evaluate to a quoted string to distinguish it from the **OR** query expression.

```
; getReceivables::pushButton
method pushButton(var eventInfo Event)
var
  qVar    Query
  myState String
  tv      TableView
endVar

; add samp alias for the PdoxWin sample directory
addAlias("samp", "Standard", "c:\\pdoxwin\\sample")

; OR is a keyword, but since it's also the abbreviation for
; Oregon, it must be enclosed in quotes
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = Query

         :samp:Customer.db | Customer No | Name  | State/Prov | Phone  |
                           | _cust       | _name |   ~myState | _phone |

          :samp:Orders.db  | Customer No | Balance Due |
                           | _cust       | 0, _balDue  |

               myCust.db   | Customer No | Name  | Balance Due | Phone  |
                 insert    | _cust       | _name | _balDue     | _phone |


       EndQuery

executeQBE(qVar, "myCust.db")    ; put results into myCust.db
tv.open("myCust.db")             ; view the table

endmethod
```

See also

- QUERY, writeQBE
- executeQBEString and executeQBEFile in the Database type

Query

isAssigned

Query

Method

Reports whether a Query variable has an assigned value.

Syntax

isAssigned( ) Logical

Description

Returns True if a Query variable has been assigned a value; otherwise, it returns False. This method does not check the validity of the assigned query.

Example

In the following example, the call to **isAssigned** returns True, because the Query variable *qVar* has been assigned a value, even though the value is not a valid query.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  qVar Query
endVar

qVar = Query

     This is not a query

     endQuery

msgInfo("Assigned?", qVar.isAssigned())    ; displays True

endmethod
```

See also

- Query

query

Query

Keyword

Begins a query statement.

Syntax

query

```
tableName | fieldName | [ fieldName | ]*
          | criteria  | [ criteria  | ]*

[ tableName | fieldName | [ fieldName | ]*
            | criteria  | [ criteria  | ]* ] *
```

endQuery

Description

Marks the beginning of a QBE statement. A QBE statement extracts data from one or more tables according to the fields specified in *fieldName* and the selection criteria, where *criteria* can be any valid QBE expression.

A query statement begins with a Query variable, the = sign, and the keyword **query** followed by a blank line. Next comes the body of the query, and another blank line. The query ends with the keyword **endQuery**.

You don't have to list all the fields in the table. Instead, you can list only those fields that affect the query, as in this example.

```
var myQBE query endvar
myQBE = query

          Customer | Customer No | Name  |
                   | Check       | A..   |

      endQuery
```

The previous query statement retrieves from the *Customer* table customer numbers whose name start with "A". (The *Customer* table has more than two fields, but only two fields are specified in the example.)

The blank lines above and below the body of the query are required. It is not neccessary to align the vertical field separators (although alignment makes it more readable). ObjectPAL interprets the following code exactly as it interprets the code in the previous example.

```
var myQBE Query endvar
myQBE = query

Customer | Customer No          |Name  |
|Check| A..   |

   endQuery
```

If you construct a query statement that includes two or more tables, you must separate each table with a blank line, as follows:

```
var myQBE Query endvar
myQBE = query

          Customer | Customer No | Name  | Phone |
```

```
                   | _x          | Check | Check |

         Orders    | Customer No | Balance Due |
                   | _x          | Check 0     |

      endQuery
```

Because this kind of query is not a quoted string, it can contain tilde variables. (Compare this kind of query with a query string in the Database type). You can use absolute paths or aliases in the query definition.

Example

For this example, the **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable, then runs it with **executeQBE**. The query statement in this example is an insert query; it retrieves certain records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria for this example uses a tilde variable *myState* that designates Oregon customers to be included in the results. Since "OR" is the abbreviation for Oregon, the *myState* variable must evaluate to a quoted string to distinguish it from the **OR** query expression.

```
; getReceivables::pushButton
method pushButton(var eventInfo Event)
var
  qVar    Query
  myState String
  tv      TableView
endVar

; add samp alias for the PdoxWin sample directory
addAlias("samp", "Standard", "c:\\pdoxwin\\sample")

; OR is a keyword, but since it's also the abbreviation for
; Oregon, it must be enclosed in quotes
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = query

      :samp:Customer.db | Customer No | Name  | State/Prov | Phone  |
                        | _cust       | _name |   ~myState | _phone |

        :samp:Orders.db | Customer No | Balance Due |
                        | _cust       | 0, _balDue  |

              myCust.db | Customer No | Name  | Balance Due | Phone  |
                 insert | _cust       | _name | _balDue     | _phone |


      endQuery

executeQBE(qVar, "myCust.db")    ; put results into myCust.db
tv.open("myCust.db")             ; view the table

endmethod
```

See also

- executeQBE

- DataBase type methods executeQBEFile and executeQBEString

writeQBE

Query

Method

Writes a query statement to a specified file.

Syntax

writeQBE (const ***fileName*** String) Logical

Description

Writes a previously defined query statement to the file specified in *fileName*. If *fileName* exists, it is overwritten without asking for confirmation. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Example

In this example, assume a form has a button named *getDest*. When the form opens, this example determines whether the GETDEST.QBE file exists in the current directory. If the file does not exist, the built-in **open** method for *pageOne* uses **writeQBE** to write a query string to GETDEST.QBE. The built-in **pushButton** for *getDest* runs the query with **executeQBEFile**, then opens the table. This code assumes the :MAST: alias has already been defined. Following is the code attached to the **open** method for *pageOne*:

```
; pageOne::open
method open(var eventInfo Event)
Var
  qVar Query
endVar

; if the GetDest.qbe query file dosn't exist
if not isfile("GetDest.qbe") then

  ; construct a query
  qVar = query

          :mastApp:Dest | Destination Name | Avg Temp (F) |
                        | Check            | Check 70     |

        endQuery

  ; write the query statement to the GetNames.qbe file
  writeQBE(qVar, "GetDest.qbe")

endif

endmethod
```

The following code is attached the built-in **pushButton** method for the *getDest* button. This code does not check whether GETDEST.QBE exists because the **open** method for the page ensures the file is available.

Query

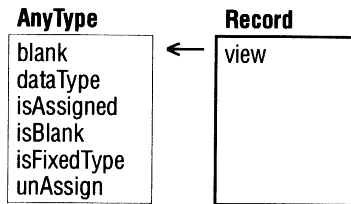
```
; getDest::pushButton
method pushButton(var eventInfo Event)
var
  tv TableView
endVar

; execute the query file and store results in MyDest.db
executeQBEFile("GetDest.qbe", "MyDest.db")
; open the table
tv.open("MyDest")

endmethod
```

Another use for this method is to use ObjectPAL to create and save a query that the user can run interactively using the Query Editor.

See also

- executeQBE
- Database type methods executeQBEFile and executeQBEString

Record



ObjectPAL provides the Record type as a programmatic, user-defined collection of information, similar to a **record** in Pascal or a **struct** in C. Records defined in ObjectPAL code are separate and distinct from records associated with a table.

Here's the syntax for declaring a Record data type:

```
type
recordName = record
                    fieldName fieldType
                  [ fieldName fieldType ]*
              endRecord
endType
```

One or more *fieldNames* identify fields (columns) of the record, and *fieldType* is one of the data types. Declare records in a design object's Type window. See Chapters 5 and 9 for more information about declaring and using records.

Once you declare a Record data type, you can use the = and <> comparison operators to compare one record to another. You can also use the assignment (=) operator to copy the contents of one record to another.

The Record type also includes methods defined for the AnyType type.

view

Record

Method

Displays in a dialog box the value of a record.

Syntax

view ([const ***title*** String])

Description

Displays in a modal dialog box the value of a record. ObjectPAL execution suspends until the user closes this dialog box. You can specify the dialog box's title in *title*, or you can omit *title* to display the variable's data type. Unlike many data types, values in a Record can't be changed when displayed in a **view** dialog box. Refer to

AnyType earlier in this chapter for more information regarding **view** and other data types.

Example

The following example uses a type named MyRecord. The **pushButton** method for *getAndViewRec* declares a variable called *myRec* of type MyRecord. This method then opens a TCursor to the *Customer* table, fills *myRec* with the *Customer No* and *Name* field values from the first record, and uses **view** to display the record in a dialog box. This operation is then repeated for the second record in *Customer*.

The following code is attached to the Type window for *getAndViewRec*. This code creates a user-defined type named MyRecord.

```
; getAndViewRec::Type
type
  MyRecord = record          ; define a Record structure
               ID    String
               Name  String
             endRecord
endType
```

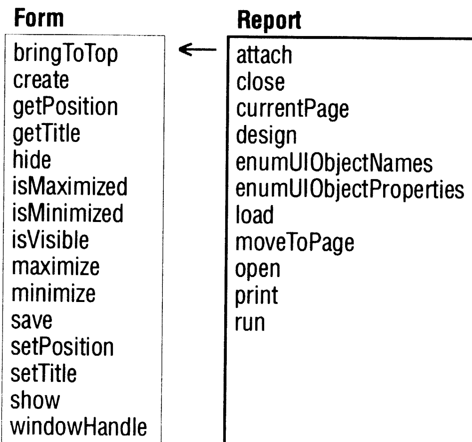
This code is attached to the **pushButton** method for a button named *getAndViewRec*:

```
; getAndViewRec::pushButton
method pushButton(var eventInfo Event)
var
  recOne, recTwo MyRecord
  tc             TCursor
endVar

if tc.open("Customer.db") then
  recOne.ID = tc."Customer No"      ; put some values into the record
  recOne.Name = tc."Name"
  recOne.view("First record")       ; display the record in a dialog box

  tc.nextRecord()                   ; move to the next record

  recTwo.ID = tc."Customer No"      ; get new values
  recTwo.Name = tc."Name"
  recTwo.view("Second record")      ; display second record

  msgInfo("recOne = recTwo?", recOne = recTwo)  ; displays False

  recOne = recTwo                   ; now both records have the same values
  msgInfo("recOne = recTwo?", recOne = recTwo)  ; displays True

else
  msgStop("Stop", "Couldn't open the Customer table.")
endIf
endmethod
```

See also

- Methods and procedures defined for the Array and DynArray types

Report



A Report variable is a handle to a report. You use Report variables in code to manipulate the report onscreen and to view and print the report. Report methods control the window's size, position, and appearance.

Use **load** to load a report file in the Report Design window; use **open** to open the report in a report window, and use **print** to open a report and print it. You cannot attach methods to objects in a report.

The Report type includes several methods defined for the Form type. See the *User's Guide* for information about working with reports interactively.

attach

Report

Method

Associates a Report variable with an open report.

Syntax

attach (const ***reportTitle*** String) Logical

Description

Associates a Report variable with an open report. *reportTitle* specifies the title of an open report.

Note The argument *reportTitle* refers to the text displayed in the title bar of the Report window, not to the file name. You can use **getTitle** to return this text, or you can use **setTitle** to specify a title yourself.

Example

In this example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **pushButton** method for *printStock* attaches to the open report by way of its title, then prints it.

```
; printStock::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
endVar
; the Stock report was opened and retitled by the form's open method
stockRep.attach("Stock Report")  ; attach by report's title
stockRep.print()                 ; print the report
endmethod
```

This code is attached to the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
var
  stockRep  Report
endVar
if eventInfo.isPreFilter()
  then
    ;code here executes for each object in form
  else
    ;code here executes just for form itself
    stockRep.open("stock.rsl")
    stockRep.setTitle("Stock Report")
    bringToTop()         ; bring this form back to the top
endif
endmethod
```

See also

- open
- getTitle, setTitle in the Form type

close

Report

Method

Closes a window.

Syntax

close ( )

Description

Closes a Report window. Closing a report with **close** is equivalent to choosing Close from the Control menu.

Example

In this example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **close** method for the form attaches to the open report by way of its title, then closes it when the form closes.

```
; thisForm::close
method close(var eventInfo Event)
```

```
var
  stockRep  Report
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; the Stock report was opened and retitled by
    ; the form's open method
    stockRep.attach("Stock Report")
    stockRep.close()
endif
endmethod
```

See also

- open

currentPage

Report

Method

Returns the current page number of a report.

Syntax

currentPage ( ) SmallInt

Description

Returns the current page number of a report.

Example

In this example, the **pushButton** method for *plusTwoPages* attempts to attach to an open report, and, if this fails, opens the report. Once the *ordersRep* variable points to an open report, the method moves the report forward two pages.

```
; plusTwoPages::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
endVar
; report might be open already, so attempt an attach first
if NOT ordersRep.attach("Report : ORDERS.RSL") then
  if NOT ordersRep.open("Orders.rsl") then
    msgStop("FYI", "Could not open or attach to report.")
    return
  endif
endif
; move to two pages past the current page
ordersRep.moveToPage(ordersRep.currentPage() + 2)
bringToTop()   ; make this form the top layer again
endmethod
```

See also

- moveToPage

design

Report

Method

Switches a running report to the Report Design window.

Syntax

design ( ) Logical

Description

Switches a running report to the Report Design window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL). For more information about saving and delivering reports, refer to the *User's Guide*.

Use **run** to run a report already loaded in the Report Design window.

Note Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

Example

In this example, assume the form's **open** method opened the STOCK.RSL report and retitled the window to "Stock Report". The **pushButton** method for *stockDesign* attaches to the open report by way of its title, then switches the report to the Report Design window.

```
; stockDesign::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
endVar
; the form's open method opened and retitled the Stock report
stockRep.attach("Stock Report")
stockRep.design()          ; switch to Report Design window
endmethod
```

See also

- load, open, run

enumUIObjectNames

Report

Method

Creates a table listing the UIObjects contained in a report.

Syntax

enumUIObjectNames (const ***tableName*** String) Logical

Description

Creates a Paradox table listing the name and type of each object contained in a report. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails. You can include an alias or path in *tableName*; if no alias

or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of *tableName* is:

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

Example

In the following example, the **pushButton** method for *describeReport* uses **enumUIObjectNames** and **enumUIObjectProperties** to document a report.

```
; describeReport::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
  tempTable TableView
endVar
ordersRep.load("Orders.rsl")                       ; load report in Report Design
                                                     window
ordersRep.enumUIObjectNames("ordnames.db")         ; write names to table
ordersRep.enumUIObjectProperties("ordprops.db")    ; write properties to table
ordersRep.close()
tempTable.open("ordnames")                         ; observe your handiwork
tempTable.wait()
tempTable.close()
tempTable.open("ordprops")
tempTable.wait()
tempTable.close()
endmethod
```

See also

- enumUIObjectProperties

enumUIObjectProperties

Report

Method

Creates a table listing the properties of each UIObject contained in a report.

Syntax

enumUIObjectProperties (const ***tableName*** String) Logical

Description

Creates a Paradox table listing the name, property name, and property value of each object contained in a report. Use the argument *tableName* to specify a name for the table. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is already open, this method fails.

The structure of *tableName* is:

Field Name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyType	A	48
PropertyValue	A	255

Example

See the example for enumUIObjectNames.

See also

- enumUIObjectNames

load

Report

Method

Opens a report in the Report Design window.

Syntax

load (const ***reportName*** String) Logical

Description

Opens *reportName* in the Report Design window. This method works only with saved reports or forms (.RSL or .FSL); it does not work with delivered reports or forms (.RDL or .FDL). For more information about saving and delivering reports, refer to the *User's Guide.*

Compare this method to **open**, which runs a report.

Note Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** method in the System type for more information.

Example

In this example, the **pushButton** method for the *loadOrders* button loads the ORDERS.RSL report in the Report Design window, creates a text box in the page header, and writes a string to the text box.

```
; loadOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
  pageTitle UIObject
endVar
if ordersRep.load("Orders.rsl") then
  ; assume report has room in the page header for a text box
  pageTitle.create(TextTool, 1440*3, 720, 1440*2, 360, ordersRep)
  pageTitle.Name = "NewTitleText"
  pageTitle.Text = "Orders Report " + String(time())
  pageTitle.Color = LightBlue
  pageTitle.Visible = True
  ordersRep.run()
endif
endmethod
```

See also

- design, open, run

moveToPage

Report

Method

Displays a specified page of a report.

Syntax

moveToPage (const ***pageNumber*** SmallInt) Logical

Description

Displays the page of a report specified in *pageNumber*. This method doesn't make the report active. If you want to make the report active, follow **moveToPage** with **bringToTop** (see the Form type for more information on **bringToTop**).

Example

See the example for currentPage.

See also

- currentPage
- bringToTop in the Form type

open

Report

Method

Opens a report.

Syntax

1. **open** (const ***reportName*** String [, ***windowStyle*** LongInt]) Logical
2. **open** (const ***reportName*** String, const ***windowStyle*** LongInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt) Logical
3. **open** (const ***openInfo*** ReportOpenInfo) Logical

Description

Displays the report specified in *reportName* in a report window. Optional arguments specify the location of the upper left corner of the report (*x* and *y*), the width and height (*w* and *h*), and style (*windowStyle*).

ObjectPAL provides constants for *windowStyle*; see WindowStyles in the Constants dialog box.

You can specify more than one window style by adding the constants. For example, the following code opens a report window that has boxes the user can click to minimize and maximize the window:

```
salesReport.open("sales.rsl", WinStyleMaximizeBox + WinStyleMinimizeBox)
```

Syntax 3 lets you specify form settings from *openInfo*, a record of type ReportOpenInfo. A ReportOpenInfo record has the following structure:

```
x, y, w, h          LongInt  ; size and position of report
name                String   ; name of report to open (preView)
masterTable         String   ; master table name
queryString         String   ; run this query (actual query string)
restartOptions      SmallInt ; one of the ReportPrintRestart constants
```

Note Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

Example

In this example, the **pushButton** method for *openSmall* opens the ORDERS.RSL report and minimizes it by supplying a window style constant of WinStyleMinimize.

```
; openSmall::pushButton
method pushButton(var eventInfo Event)
var
  ordersRep Report
endVar
ordersRep.open("Orders.rsl", WinStyleMinimize)   ; open Orders Report minimized
endmethod
```

See also

❐ close, design, load,

print

Report

Beginner

Method

Prints a report.

Syntax

1. **print** ( ) Logical
2. **print** (const ***reportName*** String, const ***reportPrintRestart*** SmallInt) Logical
3. **print** (const ***ri*** ReportPrintInfo) Logical

Description

Prints a report. With syntax 1, Paradox opens the Print dialog box for the current report, which allows the user to specify print settings. Syntax 2 lets you specify a report name and set restart options. Syntax 3 lets you set print settings with a ReportPrintInfo record. ReportPrintInfo records are predeclared and have the following structure:

```
name          String   ; run this report if not already open
masterTable   String   ; master table name
```

```
queryString     String   ; run this query (actual query string)
restartOptions  SmallInt ; do what when data changes while printing report
                         ; one of the ReportPrintRestart constants
printBackwards  Logical  ; forward FALSE, backward TRUE, default false
makeCopies      Logical  ; Who does copies: Paradox or printer?
                         ; If True, Paradox make copies
panelOptions    SmallInt ; one of the ReportPrintPanel constants
nCopies         SmallInt ; number of copies, default is 1
startPage       LongInt  ; starting page, default is 1
endPage         LongInt  ; ending page def: ending page
pageIncrement   SmallInt ; Page increment for multi-pass
                         ; printing, default is 1
xOffset         LongInt  ; horizontal page offset
yOffset         LongInt  ; vertical page offset
orient          SmallInt ; one of the ReportOrientation constants
```

Example

For examples of printing using syntax 1, see the example for attach. The following example shows how to use syntax 3 to print using a ReportPrintInfo record.

```
; printWithRecord::pushButton
method pushButton(var eventInfo Event)
var
  stockRep  Report
  repInfo   ReportPrintInfo
endVar
; first, set up the repInfo record

repInfo.nCopies = 2
repInfo.makeCopies = True
repInfo.name = "Stock"
stockRep.print(repInfo)
endmethod
```

See also

- run, open

run

Report

Method

Runs a report from the Report Design window.

Syntax

run ( ) Logical

Description

Switches a report from the Report Design window to the Report window. This method works only with saved reports (.RSL); it does not work with delivered reports (.RDL). For more information about saving and delivering reports, refer to the *User's Guide*.

To switch a running report to Report Design window, use **design**.

Note Some report actions are especially processor-intensive. In some situations, you might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. See the **sleep** procedure in the System type for more information.

Example See the example for load.

See also

- design

Session

Session

addAlias	getAliasPath
addPassword	getNetUserName
advancedWildCardsInLocate	ignoreCaseInLocate
blankAsZero	isAdvancedWildcardsInLocate
close	isAssigned
enumAliasNames	isBlankZero
enumDatabaseTables	isIgnoreCaseInLocate
enumDriverCapabilities	lock
enumDriverInfo	open
enumDriverNames	removeAlias
enumDriverTopics	removeAllPasswords
enumEngineInfo	removePassword
enumFolder	retryPeriod
enumOpenDatabases	saveCFG
enumUsers	setAliasPath
	setRetryPeriod
	unlock

A Session object represents a channel to the database engine. Opening a Paradox application opens one session by default, and you can use ObjectPAL to open other sessions from within an application; it is not necessary to open other sessions to use procedures from the Session type. The number of other sessions you can open depends on the system environment. Each session uses one user count.

Only the default session can be managed using Paradox interactively. You must manage other sessions with ObjectPAL.

Locks set by ObjectPAL interact as peers with locks set interactively in the same session.

addAlias

Session

Method/Procedure

Adds a database alias to a session.

Syntax

addAlias (const ***aliasName*** String, const ***type*** String, const ***path*** String) Logical

Description

Adds a database alias to a session. Specify the alias name in *aliasName*, the alias type ("Standard") in *type*, and the full DOS path in *path*.

An added alias using this method is known only to the session for which it is defined, and exists only until the session is closed.

Example

The following example adds an alias to the current session, then supplies the new alias to the **open** method defined for the Database type. This code is attached to the built-in **open** method for the *pageOne* page:

```
; pageOne::open
method open(var eventInfo Event)
var
  custInfo Database
endVar

; add the CustomerInfo alias to the current session
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")

; now use the alias specify the database to open
custInfo.open("CustomerInfo") ; opens the CustomerInfo database

endmethod
```

See also

- getAliasPath

addPassword

Session

Method/Procedure

Presents a password allowing access to a protected table.

Syntax

addPassword (const ***password*** String)

Description

Presents to a Paradox session the password specified in *password*. Subsequent attempts to access a table protected using that password are not challenged. The argument *password* can represent either an owner password or an auxiliary password. Auxiliary passwords generally confer less comprehensive rights than owner passwords. *password* is case-sensitive; a table protected with "Sesame" won't open for "SESAME".

Passwords added using this method are valid only for the session they are presented in, and are in effect only until the session is closed. Presenting a password does not affect the state of tables: for example, an open table remains open.

Access to tables opened before the password is presented is controlled by passwords previously presented. For instance, if a table was opened using an auxiliary password, the access rights to that table do not change upon presentation of the owner password. To confer owner rights to a previously-opened table, you should first close the table, then present the owner password, then reopen the table.

Use **removePassword** to restore protection.

Example

The following example acquires a password from the user, then presents it to the current session.

```
; getAddPass::pushButton
method pushButton(var eventInfo Event)
var
  newPass String
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  newPass.view("Enter Password to Add")
  ses.addPassword(newPass)
else
  msgStop("Help!","Session variable is not Assigned!")
endif
endmethod
```

See also

- removeAllPasswords, removePassword

advancedWildcardsInLocate

Session

Procedure

Specifies whether this session can use advanced wildcards in locate operations.

Syntax

advancedWildcardsInLocate ([const ***yesNo*** Logical])

Description

Specifies whether the current session should use advanced wildcards found in pattern strings during locate operations. If *yesNo* is Yes, pattern strings used in locate operations can contain advanced wildcard characters; if set to No, pattern strings in locate operations cannot contain advanced wildcards. If omitted, *yesNo* is Yes by default.

Example

This example calls **advancedWildcardsInLocate**, if necessary, to specify that advanced wild cards can be used in a locate operation. Then the code continues with a call to **locatePattern** that uses an advanced wildcard pattern.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  thisSession Session
endvar

if tc.open("Orders.db") then

  ; if advanced wild cards can't be used in patterns
  if NOT isAdvancedWildcardsInLocate() then
    ; specify that this session can use advanced
    ; pattern characters in subsequent locate operations
```

```
    advancedWildcardsInLocate(Yes)
  endif

  if tc.locatePattern("Ship VIA", "[^UPS]") then
    msgInfo("Order Number", tc."Order No")
  else
    msgStop("Error", "Can't find record")
  endif
else
  msgStop("Error", "Can't open Orders table.")
endif

endmethod
```

See also

- isAdvancedWildcardsInLocate

blankAsZero

Session

Method/Procedure

Specifies whether to treat blank values as zeros in calculations.

Syntax

blankAsZero (const ***yesNo*** Logical)

Description

Specifies whether to assign blank numeric fields a value of 0 in calculations. If *yesNo* is Yes, blanks are treated as zeros. If *yesNo* is No, they are not.

Calculations affected by **blankAsZero** include:

- Calculated fields in forms and reports
- Calculations in queries
- Column calculations that involve either the number of fields or the number of non-blank fields, for example, those performed with **cCount**, **cAverage**, and others

You may want to perform these calculations differently depending on the state of **blankAsZero**. You can use **isBlankZero** to test the state, and **blankAsZero** to set it.

Example

This example sets **blankAsZero**, if necessary, to True so that a call to the **cAverage** method treats blank field values as zeros.

```
; getAvgPmt::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Orders.db") then
  if isBlankZero() then
    blankAsZero(True)
  endif
```

```
  msgInfo("Average Amount Paid", tc.cAverage("Amount Paid"))

else
  msgStop("Error", "Can't open Orders table.")
endif

endmethod
```

See also

- isBlankZero

close

Session

Method

Closes a session.

Syntax

close ( ) Logical

Description

Ends a session by closing the channel to the database engine. **close** frees one user count, and makes the Session variable unassigned.

Example

For the following example, assume that the variable *ses* is assigned to an open session. This example closes the session *ses*.

```
; closeSession::pushButton
method pushButton(var eventInfo Event)
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  if ses.close() then
    msgInfo("We have TouchDown","Session close Successful.")
  else
    msgStop("Crash and Burn","Session close Unsuccessful.")
  endif
else
  msgStop("Help!","Session variable is not Assigned! Who am I?")
endif
endmethod
```

See also

- open, setCurrent

enumAliasNames

Session

Method/Procedure

Creates a Paradox table listing the names of database aliases available to a session.

Syntax

enumAliasNames (const ***tableName*** String) Logical

Description

Creates a Paradox table *tableName* listing the aliases of databases available to a session. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

You can include an alias or path in *tableName;* if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is

Field Name	Type	Size
DBName	A	32*
DBType	A	32
DBPath	A	82

Example

In this example, the **pushButton** method for *getAliasButton* writes the alias names active for the *ses* session to the table *AliasNam,* then it views the table.

```
; getAliasButton::pushButton
method pushButton(var eventInfo Event)
var
  tvl  TableView
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  ses.enumAliasNames("AliasNam.db")   ; create the table
  tvl.open("AliasNam.db")             ; view the table
else
  msgStop("Help!","Session variable is not Assigned!")
endif
endmethod
```

See also

- addAlias

enumDatabaseTables

Session

Method/Procedure

Creates a Paradox table listing the tables in a database.

Syntax

enumDataBaseTables (const ***tableName*** String, const ***databaseName*** String, const ***fileSpec*** String) Logical

Description

enumDataBaseTables creates the Paradox table *tableName* in a database *databaseName.* The table lists the tables in a database. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName;* if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is

Field Name	Type	Size
DBName	A	32*
TableName	A	32*

Example

This example lists the set of tables or other files associated with an alias.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dbName   String
  filespec String
  tv1      TableView
endvar

; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  dbName.view("Enter Database (Alias) Name")  ; prompt for alias
  filespec.view("Enter FileSpec")             ; prompt for filespec
  ses.enumDataBaseTables("TabList", dbName, filespec)
  tv1.open("Tablist")                         ; open the created table
else
  msgStop("Help!", "Session variable is not assigned!")
endif
endmethod
```

See also

❐ enumDataBaseCapabilities

enumDriverCapabilities — Session

Procedure

Creates three Paradox tables that list the capabilities of the current driver.

Syntax

enumDriverCapabilities (const ***drvCapName*** String, const ***tblCapName*** String, const ***fldCapName*** String)

Description

Creates three Paradox tables that list the capabilities of the current driver. The tables are overwritten (if they exist) without asking for confirmation. You can include an alias or path in the specified table names; if no alias or path is specified, Paradox creates the tables in the working directory.

Driver capabilities are written to the table *drvCapName* (each supported table type is described by a record), which has the following structure:

Field Name	Type	Size
DriverType	A	32*
Description	A	32
Category	A	32
DB	A	4
DBType	A	32
MultiUser	A	4
ReadWrite	A	4
Transactions	A	4
PassThruSQL	A	4
Login	A	4
CreateDb	A	4
DeleteDb	A	4
CreateTable	A	4
DeleteTable	A	4
MultiPasswords	A	4

Table capabilities are written to the table *tblCapName* (each supported table type is described by a record), which has the following structure:

Field Name	Type	Size
DriverType	A	32*
TableType	A	32*
Format	A	32*
ReadWrite	A	4
Create	A	4
Restructure	A	4
ValChecks	A	4
Security	A	4
RefInt	A	4
PrimaryKey	A	4
Indexing	A	4
NoFieldType	A	6
MaxRecSize	A	6
MaxFlds	A	6

Field capabilities are written to the table *fldCapName*, which has the following structure:

Field Name	Type	Size
DriverType	A	32*
TableType	A	32*
Format	A	32*

FieldType	A	32*
Description	A	32
NativeType	A	6
XType	A	6
XSubType	A	6
MaxUnits1	A	6
MaxUnits2	A	6
Size	A	6
Required	A	4
Default	A	4
Min	A	4
Max	A	4
RefInt	A	4
Other	A	4
Key	A	4
Multi	A	4

Example

In this example, the *describeDriver* button creates and views three tables that describe the engine driver.

```
; describeDriver::pushButton
method pushButton(var eventInfo Event)
var
  tv1, tv2, tv3  TableView
endVar
enumDriverCapabilities("dbcap", "tblcap", "fldcap")
tv1.open("dbcap")
tv2.open("tblcap")
tv3.open("fldcap")
endmethod
```

See also

- enumDriverInfo, enumDriverNames, enumDriverTopics

enumDriverInfo

Session

Procedure

Lists the available drivers.

Syntax

enumDriverInfo (const ***tableName*** String) Logical

Description

Lists the driver types currently available. Driver types are written to the table *tableName*. If *tableName* exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is

Field Name	Type	Size
DriverType	A	32*
Topic	A	32*
Property	A	32*
PropertyValue	A	68

Example

This example enumerates driver information to a table named *DriveInf*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tvl TableView
endVar
; create and view the DriveInf table
enumDriverInfo("Driveinf")
tvl.open("DriveInf")
endmethod
```

See also

- enumDriverCapabilities, enumDriverNames, enumDriverTopics

enumDriverNames

Session

Procedure

Creates a Paradox table listing the names of the drivers available in the current session.

Syntax

enumDriverNames (const ***tableName*** String) Logical

Description

Writes the driver names currently available to the table *tableName*. If *tableName* already exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is DriverType, A32*.

Example

This example enumerates available driver names to a table named *DrivName*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tvl  TableView
endVar
; create and view the DrivName table
enumDriverNames("DrivName")
```

```
tv1.open("DrivName")
endmethod
```

See also

❒ enumDriverCapabilities, enumDriverInfo, enumDriverTopics

enumDriverTopics

Session

Procedure

Creates a Paradox table listing the topics currently available for each driver type.

Syntax

enumDriverTopics (const ***tableName*** String) Logical

Description

Writes the driver topics available for each driver type to the table *tableName*. If *tableName* already exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName;* if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is:

Field Name	Type	Size
DriverType	A	32*
Topic	A	32*

Example

This example enumerates available driver topics to a table named *DrivTop*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1  TableView
endVar
; create and view the DrivTop table
enumDriverTopics("drivtop")
tv1.open("drivtop")
endmethod
```

See also

❒ enumDriverCapabilities, enumDriverInfo, enumDriverNames

enumEngineInfo

Session

Procedure

Creates Paradox table listing the current ODAPI engine properties.

Syntax

enumEngineInfo (const ***tableName*** String) Logical

Description

Creates a Paradox table that describes the contents of the ODAPI System Information dialog box. Each setting name and value is written to a record in the table *tableName*. If *tableName* already exists, it is overwritten without asking confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is:

Field Name	Type	Size
Property	A	32*
PropertyValue	A	68

Example

This example enumerates engine information to a table named *EngInf*, then views the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tvl  TableView
endvar
enumEngineInfo("EngInf")
tvl.open("EngInf")
endmethod
```

See also

- enumDriverInfo

---

enumFolder

Session

Procedure

Creates a Paradox table or array listing files in a folder.

Syntax

1. **enumFolder** (const ***tableName*** String
   [, const **fileSpec** String]) Logical
2. **enumFolder** (var ***result*** Array[] String
   [, const **fileSpec** String]) Logical

Description

Creates the Paradox table *tableName*, or array *result* listing the files in a session's folder. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

Optionally, you can specify a *fileSpec* to list files with a particular extension. For instance, to list all forms in a file, include a *fileSpec* of ".FSL".

The structure of the table is:

Field Name	Type	Size
Name	A	128
LocalName	A	68
IsReference	A	4
IsPrivate	A	4
IsTemp	A	4
Position	A	10

Example

In this example, the method prompts the user to enter a file specification (such as "*.FSL"). The file specification entered is then used by **enumFolder** to create a table listing the files that matched the specification.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  filespec String
  tvl      TableView
endvar
filespec.view("Enter file name specification")
enumFolder("PartCat", filespec)
message("Table lists files that matched your specification.")
tvl.open("PartCat")
endmethod
```

See also

- enumDatabaseTables

enumOpenDatabases

Session

Method/Procedure

Creates a Paradox table listing the open databases.

Syntax

enumOpenDatabases (const ***tableName*** String) Logical

Description

Creates the Paradox table *tableName* listing the databases open in the current session. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is:

Field Name	Type	Size
DBName	A	32*
DBType	A	32
ShareMode	A	32
OpenMode	A	32

See also

- enumDatabaseTables

enumUsers

Session

Procedure

Creates a Paradox table listing all known users with an open channel to the ODAPI engine.

Syntax

enumUsers (const ***tablename*** String) LongInt

Description

Creates the table *tableName* that lists all users with an open path to ODAPI. If *tableName* exists, it is overwritten without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of the table is:

Field Name	Type	Size
UserName	A	15
Net Session	N	
Product Class	N	
Serial Number	A	22

Example

This example writes information about current users to the *Users* table, then displays the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tv1 TableView
endVar
  enumUsers("users")
  tv1.open("users")
endmethod
```

See also

- getUserName

getAliasPath

Session

Method/Procedure

Returns the path for a specified alias.

Syntax

getAliasPath (const ***aliasName*** String) String

Description

Returns the path for the alias *aliasName*.

Example

This example prompts the user for an alias name, then shows the path currently associated with that alias.

```
; getShowPath::pushButton
method pushButton(var eventInfo Event)
var
  alname string
  alpath string
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  alname.view("Enter Alias Name")    ; prompt for an alias name
  alpath = ses.getAliasPath(alname)  ; get the path
  alpath.view("The Answer is...")    ; show the path
else
  msgStop("Help!","Session variable is not assigned!")
endif
endmethod
```

See also

- addAlias, setAliasPath

Session

getNetUserName

Session

Method/Procedure Returns the network user name for a session.

Syntax **getNetUserName** **( )** String

Description Returns the name of the current network user.

Example This example displays the current user's network name in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
msgInfo("Who Am I?", getNetUserName())
endmethod
```

See also

- enumUsers

ignoreCaseInLocate

Session

Procedure Specifies whether to ignore case in locate operations.

Syntax **ignoreCaseInLocate** **([** const ***yesNo*** Logical **])**

Description

Specifies whether the current session should ignore case-sensitivity during locate operations. If optional argument *yesNo* is Yes, this subsequent locate operations will ignore case in string comparisons; if *yesNo* is No, locate operations will be case-sensitive. If omitted, *yesNo* is Yes by default.

Example

This example calls **ignoreCaseInLocate**, if necessary, to set up for a call to the **locate** method.

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Customer.db") then
  ; if case-sensitivity is turned off
  if isIgnoreCaseInLocate() then

    ; locate values based on value as entered
    ; (do not ignore case in string compares)
    ignoreCaseInLocate()
  endif

  ; search for case-sensitive MacAnaly in Name field
  if tc.locate("Name", "MacAnaly") then
    tc.edit()
    tc.Name = "Macanaly"
    tc.endEdit()
  else
    message("Couldn't find MacAnaly...")
  endif
else
  msgStop("Error", "Can't open Customer table.")
endif

endmethod
```

See also

- isIgnoreCaseInLocate

isAdvancedWildcardsInLocate

Session

Procedure

Reports whether this session is using advanced wildcards in locate operations.

Syntax

isAdvancedWildcardsInLocate ( ) Logical

Description

Reports whether the current session is using advanced wildcards during locate operations that include pattern strings.

Example

This example calls **advancedWildcardsInLocate**, if necessary, to specify that advanced wild cards can be used in a locate operation.

Then the code continues with a call to **locatePattern** that uses an advanced wildcard pattern.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  thisSession Session
endvar

if tc.open("Orders.db") then

  ; if advanced wildcards can't be used in patterns
  if NOT isAdvancedWildcardsInLocate() then
    ; specify that this session can use advanced
    ; pattern characters in subsequent locate operations
    advancedWildcardsInLocate(Yes)
  endif

  if tc.locatePattern("Ship VIA", "[^UPS]") then
    msgInfo("Order Number", tc."Order No")
  else
    msgStop("Error", "Can't find record")
  endif
else
  msgStop("Error", "Can't open Orders table.")
endif

endmethod
```

See also

- advancedWildcardsInLocate

isAssigned

Session

Method

Reports whether a session variable is assigned.

Syntax

isAssigned ( ) Logical

Description

Reports whether a Session variable is assigned.

Example

See the example for close.

See also

- open, close

isBlankZero

Session

Method/Procedure

Reports whether blank values are being treated as zero in calculations.

Syntax

isBlankZero ( ) Logical

Description

Returns True if blank fields are treated as fields with a value of zero in calculations, or are counted as filled fields in counting calculation (for example, **cCount**). If blank fields are treated as blanks or are being ignored in calculations and counts, **isBlankZero** returns False. Use **blankAsZero** to change this setting.

Example

See the example for blankAsZero.

See also

- blankAsZero

isIgnoreCaseInLocate

Session

Procedure

Reports whether the current session is ignoring case in locate operations.

Syntax

isIgnoreCaseInLocate ( ) Logical

Description

Reports whether the current session is ignoring case-sensitivity during locate operations.

Example

See the example for ignoreCaseInLocate.

See also

- ignoreCaseInLocate

lock

Session

Procedure

Locks one or more tables.

Syntax

lock (const ***table*** { Table | TCursor | String}, const ***lockType*** String [, const ***table*** { Table | TCursor | String },
const ***lockType*** String]*) Logical

Description

Locks one or more tables specified in a comma-separated pairs of tables and lock types. You can use a TCursor or a Table to specify a table, and you can mix TCursor and Table variables in the list.

lockType must be a string expression that evaluates to one of the following values: Write, Read, and Full. ("Read" and "Full" apply only to Paradox tables.)

If this method locks all the tables in the list, it returns True; otherwise, it returns False. If it can't lock all the tables, it doesn't lock any.

Example

This example attempts to place a write lock on the *Orders* table and a full lock on the *Customer* table. If **lock** is able to lock both tables, the code displays data from both of the tables in a dialog box. Then, the code calls **unlock** to remove the explicit locks placed on *Customer* and *Orders*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ordTB    Table
  custTC   TCursor
  sampDB   Database
endvar

otherSes.open()
otherSes.addAlias("samples", "Standard", "c:\\pdoxwin\\sample")
sampDB.open("samples")

custTC.open("Customer.db", "samples")
ordTB.attach("Orders.db", "samples")

if lock(custTC, "Read", ordTB, "Write") then
  if custTC.locate("Name", "Unisco") then
    custNo = custTC."Customer No"
    ordTB.setFilter(custNo, custNo)
    msgInfo(String("Total for order ", custNo), ordTB.cSum("Total Invoice"))
    unlock(custTC, "Read", ordTB, "Write")
  else
    msgStop("Error", "Can't find Unisco.")
  endif
else
  msgStop("Error", "Can't lock one or more tables.")
endif

endmethod
```

See also

- unlock

---

open

Session

Method

Opens a session.

Syntax

open ([const ***sessionName*** String]) Logical

Description

Opens a channel to the database engine (a session), and exhausts one user count. You can open more than one session from the same workstation, and Paradox will view each session as a separate user; for example, locks set in one session block access from the other.

Example

The following code opens a new session to the variable *ses*.

```
; openSession::pushButton
method pushButton(var eventInfo Event)
var
  ses Session
endVar
if ses.open() then
  msgInfo("We Have Lift-Off","Session open successful.")
else
  msgStop("Crash and Burn","Session open unsuccessful.")
endif
endmethod
```

See also

- close

---

removeAlias

Session

Method/Procedure

Removes an alias from a session.

Syntax

removeAlias (const ***aliasName*** String) **Logical**

Description

Removes the alias *alias* from a session.

Example

The following example adds an alias to the current session, then supplies the new alias to the **open** method defined for the Database type. When the alias is no longer needed, this code calls **removeAlias** to remove the alias name from the current session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custInfo Database
endVar

; add the CustomerInfo alias to the current session
addAlias("CustomerInfo", "Standard", "D:\\pdoxwin\\tables\\custdata")

; now use the alias specify the database to open
custInfo.open("CustomerInfo")      ; opens the CustomerInfo database

; do something with the opened database,
; then when the alias is no longer needed,
; remove the alias from the current session

removeAlias("CustomerInfo")

endmethod
```

See also

- addAlias

removeAllPasswords

Session

Method/Procedure

Removes all passwords presented to a session.

Syntax

removeAllPasswords ( )

Description

Reverses the effects of all **password** statements issued for a session. It does not remove security from tables; it withdraws the passwords required to access protected tables. Open tables are not affected by **removeAllPasswords**. To remove access to open tables, you must both close the table and withdraw the passwords that provide access to the tables.

Example

This example removes all the passwords from the session *ses*.

```
; removePasses::pushButton
method pushButton(var eventInfo Event)
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  ses.removeAllPasswords()
else
  msgStop("Help!","Session variable is not Assigned!")
endif
endmethod
```

See also

- password, removePassword

removePassword

Session

Method/Procedure

Removes a password presented to a session.

Syntax

removePassword (const ***password*** String)

Description

Reverses the effect of a **password** statement issued for a session. It does not unprotect the table; it merely withdraws the password specified in the argument *password* that was presented to access the table. Note that *password* is case-sensitive.

Example

In this example, the *getRemovePass* button acquires a password to remove from the user, then removes the password from the current session. Subsequent attempts to open tables protected by that password will fail.

```
; getRemovePass::pushButton
method pushButton(var eventInfo Event)
```

```
var
  newPass string
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  newPass.view("Enter Password to Remove")
  ses.removePassword(newPass)
else
  msgStop("Help!", "Session variable is not Assigned!")
endif
endmethod
```

See also

- password, removeAllPasswords

retryPeriod

Session

Method/Procedure

Returns the number of seconds to retry an operation on a locked record or table.

Syntax

retryPeriod ( ) SmallInt

Description

Returns the number of seconds to retry an operation on a locked record or table. The default value is 0, which means that operations are not retried.

Example

The following example displays the current retry period to the user.

```
; getShowRetry::pushButton
method pushButton(var eventInfo Event)
var
  rp smallint
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  rp = ses.RetryPeriod()           ; get the current retry period
  rp.view("The Retry Period is...") ; display the value
else
  msgStop("Help!","Session variable is not assigned!")
endif
endmethod
```

See also

- setRetryPeriod

saveCFG

Session

Method/Procedure

Saves the current session's alias information to a file.

Syntax

saveCfg (const ***fileName*** String) Logical

Description

Saves the ODAPI configuration for the current session to *fileName*. The configuration file specified by *fileName* can be loaded (with the **-o** command-line option) in place of ODAPI.CFG to set session information when you start Paradox for Windows.

setAliasPath

Session

Method/Procedure

Sets the path for an alias.

Syntax

setAliasPath (const ***aliasName*** String,
const ***aliasPath*** String) Logical

Description

Sets the path *aliasPath* for the alias *aliasName*.

See also

- getAliasPath

setRetryPeriod

Session

Method/Procedure

Sets the number of seconds to retry an action on a locked table or record.

Syntax

setRetryPeriod (const ***period*** SmallInt) Logical

Description

Specifies in *period* the number of seconds to retry an action on a locked table or record. A value of 0 means that actions are not retried.

Example

This example prompts the user for a retry period, then sets the retry for the session to that value.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  rp SmallInt
endvar
; assume that the variable ses is global, and has been
; opened by another method
if ses.isAssigned() then
  rp = ses.retryPeriod()
  rp.view("Enter retry period")  ; get a retry period from user
  ses.setRetryPeriod(rp)         ; set the session's retry period
else
  msgStop("Help!","Session variable is not assigned!")
```

```
endif
endmethod
```

See also

- retryPeriod

unlock

Session

Procedure

Unlocks one or more tables.

Syntax

unlock (const ***table*** { Table | TCursor| String}, const ***lockType*** String [, const ***table*** { Table | TCursor| String }, const ***lockType*** String]*) Logical

Description

Unlocks one or more tables specified in a comma-separated list of tables and lock types.

unlock removes locks explicitly placed by a particular user or application using **lock**; it has no effect on locks placed automatically by Paradox. *lockType* must be a string expression that evaluates to one of the following values: Write, Read, and Full. "Read" and "Full" apply only to Paradox tables.)

If one **unlock** in the list fails, previous locks are not restored; the tables remain unlocked. You don't have to specify a session to use this method, because session data is set when you **open** a TCursor or **attach** to a Table.

Each time you lock a table explicitly, make sure to unlock it as soon as you no longer need the explicit lock. This ensures maximum concurrent availability of tables. Also, when you lock a table twice, you must unlock it twice. You can use the **lockStatus** method (defined for the TCursor and UIObject type to determine how many explicit locks you have placed on a table. **unlock** returns False if you try to unlock a table that isn't locked or cannot be unlocked.

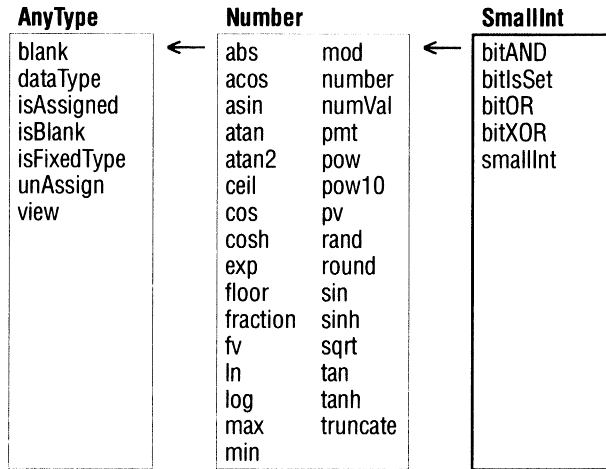
Example

See the example for lock.

See also

- lock

SmallInt



SmallInt values are small integers; that is, they can be represented by a small (short) series of digits. A SmallInt variable occupies 2 bytes of storage.

ObjectPAL converts SmallInt values to range from -32,768 to 32,767. An attempt to assign a value outside of this range to a SmallInt variable causes an error. For example,

```
var
   x, y, z SmallInt
endVar

x = 32767 ; the upper limit value for a SmallInt variable
y = 1
z = x + y ; causes an error
```

To work with boundary values, store the result in a variable of a type that can accommodate it. For example,

```
var
   x, y SmallInt
   z    LongInt   ; declare z as a LongInt so it can hold the result
endVar

x = 32767 ; the upper limit value for a SmallInt variable
y = 1
z = x + y

z.view() ; displays 32768 because z is a LongInt
         ; and can handle the large value
```

Note The SmallInt value -32,768 cannot be stored in a Paradox table because, to Paradox, -32,768 = Blank. However, you can use this value in calculations, and you can store it in a dBASE table.

Note The SmallInt type includes methods defined for the AnyType type. Refer to the AnyType section for more information. Also, run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. The syntax is the same, and the returned value is a Number. For example, the following code will work, even though **sin** does not appear in the list of methods for the SmallInt type:

```
var
   abc LongInt
   xyz Number
endVar
abc = 43
xyz = abc.sin()
```

Note ObjectPAL supports an alternate syntax:

methodName (*objVar, argument* [,*argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return the sine of a number:

```
theNum.sin()
```

The following statement uses the alternate syntax:

```
sin(theNum)
```

We recommend using standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

bitAND

SmallInt

Method

Performs a bitwise AND operation on two values.

Syntax

bitAND (const ***value*** SmallInt) SmallInt

Description

Returns the result of a bitwise AND operation on *value*. **bitAND** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitAND** is:

a	b	a.bitAND(b)
0	0	0
1	0	0
0	1	0
1	1	1

Example

In the following example, the **pushButton** method for a button named *andTwoNums* takes two integers and performs a bitwise AND calculation on them. The result of the calculation is displayed in a dialog box.

```
; andTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233    ; binary 01110110 00011001
b = 1233     ; binary 00000100 11010001
a.bitAND(b) ; binary 00000100 00010001
msgInfo("The result of 30233 bitAND 1233 is:", a.bitAND(b))
; displays 1041
endmethod
```

See also

- bitOR, bitXOR

bitIsSet

SmallInt

Method

Reports whether a bit is 1 or 0.

Syntax

bitIsSet (const ***value*** SmallInt) Logical

Description

Examines the binary representation of an integer, reporting whether the **value** bit is 0 or 1. It returns True if the bit specified is 1, and False if the bit is 0.

value is a number specified by 2^n, where *n* is an integer between 0 and 14. The exponent *n* corresponds to one less than the position of the bit to test, counting from the right. For example, to specify the third bit from the right, use 4 ($2^{(3-1)}$, which is 2^2).

Example

In the following example, the **pushButton** method for a button named *isABitSet*, examines the values in two unbound field objects: *whichBit*, and *whatNum*. *whichBit* contains the bit position (counting from the right) of the bit test. *whatNum* contains the integer to test. The **pushButton** method uses *whichBit* to calculate the value of the position, then assigns the result to *bitNum*. The method then checks *Num* to see if the *bitNum* bit is set, and displays the Logical result with a **msgInfo** dialog box.

```
; isABitSet::pushButton
method pushButton(var eventInfo Event)
var
  bitNum,
  Num       SmallInt
endVar
; get the bit position number from the whichBit
```

```
; field and convert to multiple of 2
bitNum = SmallInt(pow(2, whichBit - 1))
; get the number to test from the whatNum field
Num = whatNum
; is the bit for value bitNum 1 in Num?
msgInfo("Is Bit Set?", Num.bitIsSet(bitNum))
endmethod
```

The next example illustrates how you can use **bitIsSet** to display an integer as a binary number. The **pushButton** method for *showBinary* constructs a string of zeros and ones by testing each bit of a four-byte long integer. For readability, a blank is added to the string after 8 digits.

```
; showBinary::pushButton
method pushButton(var eventInfo Event)
var
  binString  String    ; to construct the binary string
  Num        SmallInt  ; number to test
  i          SmallInt  ; for loop index
endVar
if NOT whatNum.isBlank() then
  Num = whatNum                 ; get the number test from whatNum
  binString = ""                ; initialize the string
  for i from 0 to 14
    if Num.bitIsSet(SmallInt(pow(2, i))) then
      binString = "1" + binString    ; add a 1 to the front of the string
    else
      binString = "0" + binString    ; add a 0 to the front of the string
    endif
    if i = 7 then
      binSTring = " " + binString    ; add a space every 8 digits
    endif
  endfor
  if Num < 0 then
    binString = "1" + binString      ; set the sign bit
  else
    binString = "0" + binString
  endif
  ; show the number
  message("The binary equivalent is ", binString)
endif
endmethod
```

See also

- bitAND, bitOR, bitXOR

---

bitOR

SmallInt

Method

Performs a bitwise OR operation on two values.

Syntax

bitOR (const ***value*** SmallInt) SmallInt

Description

Returns the result of a bitwise OR operation on *value*. **bitOR** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitOR** is:

a	b	a.bitOR(b)
0	0	0
1	0	1
0	1	1
1	1	1

Example

In the following example, the **pushButton** method for a button named *orTwoNums* takes two integers and performs a bitwise OR calculation on them. The result of the calculation is displayed in a dialog box.

```
; orTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233  ; binary 01110110 00011001
b = 1233   ; binary 00000100 11010001
a.bitOR(b) ; binary 01110110 11011001
msgInfo("30233 OR 1233", a.bitOR(b)) ; displays 30425
endmethod
```

See also

- bitAND, bitXOR

bitXOR

SmallInt

Method

Performs a bitwise XOR operation on two values.

Syntax

bitXOR (const ***value*** SmallInt) SmallInt

Description

Performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary representations of two integers, comparing them one bit at a time. The truth table for **bitXOR** is:

a	b	a.bitXOR(b)
0	0	0
1	0	1
0	1	1
1	1	0

Example

In this example, the **pushButton** method for a button named *xorTwoNums* takes two integers and performs a bitwise XOR calculation on them. The result of the calculation is displayed in a dialog box.

```
; xorTwoNums::pushButton
method pushButton(var eventInfo Event)
var
  a, b SmallInt
endVar
a = 30233   ; binary 01110110 00011001
b = 1233    ; binary 00000100 11010001
a.bitXOR(b) ; binary 01110010 11001000
msgInfo("30233 XOR 1233", a.bitXOR(b)) ; displays 29384
endmethod
```

See also

- bitAND, bitOR

---

int

SmallInt

Procedure

Casts a value as an integer.

Syntax

int (const ***value*** AnyType) SmallInt

Description

Casts (converts) the numeric expression *value* to an integer. If *value* is of a more precise type (for example, Number), precision is lost.

Example

The following example assigns a number to *nn*, views the value of *nn* in a dialog box, then displays *nn* as an integer. This code is attached to the **pushButton** method for the *showInt* button.

```
; showInt::pushButton
method pushButton(var eventInfo Event)
var
   nn Number
endVar
nn = 123.12
view(nn)                              ; displays 123.12
msgInfo("nn as Integer", int(nn))   ; displays 123
endmethod
```

See also

- smallInt

---

smallInt

Beginner

SmallInt

Procedure

Casts a value as a small integer.

Syntax

smallInt (const ***value*** AnyType) SmallInt

Description

Casts (converts) the numeric expression *value* to a SmallInt. If *value* is of a more precise type (for example, Number), precision is lost.

Example

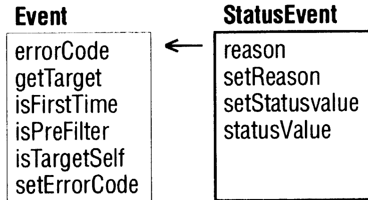
The following example assigns a number to *x*, then casts *x* to SmallInt and assigns the result to *s*. The decimal precision of *x* is lost when it is cast to a SmallInt.

```
; convertToInt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
  s SmallInt
endVar
x = 12.34             ; give x a value
x.view()              ; view x, title of dialog will be "Number"
s = SmallInt(x)       ; cast x as a LongInt and assign to s
s.view()              ; show s, note that decimal places are lost
                      ; displays 12
endmethod
```

See also

- int

StatusEvent



StatusEvent type methods control messages that appear in the Desktop status bar. The StatusEvent type includes several methods defined for the Event type.

Every design object has a built-in **status** method, which is triggered by a StatusEvent. This method, along with the rest of the built-in methods, is discussed in Chapter 2. For information about the Event model, see the *ObjectPAL Developer's Guide.*

Using StatusEvent type methods, you can attach code to the built-in method to find out where and why messages will be displayed. You can block messages or display them somewhere else, in a different status area, or in another object (for example, a field object or a text file). You can also specify the text to be displayed in the message.

You can use the StatusWindows constants ModeWindow1, ModeWindow2, ModeWindow3, and StatusWindow to refer to the areas of the status bar.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide.*

reason

StatusEvent

Method

Reports why a StatusEvent occurred.

Syntax

reason ( ) SmallInt

Description

Returns an integer value to report why a StatusEvent occurred. StatusEvent Reasons occur when a built-in **status** method is called. ObjectPAL provides the following constants for testing the value returned by **reason** (listed in the Constants dialog box under StatusReasons):

StatusWindow means the message was sent to the Status area (the largest area on the status bar, it occupies the left two-thirds or so of the status bar).

- ModeWindow1 means the message was sent to the first small window to the right of the Status area.
- ModeWindow2 means the message was sent to the second small window to the right of the Status area.
- ModeWindow3 means the message was sent to the third small window to the right of the Status area (the right-most window).

Example

In the following example, assume that a form contains two fields, *fieldOne* and *fieldTwo*. The **status** method for *fieldTwo* examines the event packet for an error code; if it finds one, the method displays a message in the status line. The **status** method for the form sets an error code if the event's target is *fieldTwo*. The following code is attached to the **status** method for *fieldTwo*.

```
; fieldTwo::status
method status(var eventInfo StatusEvent)
if eventInfo.errorCode() <> 0 then
  ; display a different message if there is an error
  eventInfo.setStatusValue("There was an error here.")
endif
endmethod
```

This code is attached to the **status** method for the form:

```
; thisForm::status
method status(var eventInfo StatusEvent)
var
  targObj   UIObject
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    eventInfo.getTarget(targObj)
    ; set an arbitrary error for fieldTwo
    if targObj.Name = "fieldTwo" AND
       eventInfo.reason() = StatusWindow then
      eventInfo.setErrorCode(1)
    endif
  else
    ; code here executes just for form itself

endif
endmethod
```

See also

- setReason

StatusEvent

setReason

StatusEvent

Method

Specifies a Reason for generating a StatusEvent.

Syntax

setReason (const ***reasonId*** SmallInt)

Description

Specifies a Reason constant for generating a StatusEvent. The StatusEvent reasons tell you which window on the status bar the message was sent to. ObjectPAL provides the following constants for setting the Reason for a StatusEvent (listed in the Constants dialog box under StatusReasons):

- StatusWindow means the message was sent to the Status area (the largest area on the status bar, it occupies the left two-thirds or so of the status bar).
- ModeWindow1 means the message was sent to the first small window to the right of the Status area.
- ModeWindow2 means the message was sent to the second small window to the right of the Status area.
- ModeWindow3 means the message was sent to the third small window to the right of the Status area (the right-most window).

Example

In this example, for StatusEvent bubbled up to the form from a field, the form's **status** method changes the reason and the content of the message. The method changes the reason to ModeWindow1, and sets the value of the message to the name of the object that started the original event (the target).

```
; thisForm::status
method status(var eventInfo StatusEvent)
var
  targObj  UIObject
  nameStr  String
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    ; after regular message has displayed, also show
    ; field name in ModeWindow1
    eventInfo.getTarget(targObj)
    if targObj.Class = "Field" then      ; if this is a field
      nameStr = targObj.Name       ; get the field name
      eventInfo.setReason(ModeWindow1)   ; set the window
      eventInfo.setStatusValue(nameStr)  ; send the string
    endif
endif
endmethod
```

See also

- reason
- errorCode, setErrorCode in the Event type

setStatusValue

StatusEvent

Method

Specifes the text of a status message.

Syntax

setStatusValue (const ***statusValue*** AnyType)

Description

Specifies the text of a status message in *messageText*.

Example

See the example for setReason.

See also

- setReason, statusValue

statusValue

StatusEvent

Method

Returns the text of a status message.

Syntax

statusValue ( ) AnyType

Description

Returns the text of a status message.

Example

This example makes the default status messages more prominent to the user by copying each message to a field on the form. This feature is controlled by the *magnifyMessage* button, also on the same form. The following code is attached to the **pushButton** method of the *magnifyMessage* button:

```
; magnifyMessage::pushButton
method pushButton(var eventInfo Event)
; toggle statusMessageField to visible or invisible and
; toggle label between "Magnified Messages" and "Normal Messages"
if self.LabelText = "Magnified Messages" then
  statusMessageField.Visible = True
  self.LabelText = "Normal Messages"
else
  statusMessageField.Visible = False
  self.LabelText = "Magnified Messages"
endif
endmethod
```

This code is attached to the form's **status** method:

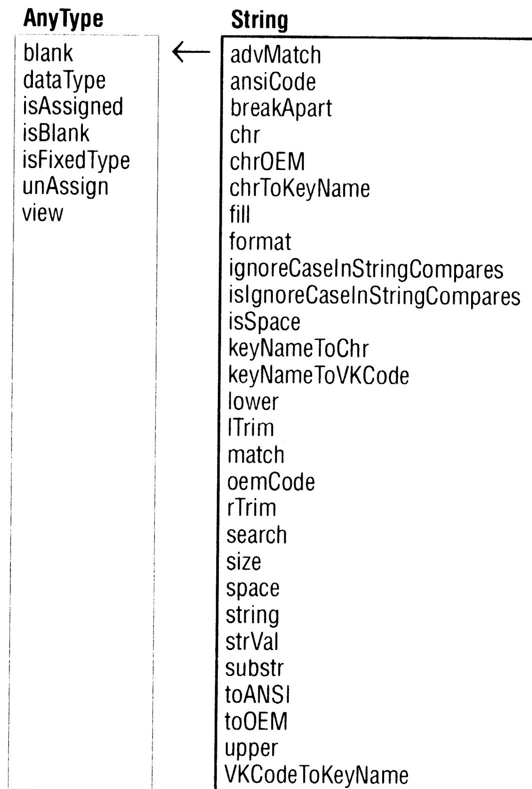
```
; thisForm::status
method status(var eventInfo StatusEvent)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    ; write every status event to a field on the form
    if statusMessageField.Visible = True then
      if eventInfo.reason() = StatusWindow then
```

```
        statusMessageField = eventInfo.statusValue()
      endif
    endif
  else
    ; code here executes just for form itself

endif
endmethod
```

See also

- reason, setStatusValue

String



A String variable can contain up to 32,767 characters (use Memo objects for longer text). A quoted string can contain up to 255 characters. Use double quotes ("") to represent an empty string. Strings occupy 1 byte of storage per character.

For more information and examples, refer to the "String" section of Chapter 9 in the *ObjectPAL Developer's Guide.*

The String type also includes methods defined for the AnyType type.

Note ObjectPAL supports an alternate syntax:

methodName (*objVar, argument* [*,argument*])

methodName represents the name of the method, *objVar* is the variable representing an object, and *argument* represents one or more arguments. For example, the following statement uses the standard ObjectPAL syntax to return a lowercase version of a string:

```
theString.lower()
```

The following statement uses the alternate syntax:

```
lower(theString)
```

It's best to use standard syntax for clarity and consistency, but you can use the alternate syntax wherever it's convenient.

advMatch

String

Beginner

Method

Searches text for a specified string.

Syntax

advMatch (const ***pattern*** String [,var ***matchVar*** String]*) Logical

Description

Returns True if *pattern* is found within the string; otherwise, it returns False. This method is case-sensitive. To specify *pattern*, use a string and the optional symbols listed in the table.

matchVar is a variable to which the matching portion will be assigned. **advMatch** assigns matched patterns to one or more *matchVar* variables as the patterns are found. The portions of the string matching wildcard elements are assigned to the variables from left to right. Since there may be multiple matches, the first matching substring is assigned to the first variable, the second matching substring to the second variable, and so on. If no match is found, variables are not assigned values.

If you supply *pattern* from within a method, you need to use two backslashes when you want to tell **advMatch** to treat a special character as a literal; for example, \\(tells **advMatch** to treat the parenthesis as a literal character. Two backslashes are required in this situation because of the ambiguity between the compiler's interpretation of a backslash (used in escape sequences such as \t for a tab) and **advMatch**'s understanding of a backslash. When the compiler sees a string with an embedded escape sequence, such as a "\tstart", it interprets the "\t" as a tab. The backslash character has a special meaning to the compiler, but it also has a special meaning to **advMatch**.

For example, if you're trying to search for a question mark embedded in a string, you might call **advMatch** like so:

```
s = "a string?"
advMatch(s, "\?")        ; this won't work!
```

You might think that you're telling **advMatch** to search for the literal question mark. However, the compiler sees the string first and returns a syntax error because "\?" is not a valid escape sequence. To prevent the compiler from interpreting the backslash as the beginning

of an escape sequence, precede the backslash by another backslash. This will work:

```
s = "a string?"
advMatch(s, "\\?")        ; this does work!
```

If you supply *pattern* from a field in a table or a TextStream, special **advMatch** symbols are recognized without a preceding backslash, and one backslash and plus symbol (\+) yields a literal character.

Symbol	Matches
\	Use backslash to include one of the following special characters as a regular character. (Remember to use two backslashes in quoted strings.)
[ ]	Match the enclosed set. For instance, [aeiou0–9] match a, e, i, o, u, and 0 through 9.
[^]	Do *not* match the enclosed set. For instance, [^aeiou0–9] match anything except a, e, i, o, u, and 0 through 9.
()	Grouping.
^	Beginning of line (do not confuse this with [^], where the ^^ acts as a logical NOT).
$	End of string.
..	Match anything.
@	Match any single character.
*	Zero or more of the preceding character or expression.
+	One or more of the preceding character or expression.
?	None or one of the preceding character or expression.
|	OR operation.

String

Example

These statements demonstrate **advMatch** functionality:

```
method pushButton(var eventInfo Event)
var
  w, x, y, z      String
  l               Logical
endVar

l = advMatch("this is", "s")
l.view()
  ; returns True (different from match)

l = advMatch("this is", "^s")
l.view()
  ; returns False, because it requires s to be at the beginning of the line

l = advMatch("this is", "S")
l.view()
  ; returns False, it is case sensitive.

l = advMatch("this is", "[sS]")
l.view()
  ; returns True, because [sS] specifies any in this set

l = advMatch("this is", "[a-z]")
```

```
l.view()
  ; returns True, because [a-z] specifies any in this set of a through z

l = advMatch("this is", "[a-c]")
l.view()
  ; returns False, because [a-c] specifies any in this set of a through c
  ; and "this is" does not contain a, b, or c

l = advMatch("this is", "[a-cs]")
l.view()
  ; returns True, because [a-cc] specifies any in this set of a through c
  ; or s and "this is" does contain s
  ; note that [a-c, s] would specify any in the set of a through c,
  ; a comma, a space, or an s

l = advMatch("this is", "(@)s", x)
l.view()
x.view()
  ; returns True, x = "i" because the "()" operators specify a group,
  ; unlike match, advMatch places only those things that you group
  ; in the variables

l = advMatch("this is a test", "((t@@s)|(t@s))|(@s)", w, x, y, z)
l.view()    ; returns True, and
w.view()    ; "this", the result of the first set of parentheses,
            ; that is, for the entire expression ((t@@s)|(t@s))
            ; also, "this" was matched before "test"
x.view()    ; also "this", for the result of the second set of
            ; parentheses, (t@@s)
y.view()    ; the result of (t@s), blank, because the t@@s
            ; satisfied the expression ((t@@s)|(t@s))
z.view()    ; also blank, because the expression ((t@@s)|(t@s)) satisfied
            ; the entire pattern ((t@@s)|(t@s))|(@s)
; NOTE: Match variables are matched to groups in the order of occurrence,
;       not in the order of precedence: The first group--starting from
;       the left--is assigned to the first variable.

l = advMatch("this is so", "(..)is(..)", x, y)
l.view()
x.view()
y.view()
  ; returns True, x = "this", y = " so"

l = advMatch("this is so", "[a-c]|[f-l]s" )
l.view()
  ; returns True, because an s is preceded by either a through
  ; c or f through l

l = advMatch("this as so", "[a-c]|[t-z]s" )
l.view()
  ; returns True, because an s is preceded by either a through
  ; c or t through z

endmethod
```

See also

- match, search

ansiCode

String

Procedure

Returns the ANSI code of a one-character string.

Syntax

ansiCode (const ***char*** String) SmallInt

Description

Returns the ANSI code of *char*. The ANSI code returned is an integer between 1 and 255.

Example

In the following example, assume a form contains four field objects: *showAllChars*, *ANSIField*, *OEMField*, and *KeyNameField*. The **keyPhysical** method for *showAllChars* examines every character, then translates it to its ANSI code, OEM code, and key-name equivalent. The various character codes are written to *ANSIField*, *OEMField*, and *KeyNameField*.

```
; showAllChars::keyPhysical
method keyPhysical(var eventInfo KeyEvent)
var
  anyChar   String
  anyANSI   SmallInt
  anyKeyN   String
  anyOEM    SmallInt
endVar
anyChar = eventInfo.char()          ; get the character typed
anyANSI = ansiCode(anyChar)         ; convert to ANSI code
ANSIField = anyANSI                 ; write ANSI code to ANSIField

anyCode = eventInfo.vCharCode()     ; get the VK_Code of character

anyKeyN = VKCodeToKeyName(anyCode)  ; convert VK_Code to key name
KeyNameField = anyKeyN              ; write key name to KeyNameField

anyOEM = oemCode(anyChar)           ; convert char to OEM code
OEMField = anyOEM                   ; write OEM code to OEMField
beep()
endmethod
```

See also

- chr, chrToKeyName

---

breakApart

Beginner

String

Method

Splits a string into substrings.

Syntax

breakApart (var ***tokenArray*** Array[] String [,const ***separators*** String])

Description

Splits a string into an array of substrings; each substring is written to an element of the array *tokenArray*. You can specify one or more delimiting characters in *separators*. If you omit *separators*, substrings are delimited by a space. In either case, delimiting characters are not included in *tokenArray*. This method is useful for importing data from a text file into a table.

Note Two delimiters with nothing in between parse as a token and result in an empty array element.

Example

In the following example, the **pushButton** method for a button named *breakToArray* creates three arrays from the same string. The first time, the call to the **breakApart** method does not specify any delimiters; by default, the method treats spaces as delimiters. The second time, the call to **breakApart** specifies the asterisk as a delimiter. Empty array elements are created each time an asterisk immediately follows another asterisk. The third time, the question mark, comma, and semicolon are listed as delimiters. The space is not used as a delimiter.

```
; breakToArray::pushButton
method pushButton(var eventInfo Event)
var
  ar Array[] String ; Must be resizable
  s String
endvar

s = "this is, a : delimited ? string"

s.breakApart(ar) ; breaks on spaces by default
ar.view()
{
ar = this
     is,
     a
     :
     delimited
     ?
     string
}

s = "this*is*a*delimited**string"
s.breakApart(ar, "*") ; breaks on specified characters
ar.view()
{
ar = this
     is
     a
     delimited

     string
}

s = "this is, a : delimited ? string"
s.breakApart(ar, ",:?") ; breaks on specified characters
                        ; this time, no space in list of delimiters
ar.view()
{
ar = this is
      a
      delimited
      string
}

endmethod
```

See also

- substr

chr

String

Beginner

Procedure

Returns the one-character string represented by an ANSI code.

Syntax

chr (const ***ansiCode*** SmallInt) String

Description

Returns a one-character string containing the ANSI character corresponding to *ansiCode*. If *ansiCode* is not an integer between 1 and 255, an error results.

You can use **chr** to generate characters that are not easily accessible through the keyboard.

Example

In the following example, the **pushButton** method for a button named *showChar* assigns the ANSI character 167 to the *sectionChar* variable, converts character 167 to its key name, and assigns it to *sectionKeyName*. The method then displays both versions of the character in a dialog box.

```
; showChar::pushButton
method pushButton(var eventInfo Event)
var
  sectionChar    String
  sectionKeyName String
endVar
sectionChar = chr(167)                           ; get the character
sectionKeyName = chrToKeyName(chr(167))          ; get the key name
msgInfo("The section character", sectionChar +   ; show the character and
       " has a key name of " + sectionKeyName)   ; the key name
endmethod
```

See also

- chrOEM, chrToKeyName, string

chrOEM

String

Procedure

Returns the one-character string of an OEM code.

Syntax

chrOEM (const ***oemCode*** SmallInt) String

Description

Returns a one-character string containing the OEM character corresponding to *oemCode*. If *oemCode* is not an integer between 1 and 255, an error results.

You can use **chrOEM** to generate characters that are not easily accessible through the keyboard. See Chapter 15 in the *ObjectPAL Developer's Guide* for more information.

Example

In the following example, a form has a button named *showOEM* and a field named *fieldOne*. The **pushButton** method for *showOEM* displays the OEM character specified by the number in *fieldOne*.

```
; showOEM::pushButton
method pushButton(var eventInfo Event)
msgInfo("OEM char described by fieldOne", chrOEM(fieldOne))
endmethod
```

See also

- chr, string

chrToKeyName

String

Procedure

Returns the virtual key-code string of a one-character string.

Syntax

chrToKeyName (const ***char*** String) String

Description

Returns the virtual key code of *char* as a string. A key name is one of the virtual key codes (such as VK_BACK for Backspace), but is returned as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*. ObjectPAL provides constants for virtual key codes; see Keyboard in the Constants dialog box.

Example

See the example for chr.

See also

- VKCodeToKeyName

fill

Beginner

String

Procedure

Returns a string containing repeated instances of a character.

Syntax

fill (const ***fillCharacter*** String, const ***fillNumber*** SmallInt) String

Description

Returns a string containing the first character in *fillCharacter* (usually a one-character string), where *fillCharacter* is repeated the number of times specified in *fillNumber*. *fillNumber* must be a non-negative integer; if *fillNumber* is 0, **fill** returns an empty string.

Example

In this example, the **pushButton** method for the *fillAndView* button creates two strings with the **fill** procedure. The method creates the

first string by filling a variable with the same letter five times. The second string is created by repeating the string "Quattro Pro" four times.

```
; fillAndView::pushButton
method pushButton(var eventInfo Event)
var
  str String
endVar
str = fill("X", 5)
str.View()                    ; displays XXXXX
str = fill("Quattro Pro
", 4)          ; add a line break after every occurrence
str.View()
; displays:    Quattro Pro
;              Quattro Pro
;              Quattro Pro
;              Quattro Pro
endmethod
```

See also

- space

format

Beginner

String

Procedure

Returns a formatted string for display or printing.

Syntax

format (const ***formatSpec*** String, const ***value*** AnyType) String

Description

Lets you control the way values are displayed or printed. *formatSpec* is a string expression containing one or more format specifications to be applied to String.

The following table lists the default format specification for each format category. This table also lists the valid data types for each format category. In addition to the data types listed, you can use AnyType values, as long as the value can be interpreted consistently with the format category.

Format Category	Meaning	Data Types Allowed	Default
Width	Set allowable field width and decimal precision	All	Entire data value
Alignment	Alignment within width	All	AR (right-justified) for all numeric types,AL (left-justified) for all others (including point)
Case	Uppercase or lowercase strings	All string types	No default

Format Category	Meaning	Data Types Allowed	Default
Edit	Specify characters and spacing	All numeric types	See following defaults
	Include a specified symbol		No default
	Decimal point character		ED. (period as decimal point)
	Whole number separator		No separator
	Number of leading zeros		None
	Symbol spacing		None
	Scientific notation		No
	Hide trailing spaces		No (show spaces)
	Use zeros as fill pattern		No
	Scale numbers up		No
	Precede with dollar sign		No
	American or International separators		American
Sign	Format of positive and negative numbers	All numeric types	See following
	Positive		No leading positive sign 999
	Negative		Leading minus sign -999
Date	Specify date formats	Date and DateTime	mm/dd/yy(yy) for Date or hh:mm:ss am(pm), mm/dd/yy(yy) for DateTime
Time	Specify time formats	Time and DateTime	hh:mm:ss am(pm) for Date or hh:mm:ss am(pm), mm/dd/yy(yy) for DateTime
Logical	Logical value representation	Logical	True/False

You can combine two or more format specifications in *formatSpec* by separating them with commas. The following table shows the format specifications you can use.

Type	Spec	Meaning
Width	W*n*	*n* specifies total format width,including all special characters, leading symbols or spaces, decimal point, and whole number separators
	W.*n*	*n* specifies number of decimal places, so W12.2 specifies a field of 12 characters, two of which are after the decimal character
	W.W	Use decimal places from Windows numbers

Type	Spec	Meaning
	W.$	Use decimal places from Windows currency
Alignment	AL	Left align in field
	AR	Right align in field
	AC	Center in field
Case	CU	Convert to uppercase
	CL	Convert to lowercase
	CC	Convert to initial capitals
Edit	E(*s*)	*s* specifies symbol to precede number
	E$W	Include currency symbol from Windows
	ED*d*	*d* specifies decimal point character
	EDW	Use Windows decimal-point character
	EN*c*	*c* specifies whole-number separator
	ENW	Use Windows whole-number separator
	EL*n*	*n* specifies the number of leading zeros
	ELW	Use Windows leading zero setting
	EP0	No symbol spacing
	EP-	Make symbol spacing for negatives
	EP+	Make symbol spacing for positives
	EPB	Make symbol spacing for all numbers
	EPW	Use Windows symbol spacing setting
	ES	Use scientific notation
	ET	Hide trailing spaces
	EZ	Use zeros as fill pattern
	EB	Use blanks as fill pattern
	E*	Use '*' as fill pattern
	E+n	Scale the number up
	E-n	Scale the number down
	E$	The same as E($)
	EC	The same as EN (or EN.D
	EI	The same as ED (or ED,N. if EC is set)
Sign	S+0	Format positives as $999
	S+1	Format positives as +$999
	S+2	Format positives as $+999
	S+3	Format positives as $999+
	S+4	Format positives as 999$
	S+5	Format positives as +999$
	S+6	Format positives as 999+$
	S+7	Format positives as 999$+
	S+8	Format positives as $999DB

Type	Spec	Meaning
	S+W	Format positives as windows currency
	S-0	Format negatives as ($999)
	S-1	Format negatives as -$999
	S-2	Format negatives as $-999
	S-3	Format negatives as $999-
	S-4	Format negatives as (999$)
	S-5	Format negatives as -999$
	S-6	Format negatives as 999-$
	S-7	Format negatives as 999$-
	S-8	Format negatives as $999CR
	S-W	Format negatives as windows currency
	SP	The same as S-0
	S-	The same as S-1
	S+	The same as S-1+1
	SC	The same as S-8
	SD	The same as S-8+8
Date	DW1	Day of week as Mon
	DW2	Day of week as Monday
	DWL	Day of week from Windows Long Date
	DM1	Month as 1
	DM2	Month as 01
	DM3	Month as Jan
	DM4	Month as January
	DML	Month from Windows Long Date
	DMS	Month from Windows Short Date
	DD1	Day as 1
	DD2	Day as 01
	DDL	Day from Windows Long Date
	DDS	Day from Windows Short Date
	DY1	Year as 1
	DY2	Year as 01
	DY3	Year as 1901
	DYL	Year from Windows Long Date
	DYS	Year from Windows Short Date
	DO(*s*)	*s* specifies order and separators, use %W for weekday,%D for numeric day, %M for month, and %Y for year,separators are literal, so 12/28/92 as DO(%W %M-%D-%Y) is Mon 12-28-92
	DOL	Order and separators as Windows Long Date
	DOS	Order and separators as Windows ShortDate

Type	Spec	Meaning
	D1	This is the default date format
	D2	As DM4Y3O(%M %D,%Y)
	D3	As DO(%M/%D)
	D4	As DO(%M/%Y)
	D5	As DM3O(%D-%M-%Y)
	D6	As DM3O(%M %Y)
	D7	As DM3Y3O(%D-%M-%Y)
	D8	As DY3O(%M/%D/%Y)
	D9	As DO(%D.%M.%Y)
	D10	As DO(%D/%M/%Y)
	D11	As DO(%Y-%M-%D)
Time	TH1	Hours as 1T
	TH2	Hours as 01
	THW	Hours from Windows
	TM1	Minutes as 1
	TM2	Minutes as 01
	TMW	Minutes from Windows
	TS1	Seconds as 1
	TS2	Seconds as 01
	TSW	Seconds from Windows
	TNA(*s*)	*s* is string to show after times before noon
	TNP(*s*)	*s* is string to show after times after noon
	TNW	Noon settings from Windows
	TO(*s*)	*s* specifies order and separators, use %H for hours, %M for minutes, %S for seconds, %N for am/pm, %D to include date in default format when formatting a DateTime value
	TOW	Order and separators from Windows
Logical	LT(*s*)	*s* specifies representation of logical Truth value
	LF(*s*)	*s* specifies representation of logical False value
	LY	Logical values as Yes and No
	LO	Logical values as On and Off

Example

In the following examples, assume a form contains a field called *formatField* and a button named *demoFormat*. The **pushButton** method for *demoFormat* demonstrates a number of different format specifications. For each example, the method fills the *formatField* with the formatted string, then shows a copy of the format specification in a dialog box (with **view**). The method won't procede to the next example until the View dialog box is closed; this gives you a way to

examine both the format specification and the formatted output before moving to the next example.

```
; demoFormat::pushButton
method pushButton(var eventInfo Event)
var
  x  AnyType
  fs String
endVar
fs = "\"w6\",\"This is a test\""
formatField = format("w6","This is a test")
; displays This i
fs.view("Format Spec")

fs = "\"w6\",1234567"
formatField = format("w6",1234567)
; displays 1.e+6
fs.view("Format Spec")

fs = "\"w1\",( =5)"
formatField = format("w1",( =5))
; returns True, displays T
fs.View()

fs = "\"w9.2\",1234.567"
formatField = format("w9.2",1234.567)
; displays   1234.57
fs.View()

; Here are some examples of alignment specifications:
fs = "\"w20,ac\",\"This is\""
formatField = format("w20,ac","This is")
; displays              This is
fs.view()

fs = "\"w20,ac\",\"The Title\""
formatField = format("w20,ac","The Title")
; displays            The Title
fs.view()

fs = "\"w20,ac\",\"Of the Book\""
formatField = format("w20,ac","Of the Book")
; displays          Of the Book
fs.view()

fs = "\"w20,al\",123456"
formatField = format("w20,al",123456)
; displays 123456
fs.view()

fs = "\"w20,ar\",123456"
formatField = format("w20,ar",123456)
; displays               123456
fs.view()

; Here are some examples of case specifications:
fs = "\"cu\",\"the quick brown fox\""
formatField = format("cu","the quick brown fox")
; displays THE QUICK BROWN FOX
fs.view()

fs = "\"cl\",\"JUMPS OVER THE LAZY\""
formatField = format("cl","JUMPS OVER THE LAZY")
; displays jumps over the lazy
fs.view()
```

```
fs = "\"cc\",\"dOG.\""
formatField = format("cc","dOG.")
; displays Dog.
fs.view()

fs = "\"cc\",\"widgets'r us \" + \"too\""
formatField = format("cc","widgets'r us " + "too")
; displays Widgets'R Us Too
fs.view()


; Here are some examples of edit specifications:
x = 34567.89
fs = "\"w10.2, e$c\", x"
formatField = format("w10.2, e$c", x)      ; displays $34,567.89
fs.view()

fs = "\"w10.2, e$ci\", x"
formatField = format("w10.2, e$ci", x)     ; displays $34.567,89
fs.view()

fs = "\"w13.2, e$c\", x"
formatField = format("w13.2, e$c", x)      ; displays    $34,567.89
fs.view()

fs = "\"w14.2, e$cb, al\", x"
formatField = format("w14.2, e$cb, al", x) ; displays $   34,567.89
fs.view()

fs = "\"w15.2, e$cz, al\", x"
formatField = format("w15.2, e$cz, al", x) ; displays $000034,567.89
fs.view()

fs = "\"w15.2, e$c*, al\", x"
formatField = format("w15.2, e$c*, al", x) ; displays $****34,567.89
fs.view()


; Here are some examples of sign specifications:
x = -3456.12
fs = "\"w8.2, s+\", x"
formatField = format("w8.2, s+", x)          ; displays -3456.12
fs.view()

fs = "\"w11.2, e$c, sc\", x"
formatField = format("w11.2, e$c, sc", x)    ; displays $3,456.12CR
fs.view()

fs = "\"w14.2, e$c*, sp\", x"
formatField = format("w14.2, e$c*, sp", x)   ; displays ($***3,456.12)
fs.view()

fs = "\"w13.2, e$c*, s+\", x"
formatField = format("w13.2, e$c*, s+", x)   ; displays -$***3,456.12
fs.view()

fs = "\"w14.2, e$c*, sd\", x"
formatField = format("w14.2, e$c*, sd", x)   ; displays $***3,456.12CR
fs.view()

; Here are some miscellaneous examples:
fs = "\"D2\", Date(\"3/7/1948\""
formatField = format("D2", Date("3/7/1948"))   ; displays March 7, 1948
fs.view()
```

```
fs = "\"W9.2, AL\", 1234.123"
formatField = format("W9.2, AL", 1234.123)
; displays 1234.12 in field of 9 digits with 2 decimal places
fs.view()

fs = "\"W9.2, AR\", 1234.123"
formatField = format("W9.2, AR", 1234.123)
; displays 1234.12 right aligned in same field
fs.view()

; to display date and time in 24-hour format

fs = "\"TNA(), TNP(), TO(%H:%M:%S %D), DO(%W %M/%D/%Y)\"," +

     " dateTime(\"2:30:00 pm 11/24/92\")"

formatField = format("TNA(), TNP(), TO(%H:%M:%S %D), DO(%W %M/%D/%Y)",

                     dateTime("2:30:00 pm 11/24/92"))

; displays   14:30:00 Tue 11/24/92

fs.view("Format Spec")

endmethod
```

See also

- string

ignoreCaseInStringCompares

String

Procedure

Specifies whether to consider case when comparing strings.

Syntax

ignoreCaseInStringCompares (const *yesNo* Logical)

Description

Specifies whether to consider case when comparing strings. Normally, upper- and lowercase letters don't match. For example, "Q" and "q" are not the same. But when you use **ignoreCaseInStringCompares(Yes)**, case doesn't matter, so "Q" equals "q." Once you call **ignoreCaseInStringCompares(Yes)**, it stays in effect until you call **ignoreCaseInStringCompares(No)**.

To find out if case is being considered, use **isIgnoreCaseInStringCompares**.

Example

In this example, the **pushButton** method for the *tryCompare* button checks whether Paradox is set to ignore case in string comparisons. If **isIgnoreCaseInStringCompares** returns Yes, the method uses **ignoreCaseInStringCompares** to set it to No—which means that case is considered—then compares an uppercase and lowercase string. A message window informs the user that the strings are not equivalent. Next, the method turns on case ignore, and attempts the same comparison, which returns True.

```
; tryCompare::pushButton
method pushButton(var eventInfo Event)
var
  s1,
  s2 String
endVar
s1 = "cat"
s2 = "CAT"
if isIgnoreCaseInStringCompares() then
  ignoreCaseInStringCompares(No)
endif
x = (s1 = s2)               ; the first "=" assigns, all others compare
msgInfo(s1 + " = " + s2 + "?", x)    ; displays False
ignoreCaseInStringCompares(Yes)
x = (s1 = s2)
msgInfo(s1 + " = " + s2 + "?", x)    ; displays True
endmethod
```

See also

- isIgnoreCaseInStringCompares

isIgnoreCaseInStringCompares

String

Procedure

Reports whether case is considered when comparing strings.

Syntax

isIgnoreCaseInStringCompares ( ) Logical

Description

Returns True if case is considered when comparing strings; otherwise, it returns False.

To specify whether to consider case, use **ignoreCaseInStringCompares**.

Example

See the example for ignoreCaseInStringCompares.

See also

- ignoreCaseInStringCompares

isSpace

Beginner

String

Method

Reports whether a string contains only spaces (or is empty).

Syntax

isSpace (const ***string*** String) Logical

Description

Returns True if *string* contains only whitespace, or if *string* is the empty string (""); otherwise, it returns False. Whitespace characters include spaces, tabs, carriage returns, linefeeds, and formfeeds.

Example

This example creates and checks several strings to see if the strings either contain only spaces, or contain nothing at all. This is the code for the **pushButton** method for the *valString* button:

```
; valString::pushButton
method pushButton(var eventInfo Event)
var
  s String
endVar
s = space(3)                          ; 3 spaces
msgInfo("3 Spaces", s.isSpace())      ; True
s = ""                                ; empty String
msgInfo("Empty String", s.isSpace())  ; True
s = "Z" + space(2)                    ; Z and 2 spaces
msgInfo("Z and 2 Spaces", s.isSpace()) ; False
endmethod
```

See also

❒ space

keyNameToChr

String

Procedure

Returns the one-character string represented by a virtual key-code string.

Syntax

keyNameToChr (const ***keyName*** String) String

Description

Returns the one-character string represented by the virtual key code *keyName*.

keyName is one of the virtual key codes (such as VK_BACK for Backspace), but must be supplied as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter J. ObjectPAL provides constants for virtual key codes; see Keyboard in the Constants dialog box.

Example

See the example for keyNameToVKCode.

See also

❒ chrToKeyName, keyNameToVKCode

keyNameToVKCode

String

Procedure

Returns the numeric VK_Code of a virtual keycode string.

Syntax

keyNameToVKCode (const ***keyName*** String) SmallInt

Description

Returns the VK_Code of the character represented by the string *keyName*.

keyName is one of the virtual key codes (such as VK_BACK for Backspace), but must be supplied as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and many symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*. ObjectPAL provides constants for virtual key codes; see Keyboard in the Constants dialog box.

Example

In this example, the **pushButton** method for *showCode* sets the string variable *keyStr* to an open bracket ([), then displays the ANSI code and the key name of *keyStr* in a dialog box.

```
; showCode::pushButton
method pushButton(var eventInfo Event)
var
  keyStr  String
endVar
keyStr = "["              ; set the key name for open bracket
msgInfo("VK_Code/Char", "VK_Code: " +           ; VK_Code 91
        String(keyNameToVKCode(keyStr)) +
        "\nCharacter: " + keyNameToChr(keyStr)) ; char "["
endmethod
```

See also

- chrToKeyName, keyNameToChr, VKCodeToKeyName

---

lower

String

Beginner

Method

Converts a string to lowercase.

Syntax

lower ( ) String

Description

Converts a string to lowercase letters. Use **upper** to convert a string to uppercase letters.

Example

In the following example, the **pushButton** method for *makeLower* creates an uppercase string, then uses **lower** to display it in lowercase.

```
; makeLower::pushButton
method pushButton(var eventInfo Event)
var
  myText String
endVar
myText = "HEY, EVERYBODY! IT'S QUITTIN' TIME"
msgInfo("Official Notice", myText.lower())
; displays "hey everybody! it's quittin' time"
endmethod
```

See also

- upper

lTrim

String

Beginner

Method

Removes leading blanks from a string.

Syntax

lTrim ( ) String

Description

Removes spaces and Tab characters from the left end of a string.

Example

In this example, the **pushButton** method for *trimLeft* creates a string with leading spaces and a leading tab (the escape sequence \t). The method displays the original string, uses **lTrim** to remove the leading nonprinting characters, then displays the trimmed version.

```
; trimLeft::pushButton
method pushButton(var eventInfo Event)
var
  trimMe, trimmed String
endVar
trimMe = "  \t   First word"  ; string with spaces and a tab
msgInfo("Original string", trimMe)

trimmed = trimMe.lTrim()      ; trim off spaces and tab
msgInfo("A slightly shorter version", trimmed)
; displays "First word"
endmethod
```

See also

- rTrim

match

String

Beginner

Method

Compares a string with a pattern.

Syntax

match (const ***pattern*** String [, var ***matchVar*** String]*) Logical

Description

Tests whether a string matches a pattern, and if so, extracts the components of the string that match the wildcard elements. The value of *pattern*, like patterns in queries, consists of characters interlaced with the wildcard operators .. and @. The .. matches any number of characters (including none), while the @ matches any single character. Also as in queries, **match** ignores case by default (but you can use **ignoreCaseInStringCompares(No)** to make **match** case-sensitive).

matchVar is a variable to which the matching portion will be assigned. **match** assigns matched patterns to one or more *matchVar* variables as the patterns are found. The portions of the string matching the wildcard elements are assigned to the variables from left to right. Since there may be multiple matches, the first matching substring is assigned to the first variable, the second matching substring to the second variable, and so on. If no match is found, variables are not assigned values.

To embed a quote in *pattern*, precede it with a backslash (\). To embed a period, surround it with double quotes and precede the quotes with backslashes (\".\").

Example

These statements demonstrate match functionality.

```
var
  s, x, y, z String
endVar

s = "this and that"

msgInfo("match?", s.match("t.."))        ; displays True
msgInfo("match?", s.match("@his.."))     ; displays True
msgInfo("match?", s.match("@ and that")) ; displays False
msgInfo("match?", s.match("..and.."))    ; displays True

msgInfo("match?", s.match("..and..", x, y))
                    ; displays True (x = this, y = that)

msgInfo("match?", s.match("T..", z))
  ; If isIgnoreCaseInString() is False, this statement displays
  ; False, and z is not assigned. Use
  ; ignoreCaseInStringCompares(Yes) to get this to display
  ; True, and set z to "his and that"
```

❒ advMatch, search

oemCode

String

Procedure

Returns the OEM code of a one-character string.

Syntax

oemCode (const ***char*** String) SmallInt

Description

Returns the OEM code of *char*. The OEM code returned is an integer between 1 and 255.

Example

See the example for ansiCode.

See also

❒ ansiCode, chrToKeyName

rTrim

String

Beginner

Method

Removes trailing blanks from a string.

Syntax

rTrim ( ) String

Description

Removes spaces, tabs, carriage returns, and linefeed characters from the right end of a string.

Example

In this example, the **pushButton** method for *trimRight* creates a string with trailing spaces. The method displays the original string, uses **rTrim** to remove the trailing nonprinting characters, then displays the trimmed version.

```
; trimRight::pushButton
method pushButton(var eventInfo Event)
var
  trimMe, trimmed String
endVar
trimMe = "Last word     "     ; string with trailing spaces
msgInfo("Original string", trimMe + "The end")
; displays "Last word     The end"

trimmed = trimMe.rTrim()      ; trim off spaces
msgInfo("A slightly shorter version", trimmed + "The end")
; displays "Last wordThe end"
endmethod
```

See also

- lTrim

search

String

Beginner

Method

Returns the position of one string inside another.

Syntax

search (const ***str*** String) SmallInt

Description

Tests for an occurrance of *str* within a target string. If *str* is found, **search** returns the starting character position of *str* within the target string; otherwise, it returns 0. The search always begins at the first character of the target string.

By default, **search** is not case-sensitive, but you can use **ignoreCaseInStringCompares** to make it case-sensitive.

Example

The following example searches for parts of the string "Goliath" and "Golgolithic". The following code is attached to the **pushButton** method for the *searchStr* button:

```
; searchStr::pushButton
method pushButton(var eventInfo Event)
var
  s String
endVar
s = "Goliath"
msgInfo("Where is lia in Goliath?", s.search("lia")) ; displays 3
msgInfo("Where is lai in Goliath?", s.search("lai")) ; displays 0
ignoreCaseInStringCompares(No)
s = "Golgolithic"
msgInfo("Where is gol in Golgolithic?", s.search("gol"))
; displays 4
; Note: If ignoreCaseInStringCompares is on, the last
; search yields a 1 instead.
endmethod
```

See also

- advMatch, match

---

size

String

Beginner

Method

Returns the length of a string.

Syntax

size ( ) SmallInt

Description

Returns the number of characters (including spaces) in a string.

Example

In this example, the **pushButton** method for *getSize* assigns a string to the variable *sourceText*, then displays the sentence and its size in a dialog box. The method then uses **size** to get the first half of *sourceText*, and assign it back to *sourceText*. The size of the *sourceText* and the smaller *sourceText* are displayed in a dialog box.

```
; getSize::pushButton
method pushButton(var eventInfo Event)
var
  sourceText String
endVar
sourceText = "This is a short sentence."
msgInfo("Size", "Length: " + String(sourceText.size()) +
                "\n" + sourceText)
; displays   Length: 25
;            This is a short sentence.

; now chop the sentence in half
sourceText = subStr(sourceText, 1, SmallInt(sourceText.size()/2))
msgInfo("Half-Size", "Length: " + strVal(sourceText.size())
                     + "\n" + sourceText)
; displays   Length: 12
```

```
;         This is a short sentence
endmethod
```

See also

❐ string

space

String

Beginner

Procedure

Creates a string of a specifed number of spaces.

Syntax

space (const ***numberOfSpaces*** SmallInt) String

Description

Returns a string of *numberOfSpaces* spaces.

Example

See the example for isSpace.

See also

❐ isSpace

string

String

Beginner

Procedure

Casts a value as a String.

Syntax

string (const ***value*** AnyType [, const ***value*** AnyType]*) String

Description

Casts (converts) an expression *value* to a String. If you specify multiple arguments, **string** will cast them all to strings and concatentate them to one string.

Example

In the following example, the **pushButton** method for *getNumToString* requests a number from the user, then casts it as a string and concatenates it with another string for display in a **msgInfo** dialog box.

```
; getNumToString::pushButton
method pushButton(var eventInfo Event)
var
   nn Number
endVar
nn = 0.0                    ; initialize the number
nn.View("Enter a number")   ; display it, and ask for input

; Note: Because you can enter only one argument for the text of
; the msgInfo dialog box, if you have any non-string elements, they
; must be cast as strings, then concatenated. Here, nn is cast
; to a String type before being concatenated with "You entered "
```

```
msgInfo("Status", "You entered " + string(nn))
msgInfo("Status", string("You entered ", nn))  ; also works
endmethod
```

See also

- format

strVal

String

Procedure

Converts a value to a string.

Syntax

strVal (const ***value*** AnyType) String

Description

Converts *value* to a string. The data type of *value* can be any of the types represented by AnyType.

Example

See the example for size.

See also

- string

substr

Beginner

String

Method

Returns a portion of a string.

Syntax

subStr (const ***startIndex*** Number
[, const ***numberOfChars*** SmallInt]) String

Description

Returns a portion of a string that starts at *startIndex* and continues for *numberOfChars* characters. The value of *startIndex* must be greater than 0 and less than or equal to the size of the string. If *numberOfChars* is 0, **subStr** returns a null string. If *numberOfChars* is omitted, **subStr** returns the character at position *startIndex*.

Example

In this example, assume a form contains a button named *getPhone* and four fields named *wholePhone, phAreaCode, phExchange,* and *phNumber*. The method in this example uses **subStr** to extract the three groups of digits from a U.S. phone number. The following code is attached to the **pushButton** method for *getPhone*.

```
; getPhone::pushButton
method pushButton(var eventInfo Event)
var
  phoneNum  String
endVar
```

String

```
phoneNum = wholePhone.Value
; assume phone number has been entered as ###-###-####
; start from first position, take three characters
phAreaCode.Value = phoneNum.subStr(1, 3)  ; get the area code
phExchange.Value = phoneNum.subStr(5, 3)  ; get the exchange
phNumber.Value   = phoneNum.subStr(9, 4)  ; get the number
beep()
endmethod
```

See also

❐ breakApart, search, size

toANSI

String

Method

Converts a string of OEM characters to ANSI characters.

Syntax

toANSI ( ) String

Description

Converts a string of OEM characters to ANSI characters.

Example

In this example, the **pushButton** method for a button named *showANSI* displays a string in two ways: in the title of the dialog box the string is displayed as is; in the window of the dialog box, the string is first converted to ANSI. The last character in the string is the copyright symbol (©). This symbol prints in the title of the dialog box; however, in the window of the dialog box, the symbol is replaced by an underscore (_).

```
; showANSI::pushButton
method pushButton(var eventInfo Event)
var
  ss String
endVar
; string plus copyright symbol
ss = "A string of characters " + chr(169)
msgInfo(ss, ss.toANSI())
; displays string plus "_" in window of dialog box - system-dependent
endmethod
```

See also

❐ toOEM

toOEM

String

Method

Converts a string of ANSI characters to OEM characters.

Syntax

toOEM ( ) String

Description

Converts a string of ANSI characters to OEM characters.

Example

In this example, the **pushButton** method for a button named *showOEM* displays a string in two ways: in the title of the dialog box the string is displayed as is; in the window of the dialog box, the string is first converted to OEM. The last character in the string is the copyright symbol (©). This symbol prints in the title of the dialog box; however, in the window of the dialog box, the symbol is replaced by the letter *c*.

```
; showOEM::pushButton
method pushButton(var eventInfo Event)
var
  ss String
endVar
; string plus copyright symbol
ss = "A string of characters " + chr(169)
msgInfo(ss, ss.toOEM())
; displays string plus "c" in window of dialog box
endmethod
```

See also

- toANSI

---

upper

Beginner

String

Method

Converts a string to uppercase.

Syntax

upper ( ) String

Description

Converts a string to uppercase letters. Use **lower** to convert a string to lowercase letters.

Example

In this example, the **pushButton** method for *makeUpper* gets a string from the user, then converts it to uppercase. The converted string is then compared to an uppercase string constant.

```
;makeUpper:pushButton
method pushButton(var eventInfo Event)
const
  ORDERTYPE = "BIDORDER"   ; concatenate two valid types
endConst
var
  myText String
  x      SmallInt
endVar
myText = ""                          ; initialize the string
myText.view("Enter 'Bid' or 'Order'") ; get a response
myText = myText.upper()              ; convert to uppercase
if search(ORDERTYPE, myText) > 0 then
  ; search for a matching string -- returns location
  ; of match, or zero if no match
```

```
  msgInfo("Status", "You entered a valid type.")
else
  msgStop("Stop", "You must enter either Bid or Order.")
endif
endmethod
```

See also

❒ lower

vkCodeToKeyName

String

Procedure

Converts a virtual keycode constant to a virtual keycode string.

Syntax

vkCodeToKeyName (const ***vkCode*** SmallInt) String

Description

Returns the virtual key-code name, as a String, of the character represented by *vkCode*.

A key name is one of the virtual key codes (such as VK_BACK for Backspace), but is returned as a string (such as "VK_BACK"), not a constant. Alphanumeric characters and most symbols have a key name that consists simply of the character, for instance, "J" for the letter *J*. ObjectPAL provides constants for virtual key codes; see Keyboard in the Constants dialog box.

Example

See the example for ansiCode.

See also

❒ ansiCode, chr, chrToKeyName, keyNameToChr

System

System

beep	errorClear	msgAbortRetryIgnore
close	errorCode	msgInfo
constantNameToValue	errorLog	msgQuestion
constantValueToName	errorMessage	msgRetryCancel
cpuClockTime	errorPop	msgStop
debug	errorShow	msgYesNoCancel
dlgAdd	errorTrapOnWarnings	pixelsToTwips
dlgCopy	execute	play
dlgCreate	exit	readEnvironmentString
dlgDelete	fail	readProfileString
dlgEmpty	fileBrowser	setMouseScreenPosition
dlgNetDrivers	formatAdd	setMouseShape
dlgNetLocks	formatDelete	sleep
dlgNetRefresh	formatExist	sound
dlgNetRetry	formatSetCurrencyDefault	sysInfo
dlgNetSetLocks	formatSetDateDefault	tracerClear
dlgNetSystem	formatSetDateTimeDefault	tracerHide
dlgNetUserName	formatSetLogicalDefault	tracerOff
dlgNetWho	formatSetLongIntDefault	tracerOn
dlgRename	formatSetNumberDefault	tracerSave
dlgRestructure	formatSetSmallIntDefault	tracerShow
dlgSort	formatSetStringDefault	tracerToTop
dlgSubtract	formatSetTimeDefault	tracerWrite
dlgTableInfo	getMouseScreenPosition	twipsToPixels
enumDesktopWindowNames	helpOnHelp	version
enumFonts	helpQuit	winGetMessageId
enumFormNames	helpSetIndex	winPostMessage
enumReportNames	helpShowContext	winSendMessage
enumRTLClassNames	helpShowIndex	writeEnvironmentString
enumRTLConstants	helpShowTopic	writeProfileString
enumRTLMethods	helpShowTopicInKeywordTable	
enumWindowNames	message	

The System type contains procedures for displaying messages, finding out about the user's system, manipulating the Browser, working with the Help system, and more.

For more information and examples, see Chapter 11 in the *ObjectPAL Developer's Guide*.

beep

Beginner

System

Procedure

Sounds the Windows default beep.

Syntax

beep ( )

Description

Activates the Windows default beep sound. The beep will be audible only if the Enable System Sounds option is checked in the Control Panel's Sound dialog box.

To send a sound of specified pitch and duration to the system speaker, use **sound**.

Example

The following code is attached to a button's **pushButton** method. It prompts the user to enter a number and beeps if the number is out of range.

```
; getANumber::pushButton
method pushButton(var eventInfo Event)
var
  someNumber  SmallInt
endVar
someNumber = 1
someNumber.view("Pick a number between 1 and 10")
while someNumber < 1 OR someNumber > 10
  beep()          ; beep
  sleep(100)      ; slight pause, otherwise beeps run together as one
  beep()
  msgStop("Oops", "That number is too large or too small. Try again.")
  someNumber.view("Pick a number between 1 and 10")
endwhile
endmethod
```

See also

- sound

---

close

System

Beginner

Procedure

Closes the current form.

Syntax

close ([const ***returnValue*** AnyType])

Description

Posts a request to close the current form. If *returnValue* is specified, a value will be returned to the calling form (if there is one). Specifying a *returnValue* when there is no calling form does not result in an error. **close** starts the process of closing the form, which includes removing the focus and departing.

Example

The following code closes the current form after asking the user for confirmation.

```
; closeButton::pushButton
method pushButton(var eventInfo Event)
var
```

```
  qAnswer String
endVar
qAnswer = msgYesNoCancel("Closing Application",
          "Do you want to close this form?")
if qAnswer = "Yes" then
  close()                   ; close the current form
else
  message("Application not closed.")
endif
endmethod
```

See also

- exit

constantNameToValue

System

Procedure

Returns the numeric value of a constant.

Syntax

constantNameToValue (const ***constantName*** String) AnyType

Description

Returns the value represented by the ObjectPAL constant specified in *constantName*. **constantNameToValue** returns values only for predefined ObjectPAL constants; it will not return a value for a constant you have defined yourself.

Note For readability and portability (among other benefits), we recommend using constant names rather than numeric values. Appendix G lists all ObjectPAL constants and their values; constants are also listed online.

Example

The following code returns the numeric value for the action constant DataBeginEdit.

```
; showValOfConst::pushButton
method pushButton(var eventInfo Event)
var
  constValue   AnyType
  constString  String
  tf           Logical
endvar
constValue = constantNameToValue("DataBeginEdit")  ; constant is passed as a
                                                    ; String
msgInfo("The value of DataBeginEdit is", constValue)
tf = constantValueToName("ActionDataCommands", constValue, constString)
if tf then   ; if the conversion worked properly, display the string
  msgInfo("The name of " + String(constValue) + " is", constString)
else
  msgInfo("Status", "Something went wrong with that conversion.")
endif
endmethod
```

See also

- constantValueToName, enumRTLConstants
- Appendix G

System

constantValueToName

System

Procedure

Reports on the name of a constant.

Syntax

constantValueToName (const ***groupName*** String, const ***value*** AnyType, var ***constName*** String) Logical

Description

Writes to *constName* the name of a constant with the *value* specified that belongs to the group *groupName*. The constant name is written to the variable *constName*. **constantValueToName** works for names of predefined ObjectPAL constants only; it will not work for a constant you have defined yourself.

Example

See the example for constantNameToValue.

See also

- constantNameToValue, enumRTLConstants
- Appendix G

cpuClockTime

Beginner

System

Procedure

Returns the number of seconds since the computer was started.

Syntax

cpuClockTime ( ) LongInt

Description

Returns the number of milliseconds since the computer was started. The minimum clock increment is 55 milliseconds. This procedure is useful for measuring the interval between two events.

Example

This example uses **cpuClockTime** to compare execution times for two **for** loops: one with an undeclared variable, one with a declared variable. Although execution times vary by system, the loop executes significantly faster when the variable is declared.

```
; clockVars::pushButton
method pushButton(var eventInfo Event)
var
  fastVar    SmallInt
  delta      String
  startTime,
  stopTime   LongInt
endvar
startTime = cpuClockTime()                    ; clock's time before starting
for slowVar from 1 to 10000                   ; slowVar is undeclared
  slowVar = slowVar + 1
endFor
```

```
stopTime = cpuClockTime()                    ; clock's time after 10000 loops
delta = String(stopTime - startTime)         ; find the elapsed time using
delta.view("Time for undeclared variable")   ; an undeclared variable --
                                             ; times vary by system
startTime = cpuClockTime()
for fastVar from 1 to 10000                  ; fastVar is declared
  fastVar = fastVar + 1
endFor
stopTime = cpuClockTime()
delta = String(stopTime - startTime)         ; find the elapsed time using
delta.view("Time for declared variable")     ; a declared variable
msgInfo("And the moral is:", "For the best performance, " +
        "declare variables!")
endmethod
```

See also

- time in the Time type

debug

System

Procedure

Halts execution of a method and invokes the Debugger.

Syntax

debug ( )

Description

Placing **debug** in a method has the same effect as setting a breakpoint. When Paradox encounters **debug** in a method, the method stops executing, and the Debugger window opens with the cursor on the line containing **debug**. Unlike breakpoints, **debug** statements are saved with the method's source code. Also, **debug** statements take effect only when you choose Debug | Enable Debug Statement; otherwise, they are ignored.

Note **debug** works only in methods and procedures that you write, not for methods and procedures in the ObjectPAL run-time library.

debug is handy for setting persistent breakpoints in methods while you are developing an application. You can test the application with Debug | Enable DEBUG Statement turned on, and deliver the application with it turned off.

Example

In this example, assume the Debug | Enable DEBUG Statement ObjectPAL Editor menu command is selected. The following code executes a **for** loop. Halfway through the loop, the call to **debug** suspends execution and opens an Editor window containing the code. Choose Debug | Run to resume execution, or use the other Debugger features described in the *ObjectPAL Developer's Guide.*

```
; startDebugAt50::pushButton
method pushButton(var eventInfo Event)
var
  i SmallInt
endVar
```

System

```
for i from 1 to 100
          message(i)
          if i = 50 then
     debug()     ; will work only if Debug|Enable DEBUG
                 ; ObjectPAL Editor menu command is checked
          endIf
endFor
endmethod
```

See also

- execute
- Chapter 4 in the *ObjectPAL Developer's Guide*, for information on the ObjectPAL Debugger

dlgAdd

System

Procedure

Invokes the Table Add dialog box.

Syntax

dlgAdd (const ***tableName*** String)

Description

Displays the Table Add dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Add. The variable *tableName* specifies the source table.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Add dialog box and fills in the *Customer* table name as the source table. The user must fill in the destination table name and close the dialog box.

```
; showAddDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Add dialog box with Customer as the source
dlgAdd("customer.db") ; same as File|Utilities|Add
endmethod
```

See also

- dlgCopy, dlgEmpty, dlgSubtract

dlgCopy

System

Procedure

Invokes the Table Copy dialog box.

Syntax

dlgCopy (const ***tableName*** String)

Description

Displays the Table Copy dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Copy. The variable *tableName* specifies the source table.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Copy dialog box and fills in the *Customer* table name as the source table. The user must fill in the destination table name and close the dialog box.

```
; showCopyDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Copy dialog box with the Customer table as the source
dlgCopy("customer.db") ; same as File|Utilities|Copy
endmethod
```

See also

- dlgAdd, dlgEmpty, dlgSubtract

dlgCreate

System

Procedure

Invokes the Create Table dialog box.

Syntax

dlgCreate (const ***tableName*** String)

Description

Displays the Create Table dialog box (described in the *User's Guide*), just as if you had chosen File | New | Table. The variable *tableName* specifies the name of table to create.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Create dialog box. The user must choose the table type, fill out the field roster, and save the created table.

```
; showCreateDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Create dialog box
dlgCreate("sometbl.db") ; same as  File|New|Table
endmethod
```

See also

- dlgCopy, dlgDelete

dlgDelete

System

Procedure

Invokes the Table Delete dialog box.

Syntax

dlgDelete (const ***tableName*** String)

Description

Displays the Table Delete dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Delete. The variable *tableName* specifies the name of table to delete.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Delete dialog box and fills in the *Customer* table name as the table to delete. The user must close the dialog box and confirm the deletion.

```
; showDeleteDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Delete dialog box for the Customer table
dlgDelete("Customer.db")  ; same as File|Utilities|Delete
endmethod
```

See also

- dlgCreate, dlgEmpty

dlgEmpty

System

Procedure

Invokes the Table Empty dialog box.

Syntax

dlgEmpty (const ***tableName*** String)

Description

Displays the Table Empty dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Empty. The variable *tableName* specifies the name of table to empty.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Empty dialog box and fills in the *Customer* table name as the table to empty. The user must close the dialog box and confirm the data loss.

```
method pushButton(var eventInfo Event)
; invokes the Table Empty dialog box for Customer table
dlgEmpty("Customer.db") ; same as File|Utilities|Empty
endmethod
```

See also

- dlgDelete, dlgSubtract

dlgNetDrivers

System

Procedure

Invokes the Drivers dialog box.

Syntax

dlgNetDrivers ( )

Description

Displays the Drivers dialog box (described in the *User's Guide*), just as if you had chosen File | System Settings | Drivers.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following code opens the Drivers dialog box.

```
; showNetDrivers::pushButton
method pushButton(var eventInfo Event)
; invoke the Drivers dialog box
dlgNetDrivers()  ; same as File|System Settings|Drivers
endmethod
```

See also

- enumDriverCapabilities, enumDriverInfo, and enumDriverNames in the Session type

dlgNetLocks

System

Procedure

Creates and displays a table of lock information.

Syntax

dlgNetLocks ( )

Description

Invokes the Select File dialog box and prompts you to choose a table, just as if you had chosen File | Multiuser | Display Locks. When you choose a table and click OK, Paradox creates a Paradox table named LOCKS.DB in your private directory. If the table exists, Paradox overwrites it without asking for confirmation. This method fails if the table is already open.

Here is the structure of LOCKS.DB:

Field Name	Type	Size
Type	S	
UserName	A	14
NetSession	S	
OurSession	S	
RecordNum	N	
Count	S	

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

After Paradox creates the LOCKS table, Paradox displays it in a Table window.

Example

The following example opens the Select File dialog box. Once the user chooses a file, a *Locks* table is created and displayed.

```
; showNetLocks::pushButton
method pushButton(var eventInfo Event)
; creates a table of lock info :PRIV:LOCKS.DB, then displays it
dlgNetLocks()  ; same as File|Multiuser|Display Locks
endmethod
```

See also

- dlgNetRetry, dlgNetSetLocks
- enumLocks in the TCursor type

---

dlgNetRefresh

System

Procedure

Invokes the Network Refresh Rate dialog box.

Syntax

dlgNetRefresh ( )

Description

Displays the Network Refresh Rate dialog box (described in the *User's Guide*), just as if you had chosen File | System Settings | Auto Refresh.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

This example opens the Network Refresh Rate dialog box.

```
; showNetRefresh::pushButton
method pushButton(var eventInfo Event)
; invoke the Network Refresh Rate dialog
dlgNetRefresh()  ; same as File|System Settings|Auto Refresh
endmethod
```

See also

- dlgNetRetry, dlgNetWho

dlgNetRetry

System

Procedure

Invokes the Network Retry Period dialog box.

Syntax

dlgNetRetry ( )

Description

Displays the Network Retry Period dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Set Retry.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

This example opens the Network Retry Period dialog box.

```
; showNetRetryDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Network Retry Period dialog box
dlgNetRetry() ; same as File|Multiuser|Set Retry
endmethod
```

See also

- dlgNetLocks

dlgNetSetLocks

System

Procedure

Invokes the Table Locks dialog box.

Syntax

dlgNetSetLocks ( )

Description

Displays the Table Locks dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Set Locks.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

This example opens the Table Locks dialog box.

```
; showSetLocks::pushButton
method pushButton(var eventInfo Event)
dlgNetSetLocks()  ; invoke the Table Locks dialog box
                  ; same as choosing File|Multiuser|Set Locks
endmethod
```

See also

- dlgNetLocks

dlgNetSystem

System

Procedure

Invokes the ODAPI System Information dialog box.

Syntax

dlgNetSystem ( )

Description

Displays the ODAPI System Information dialog box (described in the *User's Guide*), just as if you had chosen File | System Settings | ODAPI.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

This code opens the ODAPI System Information dialog box.

```
; showNetSystem::pushButton
method pushButton(var eventInfo Event)
; invoke the ODAPI System Information dialog box
dlgNetSystem()    ; same as File|System|ODAPI
endmethod
```

See also

- dlgNetDrivers
- enumDriverCapababilites, enumDriverInfo, and enumDriverNames in the Session type

dlgNetUserName

System

Procedure

Invokes the Network User Name dialog box.

Syntax

dlgNetUserName ( )

Description

Displays the Network User Name dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | User Name.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following code opens the Network User Name dialog box, which shows the current user's network name.

```
; showUserName::pushButton
method pushButton(var eventInfo Event)
; invoke the Network User Name dialog box
dlgNetUserName()    ; same as File|Multiuser|User Name
endmethod
```

See also

- dlgNetWho

dlgNetWho

System

Procedure

Invokes the Current Users dialog box.

Syntax

dlgNetWho ( )

Description

Displays the Current Users dialog box (described in the *User's Guide*), just as if you had chosen File | Multiuser | Who.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to close the dialog box.

Example

The following code opens the Current Users dialog box.

```
; showUserList::pushButton
method pushButton(var eventInfo Event)
; invoke the Current Users dialog box
dlgNetWho()  ; same as File|Multiuser|Who
endmethod
```

See also

- dlgNetUserName

dlgRename

System

Procedure

Invokes the Table Rename dialog box.

Syntax

dlgRename (const ***tableName*** String)

Description

Displays the Table Rename dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Rename. The variable *tableName* specifies the name of table to rename.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Rename dialog box and fills in the *Customer* table name as the table to rename. The user must enter a new name and close the dialog box.

```
; showRenameDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Rename dialog box
dlgRename("customer.db") ; same as File|Utilities|Rename
endmethod
```

See also

- dlgCopy, dlgDelete, dlgSort

dlgRestructure

System

Procedure

Invokes the Table Restructure dialog box.

Syntax

dlgRestructure (const ***tableName*** String)

Description

Displays the Table Restructure dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Restructure. The variable *tableName* specifies the name of table to restructure.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Rename dialog box and fills in the *Customer* table name as the table to restructure. The user must modify the structure and close the dialog box.

```
; showRestructureDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Restructure dialog box for Customer table
dlgRestructure("customer.db")   ; same as File|Utilities|Restructure
endmethod
```

See also

- dlgCreate

dlgSort

System

Procedure

Invokes the Table Sort dialog box.

Syntax

dlgSort (const ***tableName*** String)

Description

Displays the Table Sort dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Sort. The variable *tableName* specifies the name of table to sort.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Sort dialog box and chooses the *Customer* table name as the table to sort. The user must create a sort specification and close the dialog box.

```
; showSortDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Sort dialog box
dlgSort("customer.db")      ; same as File|Utilities|Sort
endmethod
```

See also

- dlgRename

dlgSubtract

System

Procedure

Invokes the Table Subtract dialog box.

Syntax

dlgSubtract (const ***tableName*** String)

Description

Displays the Table Subtract dialog box (described in the *User's Guide*), just as if you had chosen File | Utilities | Subtract. The variable *tableName* specifies the name of table to subtract records from.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Table Subtract dialog box and fills in the *Customer* table name as the source table—the table with records to subtract. The user must fill in the name of the table to subtract records from and close the dialog box.

```
; showSubtractDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Subtract dialog box
dlgSubtract("customer.db")  ; File|Utilities|Subtract
endmethod
```

See also

- dlgAdd, dlgDelete

---

dlgTableInfo

System

Procedure

Invokes the Structure Information dialog box.

Syntax

dlgTableInfo (const ***tableName*** String)

Description

Invokes the Structure Information dialog box, just as if you had chosen File | Utilities | Info Structure. The variable *tableName* specifies the name of table from which to subtract records.

ObjectPAL code suspends execution until the user closes this dialog box. ObjectPAL has no control over this dialog box once it is displayed; it's up to the user to fill in any necessary information and close the dialog box.

Example

The following example invokes the Structure Information dialog box for the *Customer* table.

```
; showTableInfo::pushButton
method pushButton(var eventInfo Event)
; invoke the Structure Information dialog box for the Customer table
dlgTableInfo("customer.db") ; same as File|Utilities|Info Structure
endmethod
```

See also

- dlgCreate, dlgRestructure

enumDesktopWindowNames

System

Procedure

Creates a table listing open Paradox windows.

Syntax

1. **enumDesktopWindowNames** (const ***tableName*** String)
2. **enumDesktopWindowNames** (const ***windowNames*** Array[] String)

Description

Lists open Paradox window names. Syntax 1 creates the Paradox table named in *tableName* listing the name, class, position, and size of each open window in the user's system. The table lists all applications opened by Paradox. By default, the table is created in the working directory. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the structure of the table:

Field Name	Type	Size
WindowName	A	32
ClassName	A	32
Position	A	12
Size	A	12
Handle	S	

Syntax 2 fills the array named in *winArray* with the names of the applications. You must declare the array before calling this method. Applications are listed in Windows z-order; that is, the application displayed on top is listed first in the array, the application in the second layer is listed second, and so on.

Compare this method to **enumWindowNames**, which lists all Windows applications running on the user's system.

Example

This example writes the open desktop window titles to an array and shows the array. Next, the method creates and displays a table that lists the open desktop window names.

```
; getDesktopWinNames::pushButton
method pushButton(var eventInfo Event)
var
  winNames Array[] String
  tempTV         TableView
endvar
tempTV.open("Customer")              ; open a table window
enumDesktopWindowNames(winNames)     ; enum desktop window names to an array
winNames.view() ; lists all windows open in the Paradox Desktop, if
                ; method editor window is open, lists first 32 chars
enumDesktopWindowNames("wNameTbl.db") ; enum to a table
```

System

```
tempTV.open("wNameTbl")                ; show the table
endmethod
```

See also

- enumFormNames, enumReportNames, enumWindowNames

enumFonts

System

Procedure

Creates a table listing fonts in the user's system.

Syntax

enumFonts (const ***tableName*** String)

Description

Creates a Paradox table *tableName* listing fonts in the user's system. By default, this method creates the table in the user's working directory. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the table structure:

Field Name	Type	Size
FaceName	A	64
FontSize	A	8
Attribute	A	64

Example

This example calls **enumFonts** to list the fonts to a table named FONTS.DB. Then it searches a TCursor for a font named Modern. If Modern is in the table, it sets the Font.TypeFace property of a field object named *balanceField* to Modern.

```
; getFonts::pushButton
method pushButton(var eventInfo Event)
var
  fontsTC TCursor
  tempTV  TableView
endVar
enumFonts("fonts.db")         ; write font names to a table
tempTV.open("fonts.db")       ; show the table
dlgTableInfo("fonts.db")      ; show the table structure
fontsTC.open("fonts.db")
if fontsTC.locate("FaceName", "Modern") then
  balanceField.Font.TypeFace = "Modern"
endIf
fontsTC.close()
endmethod
```

See also

- enumRTLConstants

enumFormNames

System

Procedure

Creates an array listing open forms.

Syntax

enumFormNames (var ***formNames*** Array[] String)

Description

Fills the array named in *formNames* with the names of the forms opened in the user's desktop. You must declare *formNames* as a resizeable array before calling this method. Forms are listed in Windows z-order; that is, the form displayed on top is listed first in the array, the form in the second layer is listed second, and so on.

Example

The code in this example writes the open forms, reports, and libraries to an array named *openForms*, then views the *openForms* array.

```
; getFormNames::pushButton
method pushButton(var eventInfo Event)
var
  openForms Array[] String
endVar
enumFormNames(openForms)
openForms.view()        ; lists forms
endmethod
```

See also

- enumDesktopWindowNames, enumReportNames, enumWindowNames

enumReportNames

System

Procedure

Creates an array listing open reports.

Syntax

enumReportNames (var ***reportNames*** Array[] String)

Description

Fills the array named in *reportNames* with the names of the reports open in the user's desktop. You must declare *reportNames* as a resizeable array before calling this method. Reports are listed in Windows z-order; that is, the report displayed on top is listed first in the array, the report in the second layer is listed second, and so on.

Example

The code in this example writes the open forms, reports, and libraries to an array named *openReports*, then views the *openReports* array.

```
; getReportNames::pushButton
method pushButton(var eventInfo Event)
var
  openReports Array[] String
```

```
endVar
enumReportNames(openReports)
openReports.view()          ; lists reports
endmethod
```

See also

- enumDesktopWindowNames, enumFormNames, enumWindowNames

enumRTLClassNames

System

Procedure

Creates a table listing the object types in ObjectPAL.

Syntax

enumRTLClassNames (const ***tableName*** String) Logical

Description

Creates a Paradox table *tableName* listing the names of all object types in the ObjectPAL run-time library. By default, this table is created in the working directory (:WORK:). If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the structure of the table:

Field Name	Type	Size
ClassName	A	32*

Example

This example writes the run-time library type names to a table named *Rtlclass* and views the table.

```
; getRTLClasses::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLCLassNames("rtlclass.db")     ; write type names to table
tempTV.open("rtlclass")              ; show the table
endmethod
```

See also

- enumRTLConstants, enumRTLMethods

enumRTLConstants

System

Procedure

Creates a table listing the constants defined by ObjectPAL.

Syntax

enumRTLConstants (const ***tableName*** String) Logical

Description

Creates a Paradox table *tableName* listing all the constants defined in the ObjectPAL run-time library. By default, this table is created in the working directory (:WORK:). If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the structure of the table:

Field Name	Type	Size
GroupName	A	32*
ConstantName	A	48*
Type	A	48
Value	A	64

Note Although Paradox provides the values of constants, you should not use the constant's value in code: refer to constants by name. Use the **constantValueToName** and **constantNameToValue** methods to convert values to constants, if necessary.

Example

This example writes the run-time library constant descriptions to a table named *Rtlconst* and views the table.

```
; getRTLConsts::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLConstants("rtlconst.db")      ; write constants to table
tempTV.open("rtlconst")              ; show the table
endmethod
```

See also

- constantNameToValue, constantValueToName, enumRTLClasses, enumRTLMethods

enumRTLMethods

System

Procedure

Creates a table listing the methods in ObjectPAL.

Syntax

enumRTLMethods (const ***tableName*** String) Logical

Description

Creates a Paradox table *tableName* listing all the methods defined in the ObjectPAL run-time library. By default, this table is created in the working directory (:WORK:). If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the table structure:

Field Name	Type	Size
ClassName	A	32*
MethodType	A	8*
MethodName	A	64*
MethodArgs	A	255*
ReturnType	A	32*

Example

This example writes the run-time library method descriptions to a table named *Rtlmeth* and views the table.

```
; getRTLMethods::pushButton
method pushButton(var eventInfo Event)
var
  tempTV TableView
endVar
enumRTLMethods("rtlmeth.db")    ; write method names to table
tempTV.open("rtlmeth")          ; show the table
endmethod
```

See also

- enumRTLClasses, enumRTLConstants

enumWindowNames

System

Procedure

Creates a table or an array listing open windows.

Syntax

1. enumWindowNames (const ***tableName*** String) Logical

2. enumWindowNames (var ***windowNames*** Array[] String)

Description

Creates a list of the applications currently running under Windows on the user's system.

Syntax 1 creates the Paradox table named in *tableName* listing the name, class, position, and size of each open window in the user's system. The table lists windows opened by any application, not just Paradox. By default, the table is created in the working directory (:WORK:). If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

Here is the structure of the table:

Field Name	Type	Size
WindowName	A	32
ClassName	A	32
Position	A	12

Size	A	12
Handle	S	

Syntax 2 fills the array named in *winArray* with the names of the applications. You must declare the array before calling this method. Applications are listed in Windows *z*-order; that is, the application displayed "on top" is listed first in the array, the application in the second layer is listed second, and so on.

Compare this method to **enumDesktopWindowNames**, which lists open windows for Paradox only.

Example

In this example, the **pushButton** method for a button named *getWindowNames* writes and displays open window information in two ways. First, the method fills an array with the titles of the open windows and displays the array. Then, the method fills a table with descriptions of the open windows and shows the table, as well as the table's structure.

```
; getWindowNames::pushButton
method pushButton(var eventInfo Event)
var
  winNames Array[] String
  tempTV         TableView
endvar
enumWindowNames(winNames)       ; write names to an array
winNames.view()                 ; lists all open windows
                                ; if a method editor window is open,
                                ; lists first 32 chars
enumWindowNames("wNameTbl.db")  ; write window descriptions to a table
tempTV.open("wNameTbl")         ; show the table
dlgTableInfo("wNameTbl.db")     ; show the table structure
endmethod
```

See also

- enumDesktopWindowNames, enumFormNames, enumReportNames

errorClear

System

Procedure

Clears the error stack.

Syntax

errorClear ( )

Description

Clears (empties) the error stack of all error codes and error messages. For more information about the error stack, refer to the *ObjectPAL Developer's Guide.*

Example

This code clears the error stack.

```
; clearError::pushButton
method pushButton(var eventInfo Event)
errorClear()          ; clear the error stack
endmethod
```

See also

- errorCode, errorMessage, errorPop, errorShow

errorCode

Beginner

System

Procedure

Returns a number describing the most recent run-time error or error condition.

Syntax

errorCode ( ) SmallInt

Description

Returns an integer describing the most recent run-time error or error condition. ObjectPAL provides constants for these integers (for example, peObjectNotFound); see Errors in the Constants dialog box.

Example

The method in this example uses a **try** clause to attempt to attach to an object named *boxOne* on the current form. If the object doesn't exist, a critical error occurs, and control moves to the **onFail** clause. The **onfail** clause uses **errorCode** to discover the error, then takes appropriate action.

```
; handleErrorcode::pushButton
method pushButton(var eventInfo Event)
var
  obj UIObject
endVar
try
  obj.attach("boxOne")
  obj.color = Red
onFail
  msgInfo("Status", "Original attempt failed. Taking a different tack.")
  if errorCode() = peObjectNotFound then
    obj.create(BoxTool, 180, 180, 360, 360)
    obj.name = "boxOne"
    obj.visible = Yes
    reTry
 else
   fail()
 endIf
endTry
endmethod
```

See also

- errorLog, errorMessage, errorPop, errorShow

errorLog

System

Procedure

Adds information to the error stack.

Syntax

errorLog (const ***errorCode*** SmallInt, const ***errorMessage*** String)

Description

Adds (pushes) the error information specified in *errorCode* and *errorMessage* onto the error stack.

For more information about the error stack, refer to the *ObjectPAL Developer's Guide.*

Example

The method in this example uses a **try** clause to attempt to attach to an object named *boxOne* on the current form. If the object doesn't exist, a critical error occurs, and control moves to the **onFail** clause. If the error code isn't peObjectNotFound, the method creates and logs a custom error.

```
; pushMessage::pushButton
method pushButton(var eventInfo Event)
var
  obj    UIObject
  eCode  LongInt
  eMsg   String
endVar
try
  obj.attach("boxOne")
  obj.color = "RedBlue"    ; invalid color constant--will cause an error
                           ; other than peObjectNotFound
onFail
  if errorCode() = peObjectNotFound then
    msgInfo("And the error was", errorMessage())
    obj.create(BoxTool, 180, 180, 360, 360)
    obj.name = "boxOne"
    obj.visible = Yes
    reTry
 else
   ; pop off the original error
   eCode = errorCode()
   eMsg = errorMessage()
   errorPop()
   ; push the original error back onto the stack, but
   ; modify the error message
   errorLog(eCode, self.Name + "::pushButton failed at " +
            String(time()) + ". " + eMsg)
   msgInfo("And the new error is", errorMessage())
   fail()
 endIf
endTry
endmethod
```

See also

- errorCode, errorMessage, errorPop, errorShow

System

errorMessage

System

Beginner

Procedure

Returns the text of the most recent error message.

Syntax

errorMessage ( ) String

Description

Returns a string containing the message displayed by the most recent run-time error or error condition. If no error occurred, **errorMessage** returns the empty string (""). This method is useful for logging error messages during a session.

Example

See the example for errorLog.

See also

- errorCode, errorLog, errorPop, errorShow

errorPop

System

Procedure

Removes the top layer of information from the error stack.

Syntax

errorPop ( ) Logical

Description

Removes the top layer (most recently added error code and error message) from the error stack, giving access to the layer below.

For more information about the error stack, refer to the *ObjectPAL Developer's Guide.*

Example

See the example for errorLog.

See also

- errorCode, errorLog, errorMessage, errorShow

errorShow

System

Procedure

Displays the error dialog box.

Syntax

errorShow ([const ***topHelp*** String **[,** const ***bottomHelp*** String **]])** Logical

Description

Opens the Error dialog box and displays the current error information. The string supplied in *topHelp* is used to label the top portion of the dialog box; *bottomHelp* labels the bottom portion of the dialog box.

For more information about the error stack, refer to the *ObjectPAL Developer's Guide.*

Example

In this example the *tryAnError* button logs several errors to the error stack, then displays them with **errorShow**.

```
; tryAnError::pushButton
method pushButton(var eventInfo Event)
; add two errors to the error stack
errorLog(1, "First error")
errorLog(2, "Second error")
; show the error dialog box (error 2 shows first)
errorShow("Title for top", "Title for bottom")
endmethod
```

See also

- errorCode, errorLog, errorMessage, errorPop

errorTrapOnWarnings

System

Beginner

Procedure

Specifies whether to handle warning errors as critical errors.

Syntax

errorTrapOnWarnings (const ***yesNo*** Logical)

Description

Specifies whether to handle warning errors as critical errors. By default, warning errors cannot be trapped in a **try...onFail** block. Specifying **errorTrapOnWarnings(Yes)** tells Paradox to trap warning errors as well as critical errors. For more information about warning errors and critical errors, refer to the *ObjectPAL Developer's Guide.*

Example

The following code attempts to open a bogus form: when **errorTrapOnWarnings** is set to No (the default), no error results from this attempt. Once **errorTrapOnWarnings** is set to Yes, the same code generates an error message.

```
; warningToError::pushButton
method pushButton(var eventInfo Event)
var
  someForm Form
endVar
someForm.open("someFile.fsl") ; attempt to attach to a nonexistent form
                              ; normally, this doesn't cause an error
errorTrapOnWarnings(Yes)      ; set the trap
someForm.open("someFile.fsl") ; this time, you get an error message
errorTrapOnWarnings(No)       ; restore to normal
endmethod
```

See also

- errorCode, errorLog, errorMessage, errorPop, errorShow

execute

Beginner

System

Procedure

Executes a DOS command.

Syntax

execute (const ***programName*** String [, const ***wait*** Logical [, const ***displayMode*** SmallInt]]) Logical

Description

Runs the DOS command specified in *dosCommand*. The optional argument *displayMode* specifies the video display mode to use when executing the command.

If the command is not in the user's path, you must specify the path in *dosCommand*. Use double backslashes in path names.

Example

The following example launches the Windows Clock application with the default window style and waits for it to return.

```
; showClock::pushButton
method pushButton(var eventInfo Event)
execute("clock.exe", Yes, WinStyleDefault)  ; execute Windows Clock
endmethod
```

See also

- play

exit

Beginner

System

Procedure

Closes Paradox and exits to Windows.

Syntax

exit ( )

Description

Closes Paradox and exits to Windows. If a Paradox application has changed, the user is prompted to save it.

Example

In this example, the **menuAction** method for the form traps for a MenuFileExit or MenuControlClose action. If the user attempts to exit Paradox, the method first asks for confirmation; if the user confirms, the method uses **exit** to close Paradox.

```
; thisForm::menuAction
method menuAction(var eventInfo MenuEvent)
if eventInfo.isPreFilter()
```

```
  then
    ; code here executes for each object in form
    ; trap for File|Exit and confirm before closing
    if eventInfo.id() = MenuFileExit OR
       eventInfo.id() = MenuControlClose  then
      if msgYesNoCancel("Exit", "Do you want to quit?") = "Yes" then
        exit()          ; leave
      else
        disableDefault  ; if user escapes, or chooses Cancel or No, then
                        ; block the action
      endIf
    endIf
  else
    ; code here executes just for form itself
  endif
endmethod
```

See also

- close

fail

System

Procedure

Causes a method to fail.

Syntax

fail ([const ***errorNumber*** SmallInt, const ***errorMessage*** String])

Description

Causes a method to fail. Executing this method in the **onFail** section of a **try...onFail** block forces a jump to the next highest block, if it exists, or to the implicit **try...onFail** block ObjectPAL wraps around every method. The optional argument *errorNumber* specifies an error code for the failure. The optional argument *errorMessage* specifes a message to display.

ObjectPAL provides constants for error codes; see Errors in the Constants dialog box.

For more information about using **try...onFail** blocks, refer to the *ObjectPAL Developer's Guide.*

Example

In this example, assume that you want to vary the position of a box called *rateBox.* The values of an unbound field object named *rateField* range from 1 to 10; the position of *rateBox* is determined by the value in *rateField.* The following code is attached to the **changeValue** method for *rateField.* This method uses a **try...fail** block to trap for and handle an improper data type entered in an unbound field.

```
; rateField::changeValue
method changeValue(var eventInfo ValueEvent)
Const
  baseXPosition = LongInt(3000)
  baseYPosition = LongInt(1000)
endConst
Var
```

```
  rateX   LongInt
endVar
try
  ; this if statement will fail if the field contents can't
  ; be compared to the integers 0 and 10 - for instance, if
  ; the user enters a string
  if eventInfo.newValue() = 0 AND eventInfo.newValue() <> 10 then
    rateX = (eventInfo.newValue() * 400) + baseXPosition
    rateBox.Position = Point(rateX, baseYPosition)
  else
    fail() ; if the value is a number but is out of range,
           ; call the fail block
  endif
onFail
  disableDefault
  eventInfo.setErrorCode(CanNotDepart)
  msgStop("Stop", "Rating should be a number between 0 and 10.")
endTry
endmethod
```

See also

- errorCode, errorMessage, errorShow

fileBrowser

System

Procedure

Displays the Browser and returns the names of one or more files selected by the user.

Syntax

1. **fileBrowser** (var ***selectedFile*** String
   [, var ***browserInfo*** FileBrowserInfo]) Logical
2. **fileBrowser** (var ***selectedFiles*** Array[] String
   [, var ***browserInfo*** FileBrowserInfo]) Logical

Description

Displays the Browser and returns the names of one or more files returned by the user. ObjectPAL execution suspends until the user closes the Browser. Use syntax 1 to return one file name in *selectedFile*, and use syntax 2 to return an array of file names in *selectedFiles*, where *selectedFiles* is declared as a resizeable array. For either syntax, you can provide a BrowserParms record containing information to pass to the Browser (see the second example, following).

Example

The first example calls **fileBrowser** twice: the first time, it returns one file name, and if it is the name of a table, opens a Table window. The second time, it returns an array of file names (selected by *Shift*-clicking) and displays the array in a dialog box.

```
; fileBrowserButton::pushButton
method pushButton(var eventInfo Event)
var
  oneFile            String
  manyFiles Array[] String
  tView              TableView
```

```
endVar
fileBrowser(oneFile)    ; display the File Browser, and wait
                        ; for the user to choose one file
                        ; variable oneFile stores the file name chosen
if isTable(oneFile) then
  tView.open(oneFile) ; open a Table window for the chosen file
endIf

fileBrowser(manyFiles) ; let the user select multiple files and store
                        ; the file names in an array
manyFiles.view()        ; displays the user's choices
endmethod
```

You can also pass a record as an argument to **fileBrowser** to specify what data the Browser displays. For example, you can make the Browser display Paradox tables only, or forms only, or forms and reports.

ObjectPAL provides a special data type, called FileBrowserInfo, that you can use *only* with the **fileBrowser** procedure. FileBrowserInfo is a predeclared record with the following structure:

```
x, y, w, h    SmallInt      ; size of Browser window in twips
WindowStyle   LongInt       ; window style (see WindowStyle constants)
AllowableTypes LongInt      ; file type (see following table)
SelectedType  LongInt       ; one of the AllowableTypes
FileFilters   String        ; the filespec in edit box
Alias         String        ; alias or drive name
Path          String        ; path relative to Alias
```

This record structure is predefined and built into ObjectPAL, so all you have to do is declare a variable of type FileBrowserInfo and assign values to its fields, as shown in the example—you don't have to declare the type yourself each time you want to use it.

After the call to **fileBrowser**, the Alias, Path, and FileFilter fields are filled in with the values that were in the Browser dialog box. In other words, you can find out what the user entered in those areas of the Browser.

The AllowableTypes field specifies what appears in the drop-down edit list for the Types panel in the Browser. The SelectedType field indicates which of the AllowableTypes is currently selected. The following table lists valid ObjectPAL constants to use in the SelectedType and AllowableTypes fields. These constants also are listed under FileBrowserFileTypes in the Constants dialog.

Constant	Extension	Description
fbFiles	*.*	All files
fbTable	*.db	Paradox tables
fbQuery	*.qbe	QBE files
fbForm	*.fsl, *.fdl	Paradox forms
fbReport	*.rsl, *.rdl	Paradox reports
fbScript	*.ssl, *.sdl	Paradox scripts

System

Constant	Extension	Description
fbFiles	*.*	All files
fbGraphic	*.bmp	Bitmap graphics
fbText	*.txt	Text files
fbSQL	*.sql	SQL files
fbAllTables	*.db, *.dbf	User and system tables
fbTableView	*.tv	Table view files
fbParadox	*.db	Paradox tables
fbDBase	*.dbf	dBASE tables
fbASCII	*.txt	Text files
fbQuattroProWindows	*.wt1	Quattro Pro for Windows worksheets
fbQuattroPro	*.wq1	Quattro Pro for DOS worksheets
fbQuattro	*.wkq	Quattro worksheets
fbLotus2	*.wk1	Lotus worksheets (version 2)
fbLotus1	*.wks	Lotus worksheets (version 1)
fbExcel	*.xls	Excel worksheets
fbConfig	*.cfg	Configuration files
fbLibrary	*.lsl	Paradox libraries

The **fileBrowser** procedure looks only at the names given in the structure. You can pass a different record structure to it and it finds the fields with the appropriate names and uses them. In other words, you can define a simpler record structure with only the items you are interested in.

Example

The following code is attached to a button's built-in **pushButton** method. When it executes, it invokes the Browser and waits for the user to choose a file. Then, it displays information about the user's choice in the status area.

```
method pushButton(var eventInfo Event)

var
  fbi FileBrowserInfo ; Declare a variable that uses the predefined
                      ; FileBrowserInfo record structure
  selectedFile String
endVar

; The following statements assign values to fields in the
; record of file browser information
fbi.Alias = "WORK" ; Search the current working directory
fbi.AllowableTypes = fbTable + fbForm ; Search for tables and forms
```

```
; Display the Browser and process the user's selection
if fileBrowser(selectedFile, fbi) then
  message("You selected ", selectedFile," with the path ", fbi.path)
else
  message("You selected cancel")
endif

endMethod
```

See also

- The FileSystem type

formatAdd

System

Procedure

Adds a format.

Syntax

formatAdd (const ***formatName*** String, const ***formatSpec*** String)
Logical

Description

Creates the format described by *formatSpec* and named *formatName*. The format created is available to the current session.

Note Field width (W*n*), alignment (AR, AL, AC), and case specifiers (CU, CL, CC) are not saved with a new format definition; but decimal precision (W.*n*) is saved. See **format** in the String type for a complete description of format specifiers.

Example

The following example adds a new format specification to the session, then sets the default Currency format to the new format.

```
; addAFormat::pushButton
method pushButton(var eventInfo Event)
var
  someNum Currency
endVar
; first, add a currency format with 4 decimal digits and
; a floating dollar sign (windows dollar sign)
formatAdd("FourCurrency", "W.4, E$W")
; then, set the default format for Currency to the new format
formatSetCurrencyDefault("FourCurrency")
someNum = 41324.09876
someNum.view()                  ; appears as $41,324.0988
endmethod
```

See also

- formatDelete, formatExist

formatDelete

System

Procedure

Deletes a format.

System

Syntax **formatDelete** (const ***formatName*** String) Logical

Description Deletes the named format *formatName* from the current session.

Example This example deletes the custom format named *FourCurrency*, if it exists.

```
; deleteAFormat::pushButton
method pushButton(var eventInfo Event)
if formatExist("FourCurrency") then
  formatDelete("FourCurrency")
else
  msgInfo("FYI", "Format was not found.")
endif
endmethod
```

See also

- formatAdd, formatExist

formatExist

System

Procedure Reports whether a format exists.

Syntax **formatExist** (const ***formatName*** String) Logical

Description Checks if the format named *formatName* is available for the current session. The method returns True if the format is available; otherwise, it returns False.

Example In this example, the method checks if a custom format named *FourCurrency* exists; if not, the method adds the new format and displays a number formatted as *FourCurrency*.

```
; addCurrFormatExist::pushButton
method pushButton(var eventInfo Event)
var
  someNum Currency
endVar
; check if custom format exists already
if NOT formatExist("FourCurrency") then
  ; if not, add a currency format with 4 decimal digits and
  ; a floating dollar sign (windows dollar sign)
  msgInfo("FYI", "Format does not exist. Adding it now.")
  formatAdd("FourCurrency", "W.4, E$W")
else
  msgInfo("FYI", "Format already exists.")
endif
; set the default format for Currency to the new format
formatSetCurrencyDefault("FourCurrency")
someNum = 41324.09876
someNum.view()          ; displays number as $41324.0988, because
                        ; someNum is a variable of Currency type
endmethod
```

See also ❐ formatAdd, formatDelete

formatSetCurrencyDefault

System

Procedure Sets the default display format for Currency values.

Syntax **formatSetCurrencyDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Currency values. This setting remains in effect for the duration of the session.

Example See the example for formatExist.

See also ❐ formatSetNumberDefault

formatSetDateDefault

System

Procedure Sets the default display format for Date values.

Syntax **formatSetDateDefault** (const ***formatName*** String) Logical

Description Sets the default format for displaying Date values. This setting remains in effect for the duration of the session.

Example In the following example, the **pushButton** method for the *setDateFormat* button sets the default display format for Date values to the Windows Long format. The method then uses **view** to display a date; the date is shown in the new default format.

```
; setDateFormat::pushButton
method pushButton(var eventInfo Event)
var
  someDate Date
endVar
if formatExist("Windows Long") then
  formatSetDateDefault("Windows Long")
  someDate = date("9/15/92")
  someDate.view()            ; displays "Tuesday, September 15, 1992"
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See also ❐ formatSetTimeDefault, formatSetDateTimeDefault

formatSetDateTimeDefault

System

Procedure

Sets the default display format for DateTime values.

Syntax

formatSetDateTimeDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying DateTime values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setDateTimeFormat* button sets the default display format for DateTime values. The method then uses **view** to display a DateTime value; the value is shown in the new default format.

```
;setDateTimeFormat::pushButton
method pushButton(var eventInfo Event)
var
  someDateTime DateTime
endVar
if formatExist("h:m:s am m/d/y") then
  formatSetDateTimeDefault("h:m:s am m/d/y")
  someDateTime = DateTime("11:45:25 am 11/24/61")
  someDateTime.view()              ; displays 11:45:25 am 11/24/61
else
  msgInfo("Status", "Requested format does not exist.")
endif
endmethod
```

See also

- formatSetDateDefault, formSetTimeDefault

formatSetLogicalDefault

System

Procedure

Sets the default display format for Logical values.

Syntax

formatSetLogicalDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying Logical values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setLogicalFormat* button sets the default display format for Logical values to the Male/Female format. The method then uses **view** to display a logical value; the value is shown in the new default format.

```
; setLogicalFormat::pushButton
method pushButton(var eventInfo Event)
var
  someLogical  Logical
```

```
endVar
if formatExist("Male/Female") then
  formatSetLogicalDefault("Male/Female")
  someLogical = True
  someLogical.view()              ; displays Male
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See also

- formatAdd

formatSetLongIntDefault

System

Procedure

Sets the default display format for LongInt values.

Syntax

formatSetLongIntDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying LongInt values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setIntegerFormat* button sets the default display format for LongInt values to the Integer format. The method then uses **view** to display a long integer; the value is shown in the new default format.

```
; setIntegerFormat::pushButton
method pushButton(var eventInfo Event)
var
  someInt  LongInt
endVar
if formatExist("Integer") then
  formatSetLongIntDefault("Integer")
  someInt = 238756
  someInt.view()                ; displays 238756
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See also

- formatSetSmallIntDefault

formatSetNumberDefault

System

Procedure

Sets the default display format for Number values.

Syntax

formatSetNumberDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying Number values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setNumberFormat* button sets the default display format for Number values to the Scientific format. The method then uses **view** to display a number; the value is shown in the new default format.

```
; setNumberFormat::pushButton
method pushButton(var eventInfo Event)
var
  someNum Number
endVar
if formatExist("Scientific") then
  formatSetNumberDefault("Scientific")
  someNum = 3489.283
  someNum.view()                    ; displays 3.4893+e3
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See also

- formatSetCurrencyDefault

---

formatSetSmallIntDefault

System

Procedure

Sets the default display format for SmallInt values.

Syntax

formatSetSmallIntDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying SmallInt values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setSmallIntFormat* button sets the default display format for SmallInt values to the Integer format. The method then uses **view** to display a small integer; the value is shown in the new default format.

```
; setSmallIntFormat::pushButton
method pushButton(var eventInfo Event)
var
  someInt  SmallInt
endVar
if formatExist("Integer") then
  formatSetSmallIntDefault("Integer")
  someInt = 324
  someInt.view()                    ; displays 324
else
  msgStop("Stop", "Requested format does not exist.")
endif
endmethod
```

See also ❐ formatSetLongIntDefault

formatSetTimeDefault — System

Procedure

Sets the default display format for Time values.

Syntax

formatSetTimeDefault (const ***formatName*** String) Logical

Description

Sets the default format for displaying Time values. This setting remains in effect for the duration of the session.

Example

In the following example, the **pushButton** method for the *setTimeFormat* button sets the default display format for Time values to the format *hh:mm:ss am*. The method then uses **view** to display a time; the value is shown in the new default format.

```
; setTimeFormat::pushButton
method pushButton(var eventInfo Event)
var
  someTime Time
  someStr  String
endVar
if formatExist("hh:mm:ss am") then
  formatSetTimeDefault("hh:mm:ss am")
  someTime = time("12:22:45 pm")
  someTime.view()                ; displays 12:22:45 pm
else
  msgInfo("Status", "Requested format does not exist.")
endif
endmethod
```

See also

❐ formatSetDateDefault, formatSetDateTimeDefault

getMouseScreenPosition — System

Procedure

Returns the mouse position as a Point.

Syntax

getMouseScreenPosition ( ) Point

Description

Returns the coordinates (in twips) of the mouse pointer relative to the screen, not the Desktop. Use Point type methods (for example **getX** and **getY**) to get more information.

This method gets the mouse position at the time of the event; the current mouse position may be different.

Example

In this example, when the user clicks the *nervousMouse* button, the mouse jumps one inch down and one inch to the left.

```
; nervousMouse::pushButton
method pushButton(var eventInfo Event)
var
  mouseP,
  newMouseP Point
endVar
mouseP = getMouseScreenPosition()
newMouseP = mouseP + Point(1440, 1440)
setMouseScreenPosition(newMouseP)    ; move mouse pointer 1 inch down and
                                     ; 1 inch to the right
endmethod
```

See also

- setMouseScreenPosition

helpOnHelp

System

Procedure

Displays information about how to use the Help system.

Syntax

helpOnHelp ( ) Logical

Description

Displays information explaining how to use the Windows Help application, opening the application if necessary.

Example

See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also

- helpShowContext, helpShowIndex

helpQuit

System

Procedure

Tells the Help application it is no longer needed.

Syntax

helpQuit (const ***helpFileName*** String) Logical

Description

Tells the Help application that Help is no longer needed. If no other applications have asked for Help, Windows closes the Help application.

Example

See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also ❐ helpOnHelp

helpSetIndex

System

Procedure Sets the Help index.

Syntax **helpSetIndex** (const ***helpFileName*** String, const ***indexId*** LongInt) Logical

Description Tells the Windows Help application to set the current index to *indexID* in *helpFileName*.

Example See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also ❐ helpShowIndex

helpShowContext

System

Procedure Displays help for a particular context.

Syntax **helpShowContext** (const ***helpFileName*** String, const ***helpId*** LongInt) Logical

Description Tells the Windows Help application to search *helpFileName* for *helpId* and display the associated help.

Example See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also ❐ helpShowIndex, helpShowTopic

helpShowIndex

System

Procedure Displays the index of a specifed Help file.

Syntax **helpShowIndex** (const ***helpFileName*** String) Logical

Description Displays the index to the file *helpFileName*.

Example See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also

- helpSetIndex, helpShowContext, helpShowTopic

helpShowTopic

System

Procedure Displays Help for a specified topic.

Syntax **helpShowTopic** (const ***helpFileName*** String, const ***topicKey*** String) Logical

Description Tells the Windows Help application to search *helpFileName* for *topicKey* and display the associated Help.

Example See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also

- helpShowContext, helpShowIndex, helpShowTopicInKeywordTable

helpShowTopicInKeywordTable

System

Procedure Displays Help for a topic identified by a keyword in an alternate keyword table.

Syntax **helpShowTopicInKeywordTable** (const ***helpFileName*** String, const ***keyTableLetter*** String, const ***topicKey*** String) Logical

Description Tells the Windows Help application to search the file *helpFileName* for *keyTableLetter* and *topicKey* and display the associated Help.

Example See the MAST application in the DIVEPLAN directory for examples of using a Windows Help application with ObjectPAL.

See also

- helpShowContext, helpShowIndex, helpShowTopic

message

System

Beginner

Procedure

Displays a message in the status line.

Syntax

message (const ***message*** String [, const ***message*** String]*)

Description

Displays a message composed of up to six strings in the status line.

Example

The following method writes a message to the status line:

```
; showMessage::pushButton
method pushButton(var eventInfo Event)
var
  lastName, firstName String
endVar
lastName = "Borland"
firstName = "Frank"
message("Hello, my name is ", firstName, " ", lastName, ".")
endmethod
```

See also

- msgInfo, msgQuestion, msgStop

msgAbortRetryIgnore

System

Procedure

Displays a dialog box containing a message and three buttons: Yes, No, and Ignore.

Syntax

msgAbortRetryIgnore (const ***caption*** String, const ***text*** String)
String

Description

Displays a three-button dialog box where *caption* specifies the text in the title bar, and *text* specifies the message displayed to the user. The return value is a string corresponding to the button clicked: "Abort", "Retry", or "Ignore". If the user presses *Esc* or clicks the Close box, the return value is "Cancel".

Example

In this example, the *showAbortRetryIgnore* button warns the user that an operation may take a long time, and asks the user whether to Abort, Retry, or Ignore.

```
; showAbortRetryIgnore::pushButton
method pushButton(var eventInfo Event)
var
  doThis String
endVar
doThis = msgAbortRetryIgnore("Note", "This may take a long time.
Do you want to stop?")  ; this message spans 2 lines
```

```
doThis.view()
; execute a custom method based on the user's choice
switch
   case doThis = "Abort"  : message("Aborting operation.")
   case doThis = "Retry"  : message("Retrying operation.")
   case doThis = "Ignore" : message("Ignoring problem.")
   case doThis = "Cancel" : message("Cancelling operation.")
endSwitch
endmethod
```

See also

- msgRetryCancel, msgYesNoCancel

---

msgInfo

Beginner

System

Procedure

Displays a dialog box containing the information icon, a message, and an OK button.

Syntax

msgInfo (const ***caption*** String, const ***text*** String)

Description

Displays a one-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself. The user must click OK or press *Esc* to close the box. This method does not return a value.

Example

This example uses the **msgInfo** method to display a fact to the user.

```
; showMsgInfo::pushButton
method pushButton(var eventInfo Event)
msgInfo("Trivia", "The capital of Oregon is Salem.")
endmethod
```

See also

- message, msgQuestion, msgStop

---

msgQuestion

Beginner

System

Procedure

Displays a dialog box containing a message, a question mark icon, a Yes button, and a No button.

Syntax

msgQuestion (const ***caption*** String, const ***text*** String) String

Description

Displays a two-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself. The return value corresponds to the button the user clicks to close the dialog box: "Yes" or "No". If the user presses *Esc* or clicks the Close box, the return value is "Cancel".

Example

The following code asks the user whether to change the desktop title. If the user presses the Yes button, the desktop title is changed, then restored.

```
; showMsgQuestion::pushButton
method pushButton(var eventInfo Event)
var
  userChoice String
  thisApp    Application
endVar
userChoice = msgQuestion("Confirm", "Are you sure you want to
change the title to 'Custom Application'?")
switch
   case userChoice = "Yes" :
     thisApp.setTitle("Custom Application")   ; change the desktop title
     sleep(2000)                              ; pause
     thisApp.setTitle("Paradox for Windows")  ; restore it
   case userChoice = "No"  :
     message("Application title not changed.")
endSwitch
endmethod
```

See also

- msgInfo, msgStop

msgRetryCancel

System

Procedure

Displays a dialog box containing a message and two buttons: Retry and Cancel.

Syntax

msgRetryCancel (const ***caption*** String, const ***text*** String) String

Description

Displays a two-button dialog box where *caption* specifies the text in the dialog box title bar, and *text* specifies a message to display to the user. The return value corresponds to the button the user clicks: "Retry" or "Cancel". If the user presses *Esc* or clicks the Close box, the return value is "Cancel".

Example

The following example poses a question to the user in response to a problem, then displays a message on the status line depending on how the user answers.

```
; showMsgRetryCancel::pushButton
method pushButton(var eventInfo Event)
var
  confirm String
endVar
confirm = msgRetryCancel("Dilemma", "What will you do?")
switch
  case confirm = "Retry"  : message("Retrying.")
  case confirm = "Cancel" : message("Giving up.")
endSwitch
endmethod
```

See also

❒ msgAbortRetryIgnore, msgYesNoCancel

msgStop

System

Procedure

Displays a dialog box containing a stop sign icon, a message, and an OK button.

Syntax

msgStop (const ***caption*** String, const ***text*** String)

Description

Displays a one-button dialog box. The text in *caption* is displayed in the title bar, and *text* is displayed in the box itself, along with a Stop icon. The user must click OK or press *Esc* to close the box. This method does not return a value.

Example

In this example, the **pushButton** method for *showMsgStop* alerts the user of a potentially dangerous action.

```
; showMsgStop::pushButton
method pushButton(var eventInfo Event)
msgStop("Stop!", "If you do that, changes to the form will not be saved.")
endmethod
```

See also

❒ msgInfo, msgQuestion

msgYesNoCancel

System

Procedure

Displays a dialog box containing a message and three buttons: Yes, No, and Cancel.

Syntax

msgYesNoCancel (const ***caption*** String, const ***text*** String) String

Description

Displays a three-button dialog box where *caption* specifies the text in the dialog box title bar, and *text* specifies text to display to the user. The return value corresponds to the button the user clicks: "Yes", "No", or "Cancel". If the user presses *Esc* or clicks the Close box, the return value is "Cancel".

Example

In this example, **msgYesNoCancel** is used to find out whether the user wants to save data before quitting, discard the data, or cancel the quit operation and stay in the application.

```
; showMsgYesNoCancel::pushButton
method pushButton(var eventInfo Event)
var
```

```
  theChoice String
endVar
theChoice = msgYesNoCancel("Quit", "Save data before quitting?")
switch
   case theChoice = "Yes"    : message("Saving data.")
   case theChoice = "No"     : message("Discarding data.")
   case theChoice = "Cancel" : message("Remaining in application.")
endSwitch
endmethod
```

See also

- msgAbortRetryIgnore, msgRetryCancel

pixelsToTwips

System

Procedure

Converts screen coordinates from pixels to twips.

Syntax

pixelsToTwips (const ***pixels*** Point) Point

Description

Converts the screen coordinates specified in *pixels* from pixels to twips. A pixel (the name comes from *pic*ture *el*ement) is a dot on the screen; a twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).

Example

The following example shows the position of the button (*self*) first in twips, then in pixels. The method goes on to find the display resolution for the system (given in pixels), then uses that information to open a window in the center of the display.

```
; convertTwipsPixels::pushButton
method pushButton(var eventInfo Event)
var
  selfP,
  sysTwips  Point
  thisSys   DynArray[] AnyType
  x, y      SmallInt
  custForm  Form
endVar
selfP = self.Position
selfP.view("Position of this button in twips")
selfP = twipsToPixels(selfP)
selfP.view("Position of this button in pixels")
; open a 2" by 2" form exactly in the center of the screen
sysInfo(thisSys)              ; fill a dynamic array with system information
sysTwips = Point(thisSys["FullWidth"], thisSys["FullHeight"])
sysTwips = pixelsToTwips(sysTwips)
x = int(sysTwips.x()/2) - 1440  ; calculate x-coordinate 1 inch left of center
y = int(sysTwips.y()/2) - 1440  ; calculate y-coordinate 1 inch above center
custForm.open("Customer.fsl", WinStyleDefault, x, y, 2880, 2880)

endmethod
```

See also

- twipsToPixels

play

Beginner

System

Procedure

Plays a standalone script.

Syntax

play (const ***scriptName*** String) AnyType

Description

Executes the statements in the script specified in *scriptName*. A standalone script consists of one or more ObjectPAL statements attached to an object that never appears. You can think of a standalone script as a special kind of custom method. For more information, refer to the *ObjectPAL Developer's Guide.*

Example

The following code plays a script named TESTSCR.SSL, which is assumed to be in the working directory.

```
; playAScript::pushButton
method pushButton(var eventInfo Event)
play("Testscr.ssl")
endmethod
```

See also

- execute

readEnvironmentString

System

Procedure

Reads an item from the DOS environment.

Syntax

readEnvironmentString (const ***key*** String) String

Description

Returns a string containing information about the DOS environment variable specified in *key*. Environment variables are assigned values using the DOS command SET. They control how DOS and some batch files and programs appear and work. Commonly used environment variables include PATH, PROMPT, and COMSPEC. For more information, consult your DOS manuals, especially the SET command.

Example

This example uses **readEnvironmentString** and **writeEnvironmentString** to get and change the value of the PATH environment variable.

```
; changeEnvironmentStr::pushButton
method pushButton(var eventInfo Event)
var
   fs            FileSystem
   thePath, myDir  String
```

```
   pathArr Array[] String
endVar
; fs.getDir() currently returns some high-ANSI char--not a meaningful string
myDir = fs.getDir()                    ; get the current directory
myDir.view("Current directory")
thePath = readEnvironmentString("PATH") ; read the path environment var
thePath.breakApart(";", pathArr)       ; break on semicolon
pathArr.view("An array of paths")      ; view the results
if NOT pathArr.contains(myDir) then    ; if current dir not in path
   msgInfo("FYI", "Adding current directory to path.")
   writeEnvironmentString("PATH", thePath + ";" + myDir)  ; add it
endif
thePath = readEnvironmentString("PATH") ; read the changed environment var
thePath.view()
thePath.breakApart(";", pathArr)       ; break it up
pathArr.view("An array of paths")      ; view the results
endmethod
```

See also

- readProfileString, writeEnvironmentString

readProfileString

System

Procedure

Reads an item from an initial settings file.

Syntax

readProfileString (const ***fileName*** String, const ***section*** String, const ***key*** String) String

Description

Returns a value from a specified section of a specified file on the user's system. This method searches the user's WINDOWS directory by default. Typically, you use this method to read the user's WIN.INI file, so *fileName* is WIN.INI.

A section in WIN.INI is marked by a string bounded by square brackets on a line by itself (for example, [windows]). However, omit the brackets when you specify *section*; that is, to specify the [windows] section, use "windows."

Within a section, a value marker is a string followed by an equal sign (for example, Beep =), but don't include the = when you specify *key*.

Example

This example uses **readProfileString** and **writeProfileString** to get and change the setting for the Windows beep.

```
; changeProfileStr::pushButton
method pushButton(var eventInfo Event)
var
   myBeep String
   winDir String
endVar
winDir = windowsDir()
myBeep = readProfileString(winDir + "\\win.ini", "windows", "Beep")
msgInfo("Beep?", myBeep) ; displays yes or no, depending on user's settings
if myBeep <>"yes" then
```

```
  msgInfo("Alert", "Changing profile string for Beep to yes.")
  writeProfileString(winDir + "\\win.ini", "windows", "Beep", "yes")
  beep()
else
  msgInfo("Alert", "Changing profile string for Beep to no.")
  writeProfileString(winDir + "\\win.ini", "windows", "Beep", "no")
  beep()
endIf
endmethod
```

See also

- readEnvironmentString, writeProfileString

---

setMouseScreenPosition

System

Procedure

Displays the pointer at a specified position.

Syntax

1. setMouseScreenPosition (const ***mousePosition*** Point)

2. setMouseScreenPosition (const ***x*** LongInt, const ***y*** LongInt)

Description

Displays the pointer at the point specified in *mousePosition* (syntax 1) or at the coordinates specified in *x* and *y* (syntax 2). Coordinates should be specified in twips.

Example

See the example for getMouseScreenPosition.

See also

- getMouseScreenPosition

---

setMouseShape

System

Procedure

Specifies the shape of the pointer.

Syntax

setMouseShape (const ***mouseShapeId*** LongInt) LongInt

Description

Specifies in *mouseShapeId* the shape of the pointer. ObjectPAL provides constants for *mouseShapeId;* see MouseShapes in the Constants dialog box.

Example

This example cycles through the available mouse shapes when the user clicks a field.

```
; changeMouseField::mouseUp
method mouseUp(var eventInfo MouseEvent)
beep()
message("Watch carefully--changing mouse shapes.")
setMouseShape(MouseIBeam)
```

```
sleep(1000)
setMouseShape(MouseCross)
sleep(1000)
setMouseShape(MouseWait)
sleep(1000)
setMouseShape(MouseUpArrow)
sleep(1000)
setMouseShape(MouseArrow)
endmethod
```

See also

- getMouseScreenPosition, setMouseScreenPosition

---

sleep

Beginner

System

Procedure

Produces a delay of a specified duration.

Syntax

sleep ([const ***numberOfMilliSeconds*** LongInt])

Description

Suspends execution of a method for the interval specified in *numberOfMilliSeconds*. Programs other than Paradox continue to execute. Printing continues, too. The Paradox Desktop, including TimerEvents, is suspended for the duration of the sleep.

Note When called with no argument, **sleep** causes the current process (method) to release control of the processor for an instant so that other processes can complete their tasks. This is called a *yield*. The Windows environment occasionally requires a yield before it will finish a simultaneous (time-sliced) process.

Example

The following code displays a message in the status line, then waits five seconds before displaying a second message.

```
; goToSleep::pushButton
method pushButton(var eventInfo Event)
var
   yourTurn SmallInt
endVar
yourTurn = 5000
beep()
message("Next message in 5 seconds.")
sleep(yourTurn)                    ; waits for 5 seconds
message("5 seconds have elapsed.")
endmethod
```

See also

- wait in the Form type

System

sound

System

Beginner

Procedure

Creates a sound of specified frequency and duration.

Syntax

sound (const ***freqHertz*** LongInt, const ***durationMilliSecs*** LongInt)

Description

Creates a sound of frequency (in Hertz) *freqHertz* for a time (in milliseconds) specified in *durationMilliSecs.* Frequency values can range between 1 and 50,000 Hertz (the audible limit is approximately 20,000 Hertz).

Example

In this example, the **pushButton** method for *makeMusic* first declares a number of constants for frequency values in a scale. These notes are used to specify the *frequency* argument in the calls to the **sound** method. After playing a few bars from a tune, the method demonstrates the calculation for notes in a chromatic scale (proceeds by half notes).

```
; makeMusic::pushButton
method pushButton(var eventInfo Event)
var
   quarterNote, octave, note LongInt
   power                     Number
endVar
; frequency values for notes in a scale
const
  noteA1  = 110
  noteA#1 = 116
  noteB1  = 123
  noteC1  = 130
  noteC#1 = 138
  noteD1  = 146
  noteD#1 = 155
  noteE1  = 164
  noteF1  = 174
  noteF#1 = 184
  noteG1  = 195
  noteG#1 = 207
  noteA2  = 220
  noteA#2 = 234
  noteB2  = 249
  noteC2  = 265
  noteC#2 = 282
  noteD2  = 300
endConst
; several bars from Peter and the Wolf
sound(noteA1, 200)
sound(noteD1, 150)
sound(noteF#1, 50)
sound(noteA2, 100)
sound(noteB2, 100)
sound(noteA2, 150)

sound(noteF#1, 50)
sound(noteA2, 100)
sound(noteB2, 100)
```

```
sound(noteC#2, 150)
sound(noteD2, 50)
sound(noteA2, 100)
sound(noteF#1, 100)
sound(noteD1, 100)
sleep(1000)

; play a few chromatic scales
quarterNote = 120
for octave from 0 to 1
   for note from 0 to 11
      sound(int(pow(2, octave + note / 12.0) * 110), quarterNote)
   endFor
endFor
sound(int(pow(2, 2) * 110), quarterNote) ; finish out the scale
endmethod
```

See also

- beep

sysInfo

System

Procedure

Creates a dynamic array of information about the system running Paradox.

Syntax

sysInfo (var ***info*** DynArray[] AnyType)

Description

Creates a dynamic array *info* of information about the system running Paradox. You must declare the DynArray before calling this method. The DynArray contains indexes for system attributes and their values. The indexes are described in the following table.

Index	Definition
AreMouseButtonsSwapped	Reports whether functions of the left and right mouse buttons are reversed
CodePage	Reports which code page is currently loaded by Windows
CPU	Processor type
FullHeight	Vertical working area (in pixels) in a maximized window
FullWidth	Horizontal working area (in pixels) in a maximized window
IconHeight	Height of icons (in pixels)
IconWidth	Width of icons (in pixels)
KeyboardFNKeys	Number of function keys
KeyboardSubType	Keyboard subtype is an OEM-dependent value
KeyboardType	Type and manufacturer of the keyboard
MathCoprocessor	Reports whether a math coprocessor is present

Index	Definition
Memory	Available memory, including swap file (if present) in bytes
Mouse	The number of mice attached to the system
NumTasks	The number of active tasks (programs)
Protected	Reports whether the system is running in protected mode
ScreenHeight	Total height of screen (in pixels)
ScreenWidth	Total width of screen (in pixels)
WindowsDir	Path to the WINDOWS directory
WindowsSystemDir	Path to the WINDOWS\SYSTEM directory
WindowsVersion	Version number of Windows

Example

The following code writes system information to a dynamic array named *userSys*, then displays *userSys* in a View dialog box. (See also the example for pixelsToTwips.)

```
; showSysInfo::pushButton
method pushButton(var eventInfo Event)
var
   userSys DynArray[] AnyType
endVar
sysInfo(userSys)    ; fill the array with system information
userSys.view()      ; show the array
endmethod
```

See also

- readProfileString

tracerClear

System

Procedure

Clears the Tracer window.

Syntax

tracerClear ( )

Description

Empties the Tracer window. The Tracer window can be opened by the **tracerOn** procedure at run time, or by selecting the ObjectPAL Editor menu's Debug | Trace Execution command.

Example

The following code clears the Tracer window. For this example, assume that the Tracer window is open and contains information.

```
; wipeTracer::pushButton
method pushButton(var eventInfo Event)
tracerClear()                  ; clear the Tracer window
endmethod
```

See also

❒ tracerHide, tracerOn

tracerHide

System

Procedure

Hides the Tracer window.

Syntax

tracerHide ( )

Description

Makes the Tracer window invisible, but it does not clear or close the Tracer window. To make the Tracer window visible again, use **tracerShow**.

Example

The following code hides the Tracer window, pauses, then displays it again. For this example, assume that the Tracer window is already open.

```
; toggleTracerWin::pushButton
method pushButton(var eventInfo Event)
tracerHide()                    ; make the Tracer window invisible
message("Hiding Tracer window. Pausing...")
sleep(2000)
message("Showing Tracer window.")
tracerShow()                    ; make the Tracer window visible again
tracerToTop()                   ; bring it to the top
endmethod
```

See also

❒ tracerOn, tracerShow

tracerOff

System

Procedure

Closes the Tracer window.

Syntax

tracerOff ( )

Description

Stops writing code traces to the Tracer window. You can resume tracing code with the **tracerOn** procedure. Tracing is on by default when the Tracer window is first opened.

Example

This code turns off code tracing.

```
; stopTracer::pushButton
method pushButton(var eventInfo Event)
tracerOff()                     ; close the Tracer window
endmethod
```

System

See also

- tracerOn, tracerSave

tracerOn

System

Procedure

Activates code tracing.

Syntax

tracerOn ( )

Description

Resumes writing code traces to the Tracer window.

Example

This example reactivates code tracing.

```
; startTracer::pushButton
method pushButton(var eventInfo Event)
tracerOn()                      ; reactivate the Tracer window
endmethod
```

See also

- tracerOff, tracerShow

tracerSave

System

Procedure

Saves the contents of the Tracer window to a file.

Syntax

tracerSave (const **fileName** String)

Description

Saves the contents of the Tracer window to *fileName*.

Example

This example saves the contents of the Tracer window to a file named MYTRACE.TXT.

```
; saveTracerToFile::pushButton
method pushButton(var eventInfo Event)
tracerSave("mytrace.txt")       ; save the Tracer window to a file
endmethod
```

See also

- tracerWrite

tracerShow

System

Procedure

Makes the Tracer window visible.

Syntax **tracerShow ( )**

Description Makes the Tracer window visible. You can make the Tracer window invisible with the **tracerHide** procedure.

Example See the example for tracerHide.

See also ❐ tracerHide, tracerToTop

tracerToTop

System

Procedure Makes the Tracer window the topmost window.

Syntax **tracerToTop ( )**

Description Brings the Tracer window to the top of the desktop.

Example See the example for tracerWrite.

See also ❐ tracerHide, tracerShow

tracerWrite

System

Procedure Writes a message to the Tracer window.

Syntax **tracerWrite** (const ***message*** String [, const ***message*** String]*)

Description Writes a message to the Tracer window.

Example The following code logs a message to the Tracer window, then brings the Tracer window to the top layer of the desktop.

```
; logTracerMsg::pushButton
method pushButton(var eventInfo Event)
tracerWrite("Tracer hit by " + String(self.Name) +
            " at " + String(time()))              ; log a message
tracerToTop()            ; make the Tracer window the topmost window
endmethod
```

See also ❐ tracerSave

System

twipsToPixels

System

Procedure

Converts screen coordinates from twips to pixels.

Syntax

twipsToPixels (const ***twips*** Point) Point

Description

Converts the screen coordinates specified in *twips* from twips to pixels. A pixel (the name comes from *pic*ture *el*ement) is a dot on the screen; the number of pixels per inch is device-dependent. A twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).

Example

See the example for pixelsToTwips.

See also

- pixelsToTwips

version

System

Procedure

Returns the Paradox version number.

Syntax

version ( ) String

Description

Returns a string containing the version number of Paradox currently being used.

Example

In this example, the **pushButton** method for *showVersion* tells the user which Paradox version is in use.

```
; showVersion::pushButton
method pushButton(var eventInfo Event)
msgInfo("FYI", "You are running version " + String(version()) + ".")
endmethod
```

See alsoes

- sysInfo

winGetMessageID

System

Procedure

Returns the ID of a Windows message.

Syntax

winGetMessageID (const ***msgName*** SmallInt)

Description

Gets the ID of the message **msgName**. This method should be used by Windows programmers only. See your Windows programming documentation for more information.

See also

- winPostMessage, winSendMessage
- data, id in the MenuEvent type
- windowClientHandle, windowHandle in the Form type

winPostMessage

System

Procedure

Posts a message to Windows.

Syntax

winPostMessage (const ***hWnd*** SmallInt, const ***msg*** SmallInt, const ***wParam*** SmallInt, const ***lParam*** LongInt) Logical

Description

Posts a message to Windows. This method should be used by Windows programmers only. See your Windows programming documentation for more information.

See also

- winGetMessageID, winSendMessage
- data, id in the MenuEvent type
- windowClientHandle, windowHandle in the Form type

winSendMessage

System

Procedure

Sends a message to Windows.

Syntax

winSendMessage (const ***hWnd*** SmallInt, const ***msg*** SmallInt, const ***wParam*** SmallInt, const ***lParam*** LongInt) Logical

Description

Sends a message to Windows. This method should be used by Windows programmers only. See your Windows programming documentation for more information.

See also

- winGetMessageID, winPostMessage
- data, id in the MenuEvent type
- windowClientHandle, windowHandle in the Form type

writeEnvironmentString

System

Procedure

Writes information about the user's system environment to a file.

Syntax

writeEnvironmentString (const ***key*** String, const ***value*** String) Logical

Description

Sets a DOS environment variable. Environment variables are assigned values using the DOS command SET. They control how DOS and some batch files and programs appear and work. Commonly used environment variables include PATH, PROMPT, and COMSPEC. For more information, consult your DOS manuals, especially the SET command.

Example

See the example for readEnvironmentString.

See also

- readEnvironmentString, writeProfileString

writeProfileString

System

Procedure

Writes information about the user's system to a file.

Syntax

writeProfileString (const ***fileName*** String, const ***section*** String, const ***key*** String, const ***value*** String) Logical

Description

Writes a value to a specified section of a file on the user's system. Typically, you use this method to modify the user's WIN.INI file, so *fileName* is WIN.INI.

A section in WIN.INI is marked by a string bounded by square brackets on a line by itself (for example, [windows]). However, omit the brackets when you specify *section*; that is, to specify the [windows] section, use "windows."

Within a section, a string followed by = specifies *key* (for example, "Beep = "), but don't include the = when you specify *key*. Specify *value* by writing a string after the equal sign (=) in the key.

Example

See the example for readProfileString.

See also

- readProfileString, writeEnvironmentString

Table

Table

add	enumFieldNamesinIndex	protect
attach	enumFieldStruct	reIndex
cAverage	enumIndexStruct	reIndexAll
cCount	enumRefIntStruct	rename
cMax	enumSecStruct	setExclusive
cMin	familyRights	setFilter
cNpv	fieldName	setIndex
compact	fieldNo	setReadOnly
copy	fieldType	showDeleted
cSamStd	isAssigned	sort
cSamVar	isEmpty	subtract
cStd	isEncrypted	tableRights
cSum	isShared	type
cVar	isTable	unattach
delete	lock	unlock
dropIndex	nFields	unprotect
empty	nKeyFields	usesIndexes
enumFieldNames	nRecords	

A Table variable represents a description of a table. It is distinct from a TCursor, which is a pointer to the data, and from a table frame and a TableView, which are objects that display the data.

Using Table objects, you can add, copy, create, and index tables, do column calculations, get information about a table's structure, and more, but you can't edit records. Use a TCursor or a table frame (UIObject) for that.

For more information and examples, refer to Chapter 10 in the *ObjectPAL Developer's Guide.*

add

Beginner

Table

Method/Procedure

Adds the data in one table to another table.

Syntax

1. **add** (const ***destTableName*** String
   [, const ***append*** Logical [, const ***update*** Logical]]) Logical
2. **add** (const ***destTableVar*** Table
   [, const ***append*** Logical [, const ***update*** Logical]]) Logical

Description

Adds the data in a table to a destination table, which you can specify using a String (*destTableName* in syntax 1) or a Table variable

(*destTableVar* in syntax 2). If the destination table does not exist, this method creates it. The source table and the destination table can be the same type or different types; in any case, the tables must have compatible field structures.

Arguments *append* and *update* can be True or False. When True, *append* adds records at the end of a non-indexed destination table, or at the appropriate place in an indexed destination table. When True, *update* compares records in both tables, and where key values match, replaces the data in the destination table. When both are True, records with matching key values are updated, and others are appended. These arguments are optional, but if you specify *update*, you must also specify *append*. If omitted, both are True.

Key violations, if any, are listed in KEYVIOLS.DB in the user's private directory. This method overwrites an existing KEYVIOLS.DB or creates one, if necessary. Following are some example statements.

When tables are keyed, **add** uses the keyed fields to determine which records to update and which to append. When the destination table is not keyed, **add** fails if *update* is True.

```
myTable.add(yourTable, False, True) ; specifies update
myTable.add(yourTable)              ; specifies update and append by default
```

This method tries, for the duration of the retry period, to place write locks on the source table and the destination table. If either lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *updateCust* runs a query from an existing file, then adds records from the *Answer* table to the *Customer* table.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  newCust Query
  ansTbl Table
  destTbl String
endVar
destTbl = "Customer.db"

if executeQBEFile("getCust.qbe") then   ; if the query succeeds
  ansTbl.attach("Answer.db")

  ; attempt to add Answer.db records to Customer.db
  if isTable(destTbl) then
    if NOT ansTbl.add(destTbl) then
      msgStop("Error", "Can't write-lock " + destTbl + " table.")
    endif
  else
    msgStop("Error", "Can't find " + destTbl + ".")
  endif
```

```
else
  msgStop("Error", "Query failed.")
endIf

endmethod
```

See also

- copy, subtract

attach

Beginner

Table

Method

Associates a Table variable with a table on disk.

Syntax

1. **attach** (const ***tableName*** String) Logical
2. **attach** (const ***tableName*** String, const ***db*** Database) Logical
3. **attach** (const ***tableName*** String, const ***tableType*** String) Logical
4. **attach** (const ***tableName*** String, const ***tableType*** String, const ***db*** Database) Logical

Description

Associates a Table variable with the data in *tableName*. Optional arguments *tableType* and *db* specify a table type ("Paradox" or "dBASE") and a database, respectively. If you don't specify *tableType*, ObjectPAL tries to determine the table type from the table name's file extension. If you don't specify *db*, ObjectPAL works in the default database.

This method returns True if successful; otherwise, it returns False.

Note **attach** does not verify that *tableName* is a table, or even that the file exists. Use the **isTable** method to verify its existence.

To free a Table variable completely, use **unAttach**. To associate the Table variable with another table, just use **attach** again; the **unAttach** happens automatically.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

In this example, the *westTable* Table variable is attached to *Orders* so that **cSum** can be used with that Table variable. This example uses **isTable** to determine whether *Orders* exists in the default database before performing a calculation on the table.

```
; getWestTotal::pushButton
method pushButton(var eventInfo Event)
var
```

```
  westTable Table
  westTotal Number
endVar

if isTable("Orders.db") then

   ; attach to Paradox table Orders in the default database
  westTable.attach("Orders", "Paradox")
   ; get total of Total Invoice field and store result in westTotal
  westTotal = westTable.cSum("Total Invoice")
   ; display total invoices
  msgInfo("Total Invoices", westTotal)

else
  msgInfo("Status", "Can't find Orders.db table.")
endif

endmethod
```

See also

- create, unAttach

---

cAverage

Table

Beginner

Method/Procedure

Returns the average value of a field (column) in a table.

Syntax

1. **cAverage** (const ***fieldName*** String) Number
2. **cAverage** (const ***fieldNum*** SmallInt) Number

Description

Returns the average of values in the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cAverage** handles blank values as specified by the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write-lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example uses **cAverage** to calculate the average order size in the *Orders* table. This code is attached to the **pushButton** method for the *getAvgSales* button.

```
; getAvgSales::pushButton
method pushButton(var eventInfo Event)
var
  ordTbl   Table
  avgSales Number
endVar
```

```
ordTbl.attach("Orders.db")
avgSales = ordTbl.cAverage("Total Invoice") ; store average invoice total
                                            ; in avgSales
msgInfo("Average Order size", avgSales)     ; display avgSales in a dialog

endmethod
```

See also

- cCount, cMax, cMin, cStd, cSum

cCount

Beginner

Table

Method/Procedure

Returns the number of nonblank values in a field (column) of a table.

Syntax

1. **cCount** (const ***fieldName*** String) Number
2. **cCount** (const ***fieldNum*** SmallInt) Number

Description

Returns the number of values in the column (field) specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) **cCount** works for all field types. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. If the field is non-numeric, **cCount** returns the number of non-blank values in the column of fields.

This method respects the limits of restricted views displayed in a linked table frame or multi-record object.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *lineItemInfo* uses **cAverage** and **cCount** to perform calculations on the *Qty* field in LINEITEM.DB. The example attempts to place a write lock on the table so that another user on a network can not make changes to the table between the calls to **cAverage** and **cCount**. If the lock is unsuccessful, this code aborts the operation.

```
; lineItemInfo::pushButton
method pushButton(var eventInfo Event)
var
  lineTbl Table
  avgQty, numItems Number
endVar
if lineTbl.attach("Lineitem.db") then
  if lineTbl.lock("Write") then          ; if write lock succeeds
```

```
    avgQty = lineTbl.cAverage("Qty")
    numItems = lineTbl.cCount(4)          ; assumes Qty is field 4
    lineTbl.unLock("Write")               ; unlock the table
    msgInfo("Average quantity", "Average quantity: " +
            String(qvgQty, "\nbased on ", numItems, " items.")
  else
    msgStop("Stop", "Can't lock Lineitem table.")
  endif
else
  msgStop("Sorry", "Can't attach to Lineitem table.")
endif

endmethod
```

See also

- cAverage, cMax, cMin, cStd, cSum

cMax

Beginner

Table

Method/Procedure

Returns the maximum value of a field (column) in a table.

Syntax

1. **cMax** (const ***fieldName*** String) Number
2. **cMax** (const ***fieldNum*** SmallInt) Number

Description

Returns the maximum value in the column of numeric fields specified by *fieldName* or *fieldNum.* (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cMax** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following code displays the greatest amount in the *Total Invoice* field of the *Orders* table.

```
; showMaxOrder::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
endVar

if orderTbl.attach("Orders.db") then
  ; display maximum order in a dialog box
  msgInfo("Biggest Order in History", orderTbl.cMax("Total Invoice"))
else
  msgStop("Sorry", "Can't open Orders table.")
endif
```

```
endmethod
```

See also

❒ cAverage, cCount, cMin, cStd, cSum

cMin

Beginner

Table

Method/Procedure

Returns the minimum value in a field (column) of a table.

Syntax

1. cMin (const ***fieldName*** String) Number

2. cMin (const ***fieldNum*** SmallInt) Number

Description

Returns the minimum value in the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cMin** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following code displays the smallest amount in the *Total Invoice* field of the *Orders* table.

```
; showMinOrder::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
endVar

if orderTbl.attach("Orders.db") then
    ; display smallest order in a dialog box
    msgInfo("Smallest Order in History", orderTbl.cMin("Total Invoice"))

else
  msgStop("Sorry", "Can't open Orders table.")
endif

endmethod
```

See also

❒ cAverage, cCount, cMax, cStd, cSum

cNpv

Table

Beginner

Method/Procedure

Returns the net present value of a field (column), based on a specifed discount or interest rate.

Syntax

1. **cNpv** (const ***fieldName*** String, const ***discRate*** AnyType) Number
2. **cNpv** (const ***fieldNum*** SmallInt, const ***discRate*** AnyType) Number

Description

Returns the net present value of the column of numeric fields specified by *fieldName* or *fieldNum.* (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cNpv** handles blank values as specified in the **blankAsZero** setting for the session.

The calculation is based on *discRate*, expressed as a decimal (for example, 0.12 for 12 percent). This method calculates net present value using the following formula:

$$CPNV = \sum_{p=1}^{N} \frac{Vp}{(1+i)^{p}}$$

where

p = number of periods, Vp = cash flow in pth period, and i = rate per period.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following defines a Table variable for the *GoodFund* table, then calculates the net present value for the *Expected Return* field. For this example, the net present value is calculated based on a monthly interest rate.

```
; calcNPV::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  goodFundNPV, apr Number
endVar
apr = .125                      ; annual percentage rate
```

```
tbl.attach("GoodFund.db")

; calculate net present value based on monthly interest rate
goodFundNPV = tbl.cNpv("Expected Return", (apr / 12))
msgInfo("Net present value", goodFundNPV)

endmethod
```

See also

- cAverage, cMax, cMin, cStd, cSum, cVar

compact

Table

Beginner

Method

Removes deleted records from a table.

Syntax

compact ([const ***regIndex*** Logical]) Logical

Description

Deleted records are not immediately removed from a dBASE table. Instead, they are flagged as deleted and kept in the table. **compact** removes deleted records. The optional argument *regIndex* specifies whether to regenerate indexes associated with the table, or just to fix them. When *regIndex* is True, this method regenerates all maintained indexes associated with the table: indexes specified by **usesIndexes**, and the .MDX index whose name matches the table name. When *regIndex* is False, only maintained indexes are regenerated. If omitted, *regIndex* is True by default.

When records are deleted from a Paradox table, they can no longer be retrieved—they are permanently deleted. However, the table file (and associated index files) contain "dead" space where the record was originally stored. **compact** removes dead space from Paradox files.

This method fails if any locks have been placed on the table, or the table is open. This method returns True if successful; otherwise, it returns False.

Example

The following example demonstrates how **compact** affects indexes specified by **usesIndexes**. In this example, the *ordTbl* Table variable is attached to ORDERS.DBF and *salesTbl* is attached to SALES.DBF. Because *ordTbl* uses INDEX1.NDX and INDEX2.NDX (specified by **usesIndexes**), **compact** regenerates INDEX1.NDX and INDEX2.NDX if *regIndex* is True. For this example, *regIndex* is set to False so **compact** affects only ORDERS.NDX.

```
; compactTbls::pushButton
method pushButton(var eventInfo Event)
var
  ordTbl, salesTbl Table
endVar
```

```
ordTbl.usesIndexes("index1.ndx", "index2.ndx")
ordTbl.attach("Orders.dbf")
ordTbl.compact(False)
  ; removes deleted records and fixes Orders.mdx

salesTbl.usesIndexes("index3.mdx")
salesTbl.attach("Sales.dbf")
salesTbl.compact()
  ; removes deleted records and regenerates all indexes

endmethod
```

See also

- showDeleted, usesIndexes

---

copy

Beginner

Table

Method/Procedure

Copies a table.

Syntax

1. **copy** (const ***destTable*** String) Logical
2. **copy** (const ***destTable*** Table) Logical

Description

Copies the records from a source table to a destination table (*destTable*). The source table and destination table can be different types of tables, but both tables must have compatible field types. **copy** fails if the source table and destination tables have incompatible field types.

If the destination table already exists, **copy** overwrites it without asking for confirmation. If the destination table is open, the method fails.

This method tries, for the duration of the retry period, to place a write lock on the source table, and a full (exclusive) lock on the destination table. If either lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *backupCust* copies the *Customer* table to *CustBak*. If *CustBak* already exists in the current directory, this method asks the user for confirmation before overwriting it.

```
; backupCust::pushButton
method pushButton(var eventInfo Event)
var
  srcTbl  Table
  destTbl String
endVar
```

```
destTbl = "CustBak.db"
srcTbl.attach("Customer.db")

if isTable(destTbl) then      ; if "CustBak.db" exists
                              ; ask for confirmation
  if msgQuestion("Copy table", "Overwrite " + destTbl + "?") <> "Yes" then
    return
  endif
endif
srcTbl.copy(destTbl)    ; this copies Customer.db to NewCust.db

endmethod
```

See also

- add, subtract, rename

create

Table

Keyword

Creates a table.

Syntax

create "*tableName*" [**as** "*tableType*"] [**database** *db*]
 [[**like** *likeObject*]
  [**with** "*fieldName*" : "*type*" [, "*fieldName*" : "*type*"] *]
  [**where** *fieldDesc* **is** "*newName*" [, *fieldDesc* **is** "*newname*"] *]
  [**without** *fieldDesc* [, *fieldDesc*] *]
  [**struct** *fieldStructTable*]
  [**indexStruct** *indexStructTable*]
  [**refIntStruct** *refIntStructTable*]
  [**secStruct** *secStructTable*]
  ] *
 [**key** *fieldDesc* [, *fieldDesc*] *]
endCreate

Description

Creates a table, where

tableName is the file name of the table to create. For example:

```
"Orders.db"
```

If *tableName* exists, **create** tries, for the duration of the retry period, to place a full (exclusive) lock on *tableName*. If the lock cannot be placed, **create** fails.

AS *tableType* specifies the table format. Example:

```
AS "Paradox"
```

If **as** is omitted, *tableType* is Paradox by default. **as** infers *tableType* from the *fileName* extension if given. (.DB is a Paradox table and .DBF is a dBASE table.)

database *db* is a database variable (opened before creating the new table) that specifies where the table will reside. If omitted, *tableName* is created in the default database. For example:

```
DATABASE megaData
```

like *likeObject* specifies an opened TCursor, table name, or TableView from which to borrow field names and field types. The **like** clause does not borrow validity checks, primary or secondary indexes, referential integrity information, or security information. (Use **struct**, **indexStruct**, **reFintStruct**, and **secStruct** options to borrow more detailed information.)

For example:

```
like "Sales.dbf"       ; table name as a string
like ordersTC          ; a TCursor variable pointing to ORDERS.DB
like ordersTV          ; a TableView variable pointing to ORDERS.DB
```

with *"fieldName"* : *"type"* adds one or more fields to the table structure. For example:

```
with "Last name" : "A20", "First name" "A15", "Quantity" : "N"
```

Specify in *type* the field type for *fieldName*. Valid values for *type* vary depending on the type of table you are creating. The following table lists valid field specifications for Paradox and dBASE tables.

Field type	Paradox table	dBASE table
Alphanumeric	A*nn*	(none)
Character	(none)	C*nn*
Date	D	D
Integer	S	N
Floating-point value	N	You cannot create a floating-point field with create.
Graphic	G	(none)
Logical	(none)	L
Money	$	(none)
Memo	M*nn*	M
Formatted memo	F*nn*	(none)
Binary	B*nn*	(none)
OLE object	O	(none)

where *fieldDesc* **is** *"newName"* changes the name of one or more fields *fieldName* (name or number) to *"newName"* (String). Example:

```
where "Last name" is "Customer last name", 2 is "Customer first name"
```

without *fieldDesc* removes one or more fields from the structure. Example:

```
without 4, "Country code"
```

struct specifies in *fieldStructTable* an opened TCursor, table name, or TableView from which to borrow the field-level structure. Unlike the **like** clause, **struct** borrows all validity check and primary key information. Use the **enumFieldStruct** method to generate *fieldStructTable* (or create it manually) before executing **create**. For example:

```
struct "CustFlds.db"
```

indexStruct specifies in *indexStructTable* an opened TCursor, table name, or TableView from which to borrow secondary index information. Use the **enumIndexStruct** method to generate *indexStructTable* (or create it manually) before executing **create**. For example:

```
indexStruct "CustIndx.db"
```

refIntStruct specifies an opened TCursor, table name, or TableView from which to borrow referential integrity information. Use the **enumRefIntStruct** method to generate *refIntStructTable* (or create it manually) before executing **create**. For example:

```
refIntStruct "Cust_Ref.db"
```

secStruct specifies in *secStructTable* an opened TCursor, table name, or TableView from which to borrow security information. Use the **enumSecStruct** method to generate *secStructTable* (or create your own) before executing **create**. For example:

```
secStruct "Cust_Sec.db"
```

key *fieldID* specifies one or more key fields (Paradox tables only). You must specify key fields in order from left to right. For example:

```
key "Last name", "First name"
```

Fields are created in the order you specify them, whether explicitly using a **with** clause, or as implied by one or more **like** clauses. **where** and **without** clauses have no meaning unless preceded by a **like** clause.

Note **create** is not a method, so dot notation, as in the following statement, is inappropriate:

```
tableVar.create()
```

Instead, use = to assign the **create** structure to a Table variable.

Example

The following example creates the Paradox table PARTS.DB. The table has three fields: Part number, Part name, and Quantity. It has one key field: Part number.

```
; createParts::pushButton
method pushButton(var eventInfo Event)
var
  newParts Table
  partsTV TableView
endVar
if isTable("Parts.db") then
  if msgQuestion("Confirm", "Parts.db exists. Overwrite it?") <> "Yes" then
    return
  endif
endif
newParts = create "Parts.db"
           with "Part number" : "A20", "Part name" : "A20", "Quantity" : "S"
           key "Part number"
           endCreate

partsTV.open("Parts.db")      ; open the new table
endmethod
```

The following examples show two ways to create the dBASE table NEWSALES.DBF using the same structure as the dBASE table SALES.DBF.

```
; version 1
var
  newSales Table
endVar
newSales = create "Newsales.dbf"
           like "Sales.dbf"
           endCreate

; version 2
var
  newSales Table
  salesTC TCursor
endVar
salesTC.open("Sales.dbf")
newSales = create
           like salesTC
           endCreate
```

The next example uses the **struct** option to borrow field-level information, including primary keys and validity checks, for use in a new table. (See **enumFieldStruct** for more information.)

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
  custTC TCursor
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  ; include from Customer field names, ValChecks,
  ; and key fields in new table CustFlds
  if custTbl.enumFieldStruct("CustFlds.db") then

    ; open a TCursor for CustFlds table
    custTC.open("CustFlds.db")
    custTC.edit()

    ; this loop scans through the CustFlds table and
```

```
    ; changes ValCheck definitions for every field
    scan custTC :
      custTC." Required Value" = 1    ; make all fields required
    endscan

    ; now create newcust.DB and borrow field names,
    ; ValChecks and key fields from CUSTFLDS.DB
    newCustTbl = create "NewCust.db"
                   struct "CustFlds.db"
                 endCreate

    ; NEWCUST.DB requires that all fields be filled

  else
    msgStop("Error", "Can't get field structure for Customer table.")
  endif

else
  msgStop("Error", "Can't find Customer table.")
endif

endmethod
```

See also

- copy, enumFieldStruct enumIndexStruct, enumRefIntStruct, enumSecStruct

---

cSamStd

Table

Beginner

Method/Procedure

Returns the sample standard deviation of a field (column) of a table.

Syntax

1. **cSamStd** (const ***fieldName*** String) Number
2. **cSamStd** (const ***fieldNum*** SmallInt) Number

Description

Returns the sample standard deviation for the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered form left to right, beginning with 1). This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cSamStd** handles blank values as specified in the **blankAsZero** setting for the session.

The calculation is based on the sample variance. The sample (as opposed to population) standard deviation is calculated using the formula:

sqrt((*sampVar*) * (*n*/*n*-1))

where

sampVar = cVar(*tableName, fieldName*)

n = cCount(*tableName, fieldName*).

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following example uses both forms of the syntax to calculate the sample standard deviation of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamStd.*

```
; showSamStd::pushButton
method pushButton(var eventInfo Event)
var
  empTbl Table
  tblName String
  calcSalary, calcYears Number
endVar
tblName = "Answer"

empTbl.attach(tblName)
calcSalary = empTbl.cSamStd("Salary")  ; get sample std deviation for Salaries
calcYears  = empTbl.cSamStd(2)         ; assume "Years in service" is field 2
msgInfo("Sample Std Deviation",        ; display info in a dialog box
       "Salaries : " + String(calcSalary,
       "\nYears in service : ", calcYears))

endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamVar, cStd, cSum, cVar

---

cSamVar

Table

Beginner

Method/Procedure

Returns the sample variance of a field (column) in a table.

Syntax

1. cSamVar (const ***fieldName*** String) Number

2. cSamVar (const ***fieldNum*** SmallInt) Number

Description

Returns the sample variance of the column of numeric fields specified by *fieldName* or *fieldNum.* (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cSamVar** handles blank values as specified in the **blankAsZero** setting for the session.

The sample (as opposed to population) variance is calculated using the formula:

$$\sqrt{(variance) * \left(\frac{n}{n-1}\right)}$$

where...

n = cCount(*tableName, fieldName*)

This method tries, for the duration of the retry period, to place a lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following example uses both forms of the syntax to calculate the sample variance of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamVar*.

```
; showSamVar::pushButton
method pushButton(var eventInfo Event)
var
   empTbl Table
   tblName String
   calcSalary, calcYears Number
endVar
tblName = "Answer"

empTbl.attach(tblName)
calcSalary = empTbl.cSamVar("Salary")  ; get sample variance for Salaries
calcYears  = empTbl.cSamVar(2)         ; assume "Years in service" is field 2
msgInfo("Sample Variance",             ; display info in a dialog box
       "Salaries : " + String(calcSalary,
       "\nYears in service : ", calcYears))

endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cStd, cSum, cVar

cStd

Table

Beginner

Method/Procedure

Returns the standard deviation of a field (column) in a table.

Syntax

1. **cStd** (const ***fieldName*** String) Number
2. **cStd** (const ***fieldNum*** SmallInt) Number

Description

Returns the population standard deviation of the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. The calculation is based on the variance; see **cVar**. This

method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *thisButton* calculates the population standard deviation for two separate fields and displays the results in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable Table
  test1, test2 Number
endVar
myTable.attach("scores.db")
test1 = myTable.cStd("Test1")
test2 = myTable.cStd(2)        ; assumes Test2 is field 2
msgInfo("Standard Deviation",
       "Test1 results : " + String(test1) + "\n" +
       "Test2 results : " + String(test2))
endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cSum, cVar

---

cSum

Table

Beginner

Method/Procedure

Returns the sum of the values in a field (column) of a table.

Syntax

1. **cSum** (const ***fieldName*** String) Number
2. **cSum** (const ***fieldNum*** SmallInt) Number

Description

Returns the sum of the values in the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cSum** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write-lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *sumOrders* uses both forms of **cSum** syntax to calculate totals for two fields in ORDERS.DB.

```
; sumOrders::pushButton
method pushButton(var eventInfo Event)
var
  orderTbl Table
  orderTotal, amtPaid Number
  tblName String
endVar
tblName = "Orders"

orderTbl.attach(tblName)
orderTotal = orderTbl.cSum("Total Invoice")
amtPaid    = orderTbl.cSum(7)    ; assumes Amount Paid is field 7
msgInfo("Order Totals",
        "Total Orders : " + String(orderTotal) + "\n" +
        "Total Receipts : " + String(amtPaid))

endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cStd, cVar

cVar

Table

Beginner

Method/Procedure

Returns the variance of a field in a table.

Syntax

1. cVar (const ***fieldName*** String) Number

2. cVar (const ***fieldNum*** SmallInt) Number

Description

Returns the population variance of the column of numeric fields specified by *fieldName* or *fieldNum*. (Fields are numbered from left to right, beginning with 1.) This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write-lock on the table. If a lock cannot be placed, the method fails.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *thisButton* calculates the population variance deviation for two separate fields and displays the results in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
```

```
var
  myTable Table
  test1, test2 Number
endVar
myTable.attach("scores.db")
test1 = myTable.cVar("Test1")
test2 = myTable.cVar(2)       ; assumes Test2 is field 2
msgInfo("Population Variance",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))

endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cStd, cSum

delete

Table

Beginner

Method/Procedure

Deletes a table.

Syntax

delete ( ) Logical

Description

Deletes a table without asking for confirmation. Compare this method to **empty**, which removes data from a table but does not delete it.

This method fails if the table is open or is locked.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following code deletes ANSWER.DB from the private directory if it exists.

```
; delAnswer::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  tblName String
endVar

tblName = privDir() + "\\Answer.db"

tbl.attach(tblName)
if tbl.isTable() then
  tbl.delete()
  message(tblName, " deleted.")
else
  message("Can't find ", tblName, ".")
endif

endmethod
```

See also

- empty

dropIndex

Table

Method

Deletes an index file associated with a table.

Syntax

1. (Paradox tables) **dropIndex** ([const ***indexName*** String]) Logical
2. (dBASE tables) **dropIndex** ([const ***indexName*** String [, const ***tagName*** String]]) Logical

Description

Deletes a specified index file or index tag.

When working with a Paradox table, *indexName* is optional. If *indexName* is omitted, **dropIndex** deletes the table's primary index (unless the table has a secondary index, in which case the method fails).

When working with a dBASE table, you can use *indexName* to specify a .NDX file, or use *indexName* and *tagName* to specify a .MDX file and an index tag.

This method requires exclusive rights to the table if you're dropping a maintained index; otherwise, it requires a write lock.

dropIndex fails if the index you're trying to delete is currently being used, or if the table is open.

Example

For this example, the **pushButton** method for *thisButton* deletes the *CustName* tag from a .MDX file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  salesTbl Table
endVar

salesTbl.attach("Sales.dbf")          ; Sales.dbf is a dBASE table
if isTable(salesTbl) then             ; if salesTbl is a table

  ; delete CustName tag from index2.mdx file
  if salesTbl.dropIndex("index2.mdx", "CustName") then
    msgInfo("Status", "CustName index deleted.")
  else
    msgInfo("Error", "Can't drop CustName from Index2.")
  endif

else
  msgStop("Stop!", "Could not find Sales.dbf table.")
endif

endmethod
```

See also

- setIndex, usesIndexes

---

empty

Table

Beginner

Method/Procedure

Removes all records from a table in a database.

Syntax

empty **( )** Logical

Description

Removes all records from a table without asking for confirmation. This operation cannot be undone. **empty** fails if the table is open.

empty removes information from the table, but does not delete the table itself. Compare this method to **delete**, which does delete the table.

This method first tries to gain exclusive rights to the table. If exclusive rights are not possible, **empty** tries to place a write lock on the table.

If exclusive rights are possible, this method deletes all records in the table at once. If only a write lock is possible, **empty** must delete each record one at a time. (This can be expensive for large tables.)

If **empty** is able to gain exclusive rights to a dBASE table, all records are deleted and the table is compacted (records are permanently removed). If only a write lock is possible, this method flags all records as deleted, but does not remove them from the table. (Records can be undeleted from a dBASE table if they have not been removed with the **compact** method.)

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table. If the user does not confirm the action, this code uses **nRecords** to indicate how many records exist in SCRATCH.DB.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
  tblName String
  tblVar Table
endVar
tblName = "Scratch.db"

tblVar.attach(tblName)
if isTable(tblName) then
  if msgYesNoCancel("Confirm", "Empty " + tblName + " table?") = "Yes" then
    if tblVar.empty() then
```

```
    message("All " + tblName + " records have been deleted.")
  else
    message("Empty failed." +
      tblname + " has " + String(tblVar.nRecords()) + " records.")
  endif
endif
else
  msgInfo("Error", "Can't find " + tblName + " table.")
endif
endmethod
```

See also

- delete

enumFieldNames

Table

Method

Fills an array with the names of fields in a table.

Syntax

enumFieldNames (var ***fieldArray*** Array[] String) Logical

Description

Fills *fieldArray* with the names of the fields in a table. You must declare *fieldArray* as a resizeable array before calling this method. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

Example

For this example, the **pushButton** method for the *enumFields* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; showIndexFlds::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  fieldNames Array[] AnyType
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
  tbl.enumFieldNamesInIndex("DateIndx", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't find Sales.dbf.")
endIf

endmethod
```

See also

- enumFieldNamesInIndex, enumFieldStruct, enumIndexStruct, enumRefIntStruct, enumSecStruct

enumFieldNamesInIndex

Table

Method

Fills an array with the names of fields in a table's index.

Syntax

1. (Paradox tables) **enumFieldNamesInIndex**
   ([const ***indexName*** String,]
   var ***fieldArray*** Array[] String) Logical
2. (dBASE tables) **enumFieldNamesInIndex**
   ([const ***indexName*** String, [const ***tagName*** String,]]
   var ***fieldArray*** Array[] String) Logical

Description

Fills *fieldArray* with the names of the fields in a table's index, as specified in *indexName*. You must declare *fieldArray* as a resizeable array before calling this method. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

When working with a dBASE table, the argument *tagName* is required to specify an index tag within a .MDX file.

If omitted, *indexName* corresponds to the index currently being used.

Example

For this example, the **pushButton** method for the *showIndexFlds* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; showIndexFlds::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  fieldNames Array[] String
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
  tbl.enumFieldNamesInIndex("DateIndx", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't find Sales.dbf.")
endIf

endmethod
```

See also

- enumFieldNames, enumFieldStruct, enumIndexStruct, enumRefIntStruct, enumSecStruct

enumFieldStruct

Table

Method

Creates a Paradox table listing the field structure of a table.

Syntax

enumFieldStruct (const ***tableName*** String) Logical

Description

Creates the Paradox table specified in *tableName* listing the structure of a Table. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **struct** option in a **create** statement to borrow a table's field structure (including primary keys and validity checks) for use in the new table. The structure for *tableName* is listed in the following table.

Field Name	Type	Size
Field Name	A	31
Type	A	31
Size	S	
Dec	S	
Key	A	1
_Required Value	A	1
_Min Value	A	255
_Max Value	A	255
_Default Value	A	255
_Picture Value	A	175
_Table Lookup	A	81
_Table Lookup Type	A	1
_Invariant Field ID	S	

Example

For this example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you want all of the fields in *NewCust* to be required fields. To accomplish this, the following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans through *CustFlds* and modifies the field definitions so that each record describes a field that will be required. *CustFlds* is then supplied in the **struct** clause of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
  custTC TCursor
```

Table

```
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  ; include from Customer field names, ValChecks,
  ; and key fields in new table CustFlds
  if custTbl.enumFieldStruct("CustFlds.db") then

    ; open a TCursor for CustFlds table
    custTC.open("CustFlds.db")
    custTC.edit()

    ; this loop scans through the CustFlds table and
    ; changes ValCheck definitions for every field
    scan custTC :
      custTC."_Required Value" = 1    ; make all fields required
    endscan

    ; now create NEWCUST.DB and borrow field names,
    ; ValChecks and key fields from CUSTFLDS.DB
    newCustTbl = create "NewCust.db"
                   struct "CustFlds.db"
                 endCreate

    ; NEWCUST.DB requires that all fields be filled

  else
    msgStop("Error", "Can't get field structure for Customer table.")
  endif

else
  msgStop("Error", "Can't find Customer table.")
endif

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumIndexStruct, enumRefIntStruct, enumSecStruct

enumIndexStruct

Table

Method

Creates a Paradox table listing the structure of a table's secondary indexes.

Syntax

enumIndexStruct (const ***tableName*** String) Logical

Description

Creates the Paradox table specified in *tableName* listing the structure of a table's secondary indexes. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow secondary indexes for use in the new table.

The structure of *tableName* is listed in the following table.

Field Name	Type	Size
infoHeader	A	1
szName	A	127
szTagName	A	31
szFormat	A	31
bPrimary	A	1
bUnique	A	1
bDescending	A	1
bMaintained	A	1
bCaseInsensitive	A	1
bSubset	A	1
bExpIdx	A	1
bKeyExpType	N	
szKeyExp	A	220
szKeyCond	A	220
FieldNo	N	
FieldName	A	31

For dBASE tables, *tableName* includes information for indexes which would be used if the table were open. To specify which indexes to associate to a Table variable, use the **usesIndexes** method, then call **enumIndexStruct** to create a table that list those indexes.

Example

For this example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you don't want to borrow referential integrity or security information. To accomplish this, the following code uses **enumFieldStruct** and **enumIndexStruct** to generate two tables: CUSTFLDS.DB and CUSTINDX.DB. *CustFlds* and *CustIndx* are then supplied to the **struct** and **indexStruct** clauses of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  custTbl, newCustTbl Table
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

  custTbl.enumFieldStruct("CustFlds.db")
  custTbl.enumIndexStruct("CustIndx.db")

  ; now create NEWCUST.DB--
  ; borrow field names, ValChecks, and key fields from CUSTFLDS.DB
```


Table

```
  ; borrow secondary indexes from CUSTINDX.DB
  newCustTbl = create "NewCust.db"
                 struct "CustFlds.db"
                 indexStruct "CustIndx.db"
               endCreate

else
  msgStop("Error", "Can't find Customer table.")
endif

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct enumRefIntStruct, enumSecStruct

enumRefIntStruct

Table

Method

Creates a Paradox table listing a table's referential integrity information.

Syntax

enumRefIntStruct (const ***tableName*** String) Logical

Description

Writes the referential integrity information for the current table to *tableName*. If *tableName* exists, this method overwrites it without confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow referential integrity information for use in the new table. The structure for *tableName* is listed in the following table.

Field Name	Type	Size
infoHeader	A	1
Ref Name	A	31
Other Table	A	81
Slave	A	1
Modify	A	1
Delete	A	1
FieldNo	N	
aiThisTabField	A	31
Other FieldNo	N	
aiOthTabField	A	31

Example

This example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. Then, the code supplies *CustRef* to the **refIntStruct** clause in a **create** statement.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tbl, tb2 Table
endVar

tbl.attach("Customer.db")
tbl.enumRefIntStruct("CustRef.db")    ; write ref. integ. info to CustRef
tbl.enumFieldStruct("CustFlds.db")    ; write field structure to CustFlds

tb2 = create "NewCust.db"
       struct "CustFlds.db"           ; use field-level info. from CustFlds
       refIntStruct "CustRef.db"      ; use ref. integ. infor from CustRef
     EndCreate

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct, enumIndexStruct, enumSecStruct

enumSecStruct

Table

Method

Creates a Paradox table listing a table's security information.

Syntax

enumSecStruct (const ***tableName*** String) Logical

Description

Creates the Paradox table specified in *tableName* listing security information (access rights) of a table. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no

alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **secStruct** option in a **create** statement to borrow security information for use in the new table. The structure of *tableName* is listed in the following table.

Field Name	Type	Size
infoHeader	A	1
iSecNum	N	
eprvTable	N	
eprvTableSym	N	
iFamRights	N	
iFamRightsSym	A	10
szPassword	A	31
fldNum	N	
aprvFld	N	
aprvFldSym	A	10

Example

This example creates a new table based on the security information associated with the *Secrets* table. The code uses **enumSecStruct** to write security information to the *SecInfo* table, then uses the table to create the *MySecrts* table.

```
; getSecrets::pushButton
method pushButton(var eventInfo Event)
var
  tbl, tb2 Table
endVar

tbl.attach("Secrets.db")
tbl.enumSecStruct("SecInfo.db")

tb2 = create "MySecrts.db"
        like "Secrets.db"
        secStruct "Secrets.db"
      endCreate

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct, enumIndexStruct, enumRefIntStruct

familyRights

Table

Method

Tests for a user's ability to create or modify objects in a table's family.

Syntax

familyRights (const ***rights*** String) Logical

Description

Returns True if you have rights to the type of object specified in *rights*; otherwise, it returns False. *rights* is a single-letter string—either "F" (form), "R" (report), "S" (image settings), or "V" (validity checks)— that indicates the type of object you are interested in.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example indicates in a dialog box whether you have "F" rights to CUSTOMER.DB.

```
; showFRights::pushButton
method pushButton(var eventInfo Event)
var
  custTB Table
endVar

custTB.attach("Orders.db")
if custTB.isTable() then
  msgInfo("Rights", "Form Rights: " +
          String(custTB.familyRights("F")))
  ;displays True if you have Form rights to Orders.db
else
  msgStop("Error", "Can't find Orders.db.")
endif

endmethod
```

See also

- tableRights

Table

fieldName

Table

Method/Procedure

Returns the name of field in a table, given a field number.

Syntax

fieldName (const ***fieldNum*** SmallInt) String

Description

Returns the name of the field specified in *fieldNum*. If *fieldNum* is greater than the number of fields in the table, **fieldName** returns an empty string.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following example uses **fieldName** to display the name of field number two in the *Answer* table. This code is attached to the built-in **pushButton** method of a button.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
```

```
  fldName, tblName String
  fldNum SmallInt
endVar
tblName = "Answer.db"
fldNum = 2

tbl.attach(tblName)
if isTable(tbl) then
  fldName = tbl.fieldName(fldNum)   ; store name of field 2 in fldName
  msgInfo("The name of field " + String(fldNum) + " is:", fldName)
else
  msgStop("Sorry", "Can't find " + tblName + " table.")
endIf

endmethod
```

See also

- fieldNo

fieldNo

Table

Method/Procedure

Returns the position of a field in a table.

Syntax

fieldNo (const ***fieldName*** String) SmallInt

Description

Returns the position of *fieldName* in a table, or 0 if *fieldName* is not found. Fields are numbered from left to right, beginning with 1.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This code displays the field number of the Date field if it exists in the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ord Table
  fldNo SmallInt
endVar

ord.attach("Orders.db")
fldNo = ord.fieldNo("Date")

if fldNo = 0 then
  msgInfo("Orders table", "Date is not a field in this table.")
else
  msgInfo("Orders table", "Date is field number " + String(fldNo))
endif

endmethod
```

See also

- fieldName

fieldType

Table

Method/Procedure

Returns the type of a field in a table.

Syntax

1. **fieldType** (const ***fieldName*** String) String
2. **fieldType** (const ***fieldNum*** SmallInt) String

Description

Returns the data type of a field. If the field is not found, this method returns "Unknown". The following table list the possible return values:

Field type	Paradox table	dBASE table
Alphanumeric	ALPHA	(none)
Character	(none)	CHARACTER
Date	DATE	DATE
Integer	SHORT	(none)
Floating-point value	NUMERIC	FLOAT (IV)
		NUMERIC (III+ or IV)
Graphic	GRAPHIC	(none)
Logical	(none)	BOOLEAN
Money	MONEY	(none)
Memo	MEMO	MEMO
Formatted memo	FMTMEMO	(none)
Binary	BINARYBLOB	(none)
OLE object	OLEOBJ	(none)

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example uses a dynamic array to store the type of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldTypes::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  i SmallInt
  fldTypes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if isTable(tblName) then
  tblVar.attach(tblName)
    ; this FOR loop loads the DynArray with BioLife.db field types
  for i from 1 to tblVar.nFields()
    fldTypes[tblVar.fieldName(i)] = tblVar.fieldtype(i)
```

Table

```
  endFor
    ; now show the contents of the DynArray
  fldTypes.view(tblName + " field types")
else
  msgStop("Sorry", "Can't find " + tblName + " table.")
endif
endmethod
```

See also

- fieldNo

index

Table

Keyword

Creates a primary or secondary index on the specified fields of a table.

Syntax

```
1. index
     [ maintained ] tableDesc on fieldID
   endIndex
2. index tableDesc
     [ maintained ]              (Paradox)
     [ descending ]              (dBASE)
     [ unique ]                  (dBASE)
     [ primary ]                 (Paradox)
     [ caseInsensitive ]         (Paradox)
     on
       { fieldDesc [, fieldDesc ] [to indexName]
         |
       { keyExp
         to ndxFileName | tag tagName [ of mdxFileName ]
           |
         for condition } }
   endIndex
```

Description

Generates a secondary index for a field of a table. Paradox uses the index to speed up queries and searches that access the field.

If you are building an index on a keyed table, the **maintained** keyword specifies to Paradox that you want the index you are creating to be *incrementally maintained*. Incremental maintenance means that after changes made to a table are saved, only that portion of the index affected by the change will be updated. Tables that are keyed and have incrementally maintained secondary indexes

typically result in far better performance than those that are not set up this way, although there is a hidden performance cost because maintaining the index takes resources.

If you use the **maintained** keyword for Paradox tables and specify a non-keyed table to index, **index** fails. Without the **maintained** keyword, a secondary index is not maintained automatically. Instead, you must use **reIndex** or **reIndexAll**. For dBASE tables, all opened index files are automatically maintained.

The **on** clause, which specifies which fields to use, has two forms: one for Paradox tables, and one for dBASE tables.

Paradox tables

For Paradox tables, use

on *fieldDesc* [, *fieldDesc*] to *indexName*

where *fieldDesc* specifies one or more field names or field numbers, and *indexName* specifies the index file to write to. For Paradox tables, secondary index files have extensions .X*nn* and .Y*nn*, where *nn* refers to the structural position of the field being indexed, expressed in hexadecimal notation.

dBASE tables

For dBASE tables, use

keyExp to *ndxFileName* | tag *tagName* [of *mdxFileName*]

which lets you choose between a .NDX file or a tag in a .MDX file. If *mdxFileName* is omitted, the default .MDX file name is the same as the table

In multiuser applications, **index** automatically places a full lock on the table while it is being indexed. If the table has been locked by another user or application, the command is continuously retried for the duration of the currently set retry period. If the lock cannot be obtained by the end of the period, a **index** fails. You can use the **lock** method to make certain that you can lock the table *before* you use the **index** command.

Because keyed tables have primary indexes on their key fields, you don't have to create a secondary index for these fields; doing so would only duplicate the primary index. However, you can use **index** to create a primary index. Memo fields, OLE, and Graphic fields cannot be indexed.

It's convenient to develop your applications without worrying about indexes, then introduce them where appropriate to speed up queries and searches.

In the following situations, the index command will not successfully complete:

- ❐ Too many indexes already exist (maximum of 255 for a single table).
- ❐ An index being defined is already in use.

index is not a method, so dot notation, as in

```
tableVar.index()
```

is inappropriate. Instead, you create an index structure to specify how to index the table.

Example

The following example builds a primary index for a Paradox table named CUSTOMER.DB. If the *Customer* table can not be found, or can not be locked, this method aborts the **index** operation. If this code successfully indexes the table, the code enumerates indexed fields to an array and displays the contents of the array in a dialog box.

```
; newCustKeys::pushButton
method pushButton(var eventInfo Event)
var
  tblToIndex String
  tblVar Table
  indexedFlds Array[] String
endVar
tblToIndex = "Customer.db"

if isTable(tblToIndex) then
  tblVar.attach(tblToIndex)
  if not tblVar.lock("Full") then
    msgStop("Stop!", "Can't lock " + tblToIndex + " table.")
    return
  endif
  index tblVar              ; create new primary index for Customer.db
    primary
    on "Customer No", "Name", "Street"
  endIndex

    ; now display Customer's keyed fields in a dialog box
  tblVar.enumFieldNamesInIndex(indexedFlds)
  indexedFlds.view("Primary key fields for " + tblToIndex)

else
  msgStop("Stop!", "Can't find " + tblToIndex + " table.")
endif

endmethod
```

The following example builds a maintained secondary index named *CityState* for the Paradox table, CUSTOMER.DB. If successful, this code enumerates the indexed field names to an array and displays them in a dialog box.

```
; cityStateIndex::pushButton
method pushButton(var eventInfo Event)
var
  tblToIndex String
  tblVar Table
  indexedFlds Array[] String
```

```
  tv TableView
endVar
tblToIndex = "Customer.db"

if isTable(tblToIndex) then
  tblVar.attach(tblToIndex)
  if not tblVar.lock("Full") then
    msgStop("Stop!", "Can't lock " + tblToIndex + " table.")
    return
  endif

  index tblVar              ; create secondary index for Customer.db
    maintained              ; maintain index incrementally
    on "City", "State/Prov" ; index on these two fields
    to "CityState"          ; name the index "CityState"
  endIndex

    ; now display Customer's keyed fields in a dialog box
  tblVar.enumFieldNamesInIndex("CityState", indexedFlds)
  indexedFlds.view("Fields in the CityState index")

else
  msgStop("Stop!", "Can't find " + tblToIndex + " table.")
endif

endmethod
```

See also

- create, enumIndexStruct, setIndex, usesIndexes

isAssigned

Table

Method

Reports whether a Table variable has been assigned a value.

Syntax

isAssigned ( ) Logical

Description

Returns True if a Table variable has an assigned value; otherwise, it returns False. You can assign a value to a Table variable using **create** or **attach**.

Note A return value of True does not guarantee that the variable is attached to a table, or, if attached, that the table exists. For example, the following code displays True in a dialog box:

```
var tb Table endVar
tb.attach("zxcv.qw")                 ; attach to some nonsense file name
msgInfo("Assigned?", tb.isAssigned()) ; displays True
```

Example

This example tests whether the *tblVar* Table variable is assigned before attaching to a table. The following code goes in the Var window for the *thisForm* form:

```
; thisForm::var
var
```

```
  tblVar
endVar
```

The following code is attached to the **pushButton** method for the *thisButton* button. In this code, if *tblVar* is not already assigned, it is attached to the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)

if NOT tblVar.isAssigned() then
  tblVar.attach("Orders.db")
else
  msgStop("Error", "Can't attach tblVar to Orders.db")
endif

endmethod
```

See also

- attach, close isTable

isEmpty

Table

Method/Procedure

Reports whether a table contains any records. This can be an expensive operation for dBASE tables.

Syntax

isEmpty ( ) Logical

Description

Returns True if there are no records in a table; otherwise, it returns False.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for the *rptRecNo* button displays the number of records in the *Orders* table. If *Orders* is empty, this code alerts the user that the table is empty.

```
; rptRecNo::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "Orders.db"

if isTable(tblName) then
  tblVar.attach(tblName)
  if tblVar.isEmpty() then     ; if Orders.db table is empty
    msgStop("Hey!", tblName + " table is empty!")
  else
    msgInfo(tblName + " table has", String(tblVar.nRecords()) + " records")
  endIf
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
```

```
endif
endmethod
```

See also

- empty, isTable

isEncrypted

Table

Method

Reports whether a table is encrypted.

Syntax

isEncrypted **( )** Logical

Description

Returns True if a table is password-protected; otherwise, it returns False. A TCursor can't be opened on an encrypted table until you use the Session type method **addPassword** to present the required password. This method does not report whether a user has access rights to the table—use **tableRights** for that.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example uses **isEncrypted** to determine whether the *Secrets* table is protected (encrypted); if it is, the code calls **unProtect** to permanently remove password-protection from the table.

```
; decrypt::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar

tblName = "Secrets.db"
tblVar.attach(tblName)
if tblVar.isEncrypted() then
  tblVar.unprotect("Get007")    ; permanently remove password
                                ; this assumes Get007 is the master password
endif

endmethod
```

See also

- protect, tableRights
- addPassword, removePassword in the Session type

isShared

Table

Method/Procedure

Reports whether a table is currently shared.

Syntax **isShared ( )** Logical

Description

Returns True if a table is being shared by another user on a network; otherwise, it returns False. **isShared** does not report whether a table is being shared by another session.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

In this example, a Table variable is attached to the *Customer* table. This code uses **setExclusive** to give the user exclusive rights to *Customer* then uses **isShared** to demonstrate the effect **setExclusive** has on tables in a multiuser environment.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "Customer.db"

tblVar.attach(tblName)

tblVar.setExclusive(True)   ; give user exclusive rights to Customer.db
if tblVar.isShared() then   ; this is never True!
                            ; exclusive tables can't be shared
  msgStop("", "This message will never appear!")
else
  msgInfo("Multiuser Status", tblName + " is not shared.")
endif

endmethod
```

See also

- tableRights

isTable

Table

Method/Procedure

Reports whether a table exists in a database.

Syntax **isTable ( )** Logical

Description

Returns True if the Table variable represents a table that can be opened; otherwise, it returns False.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example uses **isTable** to verify that the *Customer* table exists before doing anything with the table. If *Customer* exists in the default

database, this code stores *Customer* field names in an array, then displays the contents of the array in a dialog box.

```
; showCustFlds::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
  fldNames Array[] AnyType
endVar
tblName = "Customer.db"

tblVar.attach(tblName)
if isTable(tblVar) then
  tblVar.enumFieldNames(fldNames)
  fldNames.view(tblName + " fields")
else
  msgStop("Stop!", "Can't find " + tblName + " table.")
endif

endmethod
```

See also

- isAssigned

lock

Table

Beginner

Method

Locks a specified table.

Syntax

lock (const ***lockType***) String Logical

Description

Attempts to place a lock on the table, where *lockType* is one of the following String values: Write, Read, Full, or Any. If successful, this method returns True; otherwise, it returns False.

Example

The following example attaches a Table variable to *Customer*, places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

tblVar.attach(pdoxTbl)
if tblVar.isTable() then
  if tblVar.lock("Full") then    ; attempt to gain exclusive access
    tblVar.reIndex("Phone_Zip")  ; rebuild Phone_Zip index
    tblVar.unLock("Full")        ; unlock the table
  else
```

```
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endif
else
  msgStop("Sorry", "Can't find " + pdoxTbl + " table.")
endif
endmethod
```

See also

- lockStatus, unlock, unlockRecord
- lock and unlock in the Session type

nFields

Table

Method/Procedure

Returns the number of fields in a table.

Syntax

nFields ( ) LongInt

Description

Returns the number of fields in a table.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *thisButton* displays the number of fields in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar

tblVar.attach("BioLife.db")
msgInfo("BioLife", "BioLife has " +
        String(tblVar.nFields(), " fields.")

endmethod
```

See also

- nKeyFields, nRecords

nKeyFields

Table

Method/Procedure

Returns the number of fields in the primary or current index for a table.

Syntax

nKeyFields ( ) LongInt

Description

Returns the number of fields in the current index for a table. When used on a Paradox table, this method works with the primary index; when used on a dBASE table, it works with the index specified by the **setIndex** method.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example reports the number of primary key fields in a Paradox table (ORDERS.DB) and the number of primary key fields in the LASTNAME.MDX index for a dBase table (SCORES.DBF).

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  pdoxTbl, dBaseTbl Table
  nkf LongInt
endVar

pdoxTbl.attach("Orders.db")
nkf = pdoxTbl.nKeyFields()       ; number of key fields in the primary index
msgInfo("Orders", "Orders.db has " + String(nkf) + " key fields.")


dBaseTbl.attach("Scores.dbf")
dBaseTbl.setIndex("LastName.MDX") ; number of key fields in LastName.MDX index
nkf = dBaseTbl.nKeyFields()
msgInfo("Scores.dbf", "Scores.dbf has " + String(nkf) + " key fields.")

endmethod
```

❐ nFields

nRecords

Table

Method/Procedure

Returns the number of records in a table.

Syntax

nRecords ( ) LongInt

Description

Returns the number of records in a table. For dBASE tables, this may be an expensive operation.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table. If the user does not confirm the action, this code uses **nRecords** to indicate how many records exist in SCRATCH.DB.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
```

```
var
  tblName String
  tblVar Table
endVar
tblName = "Scratch.db"

tblVar.attach(tblName)
if isTable(tblName) then
  if msgYesNoCancel("Confirm", "Empty " + tblName + " table?") = "Yes" then
    tblVar.empty()
    message("All " + tblName + " records have been deleted.")
  else
    message(tblname + " has " + String(tblVar.nRecords()) + " records.")
  endif
else
  msgInfo("Error", "Can't find " + tblName + " table.")
endif
endmethod
```

See also

- nFields, nKeyFields

protect — Table

Method/Procedure

Encrypts and assigns an owner password to a table.

Syntax

protect ([const ***password*** String]) Logical

Description

Assigns an owner password to a table. A protected table is encrypted and cannot be accessed without presenting the password specified in *password*. If optional argument password is omitted, this method permanently removes the owner password. If the table already has a password, **protect** fails.

Once a table is protected, you can use the **addPassword** method to present the password of a protected table, and the **removePassword** method to withdraw the password and reprotect the table. *password* is case-sensitive; a table protected with "Sesame" won't open for "SESAME".

Do not confuse **protect** with **lock**: **protect** encrypts tables, while **lock** controls simultaneous access to tables.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

For this example, the **pushButton** method for *protectSecrets* password-protects the *Secrets* table in the default database.

```
; protectSecrets::pushButton
method pushButton(var eventInfo Event)
var
  secretData Table
```

```
endVar

secretData.attach("Secrets.db")
if not secretData.isEncrypted() then
  secretData.protect("Get007")      ; password-protect table with "Get007"
endIf

endmethod
```

See also

- isEncrypted
- addPassword, removePassword in the Session type

reIndex

Table

Method

Rebuilds specified index files.

Syntax

1. (Paradox tables) **reIndex** (const ***indexName*** String) Logical
2. (dBASE tables) **reIndex** (const ***indexName*** String
   [, const ***tagName*** String]) Logical

Description

Rebuilds an index (or index tag) that is not automatically maintained. When working with a Paradox table, use *indexName* to specify an index. When working with a dBASE table, use *indexName* to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file. This method requires exclusive access to the table.

Example

The following example attaches a Table variable to *Customer* (a Paradox table), places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

tblVar.attach(pdoxTbl)
if tblVar.lock("Full") then    ; attempt to gain exclusive access
  tblVar.reIndex("Phone_Zip")  ; rebuild Phone_Zip index
  tblVar.unLock("Full")        ; unlock the table
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endif

endmethod
```

See also

- reIndexAll

reIndexAll

Table

Method

Rebuilds all index files associated with a table.

Syntax

reIndexAll ( ) Logical

Description

Rebuilds all index files associated with a table. This method requires exclusive rights to the table to rebuild a maintained index, and it requires a write lock to rebuild a non-maintained index.

Example

For this example, the **pushButton** method for a button attempts to place a full lock on the *Customer* table. If **lock** is successful, this code rebuilds all indexes for the *Customer* table then unlocks the table.

```
; reindexAllCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

tblVar.attach(pdoxTbl)
if tblVar.lock("Full") then      ; attempt to lock Customer.db
  tblVar.reIndexAll()            ; rebuild all Customer.db indexes
  tblVar.unLock("Full")          ; unlock the table
else
  msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endif

endmethod
```

See also

- reIndex

rename

Beginner

Table

Method/Procedure

Renames a table.

Syntax

rename (const ***destTableName*** String) Logical

Description

Changes the name of a table to *destTableName*. If the table named by *destTableName* exists, an error results.

This method tries, for the duration of the retry period, to place a full lock on the table. If the lock cannot be placed, an error results.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following code renames CUSTOMER.DB to OLDCUST. If *OldCust* exists, this example offers the user an opportunity to abort the operation.

```
; renameCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  oldName, newName String
endVar

oldName = "Customer.db"
newName = "OldCust.db"

tblVar.attach(oldName)
if tblVar.isTable() then
  if isTable(newName) then
    if msgQuestion("Confirm", newName + " exists. Overwrite it?") <> "Yes" then
      message("Operation canceled.")
      return
    endif
  endif
  tblVar.rename(newName)
  message(oldName + " renamed to " + newName)
else
  msgStop("Stop!", "Can't find " + oldName + " table.")
endif

endmethod
```

See also

- copy

---

setExclusive

Table

Method

Specifies whether to give the user exclusive rights to a table when it is opened.

Syntax

setExclusive ([const ***yesNo*** Logical])

Description

Specifies in *yesNo* whether to open a table with shared or exclusive rights. This method does not place any locks on the table—an exclusive lock is placed on the table only when it is opened.

By default, tables are opened in shared mode. Optional argument *yesNo* specifies whether to set exclusive rights: a value of Yes requests exclusive rights, a value of No allows the table to be opened in shared mode. If omitted, *yesNo* defaults to Yes.

Example

This example demonstrates how **setExclusive** affects access rights to a table. The code defines a Table variable for the *Customer* table, then calls **setExclusive** so *Customer* is opened exclusively. Then, a TCursor is opened for *Customer*. If the TCursor is sucessfully opened, it has exclusive rights to the table and the code calls the TCursor method **lockStatus** to indicate that an exclusive lock has been placed on *Customer*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc     TCursor
endvar

tblVar.attach("Customer.db")
if tblVar.isTable() then
  ; set exclusive rights for the Table variable
  tblVar.setExclusive()

  ; attempt to open a TCursor on Customer.db--
  ; if successful, tc has exclusive rights to Customer.db
  if tc.open(tblVar) then

    ; if tc.open was successful, this message indicates
    ; that tc has 1 exclusive lock on Customer.db
    msgInfo("Lock Status", tc.lockStatus("Exclusive"))

  else
    ; else open failed
    msgInfo("Status", "Can't open Customer.db")
  endif

else
  msgInfo("Status", "Can't find Customer.db table.")
endif

if tc.isAssigned() then     ; if the TCursor was opened
  tc.close()                ; close tc--now Customer.db is not
                            ; locked and can be opened by another user
endif

endmethod
```

See also

- attach, setIndex, setReadOnly

setFilter

Table

Method

Specifies a range of records to include.

Syntax

setFilter ([const ***exactMatchVal*** AnyType,] *
const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description

Specifies conditions for including a range of records. Records that meet the conditions are included when the table is opened, records that don't are filtered out.

This method compares the criteria you specify with values in the corresponding fields of a table's index; **setFilter** fails if the table is not indexed. To filter records based on the value of a single field, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record.

```
tableVar.setFilter(14, 88)
```

If a value is less than 14 or greater than 88, that record is filtered out. To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55.

```
tableVar.setFilter(55, 55)
```

You can filter records based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field.

```
tcVar.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table.

Example

In this example, assume that *Lineitem*'s key field is *Order No.* and you want to know the total for order number 1005. The following code attaches a Table variable to the *Lineitem* table, limits the range of records to those with 1005 in the first field of the primary index, then uses **cSum** to calculate the total for order 1005.

```
; getDetailSum::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar
tblName = "LineItem.db"
tblVar.attach(tblName)

  ; this limits TCursor's view to records that have
  ; 1005 in the first field of the primary index
tblVar.setFilter(1005, 1005)

  ; now display the total for Order No. 1005
msgInfo("Total for Order 1005", tblVar.cSum("Total"))

endmethod
```

See also

- setIndex

setIndex

Table

Method

Specifies an index for a table.

Syntax

1. (Paradox tables) **setIndex** (const ***indexName*** String) Logical
2. (dBASE tables) **setIndex** (const ***indexName*** String
   [, const ***tagName*** String]) Logical

Description

Specifies an index to use when a table is opened.

When working with a Paradox table, use *indexName* to specify an index. When working with a dBASE table, you can use *indexName* to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file.

Example

In this example, assume the Paradox *Customer* table has a secondary index named CityState. The following code specifies CityState with **setIndex** to set up for a call to **setFilter**. When the designated filter is set for *Customer*, this example loads a DynArray with information from the filtered table then displays the contents of the DynArray in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custTbl Table
  tc TCursor
  dy DynArray[] Anytype
endVar

custTbl.attach("Customer.db")
if isTable(custTbl) then

  ; now use the secondary index named CityState
  custTbl.setIndex("CityState")

  ; filter out everything but St. Thomas
  custTbl.setFilter("St. Thomas", "St. Thomas")

  ; open a TCursor for the filtered Customer table
  if tc.open(custTbl) then

    ; scan the table and load the DynArray with
    ; company names (Name) and phone numbers
    scan tc:
      dy[tc.Name] = tc.Phone
    endScan
    ; display contents of the DynArray
    dy.view("St. Thomas Phone Numbers")

  else
```

```
    msgStop("Error", "Can't open TCursor.")
  endif

else
  msgStop("Error", "Can't find Customer.db")
endif
endmethod
```

See also

- setFilter

setReadOnly

Table

Method

Specifies whether to give the user read-only rights to a table when it is opened.

Syntax

setReadOnly ([const ***yesNo*** Logical])

Description

Specifies whether to give the user read-only rights to a table when it is opened. This method fails if the table has been locked by another user or if the table is open.

Optional argument *yesNo* specifies whether to set read-only rights: a value of Yes grants read-only rights, a value of allows Full rights to the table. If omitted, *yesNo* is Yes by default.

Example

The following code attaches a Table variable to the *Orders* table, issues **setReadOnly** to limit user rights, then opens a TCursor for *Orders*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc TCursor
endVar

tblVar.attach("Orders.db")  ; attach Table var to Orders.db
tblVar.setReadOnly()        ; set Table to read-only
tc.open(tblVar)             ; open a TCursor for Orders.db

endmethod
```

See also

- setExclusive

showDeleted

Table

Method

Specifies whether to display deleted records in a dBASE table.

Syntax

showDeleted ([const ***yesNo*** Logical]) Logical

Description

Records deleted from a dBASE table aren't immediately removed. Instead, they are flagged for deletion and remain in the table. **showDeleted** specifies whether to display these records when the table is opened. **showDeleted** is relevant only for dBASE table.

Optional argument *yesNo* specifies whether to show deleted records (a value of Yes) or hide deleted records (a value of No). If omitted, *yesNo* is Yes by default. If you don't call this method before using the Table variable associated with the table, deleted records are not shown.

Example

For this example, the **pushButton** method attached to the *showDeletedRecs* button instructs a Table variable's deleted records be shown.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar

tblVar.attach("Orders.dbf")
if isTable(tblVar) then

  ; show deleted records in Orders.dbf
  tblVar.showDeleted(Yes)

  ; display sum of deleted and undeleted records
  msgInfo("Total # of Records", tblVar.nRecords())
else
  msgStop("Error", "Can't find Orders table.")
endif

endmethod
```

See also

- compact, delete

sort

Table

Keyword

Sorts a table.

Syntax

sort ***sourceTable***
  [on *fieldNameList* **[D]]**
  [to *destTable* **]**
endSort

Description

Fills in a sort form for the table specified in *sourceTable* and performs the sort.

sourceTable can be of type Table, TCursor, or String. *destTable* can be of type Table or String.

If you include the optional **on** clause, the table is sorted on the first field specified in *fieldNameList*. Each subsequent field is used, in turn, to settle ties in the preceding fields. An optional **D** after a field name specifies a sort in descending order. If you omit the **on** clause, records are sorted in ascending order, moving from left to right across the fields.

If you include the optional **to** clause, the result of the sort is written to the table described by *destTable*. If that table already exists, it is overwritten without asking for confirmation. If you omit the **to** clause, the sorted records are placed back *sourceTable* (this fails if the table is open). You must specify the **to** clause if the source table is keyed.

sort automatically places a full lock on tables being sorted if the result will be written to the same table. Otherwise, a write lock is required for the source table and a full lock for the destination table.

sort is not a method, so dot notation, as in the following statement, is inappropriate.

```
tableVar.sort()
```

Instead, you create a structure to specify how to sort the table.

Table

Example

This example sorts *Customer* on the Last Name and First Name fields, and places the results in in the *CustSort* table.

```
; sortCustTable::pushButtton
method pushButton(var eventInfo Event)
var
  custTbl Table
  tv TableView
endVar

custTbl.attach("Customer.db")

sort custTbl
  on "Last Name" D, "First Name" D     ; sort in descending order
  to "CustSort.db"
endSort

tv.open("CustSort.db")                 ; open the sorted table

endmethod
```

See also

- index, sortTo

subtract

Beginner

Table

Method/Procedure

Subtracts the records in one table from another table.

Syntax

1. **subtract** (const ***destTableName*** String) Logical
2. **subtract** (const ***destTableName*** Table) Logical

Description

Checks whether any records in the source table are also in *destTableName*. If so, **subtract** deletes them from *destTablename* without asking for confirmation.

If *destTableName* is not keyed, **subtract** deletes all records that exactly match any record in the source table. If *destTableName* is keyed, **subtract** deletes all records with keys that exactly match values in corresponding key fields in the source table. Whether *destTableName* is keyed or not, this method only considers fields that *could* be keyed. For example, numeric fields are considered, but formatted memos are not.

This method tries, for the duration of the retry period, to place a full lock on both tables. If locks cannot be placed, an error results.

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

The following code subtracts from *Customer* matching records found in the *Inserted* table in the private directory.

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
  insTbl, CustTbl Table
  fs FileSystem
  tblName String
endVar
tblName = privDir() + "\\Inserted.db"

insTbl.attach(tblName)
if insTbl.isTable() then
  insTbl.subtract(custTbl)     ; remove from custTbl matching records in insTbl
else
  msgInfo("Sorry", "Can't find " + tblName + " table.")
endif

endmethod
```

See also

- add

tableRights

Table

Method/Procedure

Specifies whether the user has rights to perform certain operations on a table.

Syntax

tableRights (const ***rights*** String) Logical

Description

Reports about a user's rights to a table, where *rights* is one of:

- "ReadOnly" (read from the table, but not change it)
- "Modify" (enter or change data)
- "Insert" (add new records)
- "InsDel" (add and delete records)
- "All" (perform all operations)

Note This method is also provided as a compatibility procedure. See Appendix E for more information.

Example

This example reports whether the user has "All" rights to the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myRights Logical
  ordTbl   Table
endVar

ordTbl.attach("Orders.db")
if ordTbl.isTable() then
  myRights = ordTbl.tableRights("All")

  ; this displays True if you have All rights to Orders.db
  msgInfo("All Rights?", myRights)

else
  message("Can't find Orders table.")
endif

endmethod
```

See also

- fieldRights

type

Table

Method

Returns the type of a table.

Syntax

type ( ) String

Description

Returns the string value "Paradox" or "dBASE" to report the type of a table.

For this example, assume a form has a button named *compactButton* and a table frame bound to the *Orders* table. The following code (attached to the **pushButton** method) compacts (removes deleted records) from the table if **type** returns dBASE; otherwise a message informs the user that *Orders* is a Paradox table.

```
; compactButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
endVar
tblVar.attach(ORDERS)
if tblVar.type() = "dBASE" then
  tblVar.compact()
else
  msgStop("Stop!", "Orders is a " + tblVar.type() + " table.")
endif

endmethod
```

See also

- attach

---

unAttach

Table

Method

Ends the association between a Table variable and a table on disk.

Syntax

unAttach ( ) Logical

Description

Ends the association (created using **attach**) between a Table variable and a table in a file. You don't have to end the association between a Table variable and a table to attach the same variable to another table—**unAttach** is automatically called when a Table variable is assigned to a different table.

Example

In this example, a single Table variable is used to summarize sales information from two different tables. Once the Table variable (*tableVar*) is no longer needed this code calls **unAttach** to end the association between *tableVar* and the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tableVar Table
  q1, q2   Number
  msg      String
```

```
endVar

tableVar.attach("q1 sales.db")  ; attach to q1 sales table
q1 = tableVar.cSum("Amount")    ; get a summary

tableVar.attach("q2 sales.db")  ; no need to unattach
q2 = tableVar.cSum("Amount")    ; get summary from q2 sales

tableVar.unAttach()             ; we don't need tableVar anymore
                                ; so end the association to q2 sales

if q1 = q2 then
  msg = "Sales must increase."
else
  msg = "Sales are up."
endIf
msgInfo("Sales", msg)

endmethod
```

See also

- attach

unlock

Table

Beginner

Method

Unlocks a specified table.

Syntax

unlock (const ***lockType*** String) Logical

Description

Attempts to remove locks explicitly placed on a table. *lockType* must be an expression that evaluates to one of the following string values: Write, Read, Full, or Any. If successful, this method returns True; otherwise, it returns False.

Example

For this example, the **pushButton** method for *updateCust* runs a query from an existing file, then adds records from the *Answer* table to the *Customer* table. This code attempts to place a write lock on the *Customer* table before adding records to it. If the lock succeeds, this code proceeds to add *Answer* records, then uses **unlock** to unlock *Customer*.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  newCust Query
  ansTbl Table
  destTbl String
endVar
destTbl = "Customer.db"

if executeQBEFile("getCust.qbe") then  ; if the query succeeds
  ansTbl.attach("Answer.db")
  if destTbl.lock("Write") then        ; attempt to write lock table
    ansTbl.add(destTbl)                ; add records from Answer.db
```

```
    destTbl.unLock("Write")          ; unlock the table
  else
    msgStop("Stop", "Can't write lock " + destTbl + " table.")
  endif
else
  msgStop("Stop!", "Query failed.")
endIf

endmethod
```

See also

- lock, lockStatus, unlockRecord

---

unProtect

Table

Method/Procedure

Decrypts and removes an owner password from a table.

Syntax

1. (Procedure) **unProtect** (const ***tableName*** String
   [, const ***Password*** String])
2. (Method) **unProtect** ([const ***password*** String])

Description

Permanently removes an owner password from a table. A protected table is encrypted and cannot be accessed without presenting the password specified in *password*. If you have already issued the master password for a table, *password* is not necessary.

Example

This example permanently removes password protection from the *Secrets* table.

```
; decrypt::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tblName String
endVar

tblName = "Secrets.db"
tblVar.attach(tblName)
if tblVar.isEncrypted() then
  tblVar.unprotect("Get007")    ; permanently remove password
                                ; this assumes Get007 is the master password
endif

endmethod
```

See also

- isEncrypted, protect
- addPassword, removePassword in the Session type

usesIndexes

Table

Method

Specifies index files to use and maintain with a dBASE table.

Syntax

usesIndexes (const ***indexFileName*** String
[, const ***indexFileName*** String]* Logical

Description

Specifies in *indexFileName* one or more index files (.NDX and .MDX) to use with a dBASE table. This method does not open the table, but specifies index files to open when the table is opened. You don't need to use this method to open production file (such as .MDX files) for a dBASE table—these files are automatically opened.

This method fails if all of the specified index files do not exist.

Example

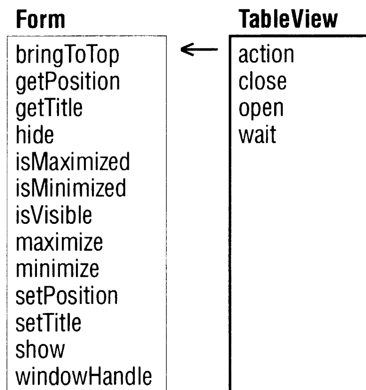
This example calls **usesIndexes** to specify two different indexes in the *Orders* table, then opens a TCursor for the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  tc     TCursor
endvar

tblVar.attach("Orders.dbf")
if tblVar.isTable() then

  ; specify NameStat and Ord_Name indexes
  tblVar.usesIndexes("NAMESTAT.NDX", "ORD_NAME.NDX")

  ; now attempt to open the table, using the specified indexes
  if tc.open(tblVar) then
    if tc.locate("State", "FL", "Contact", "Simons") then
      msgInfo("Order Date", tc."Order Date")
    else
      msgStop("Error", "Can't find values.")
    endif
  endif
else
  msgStop("Error", "Can't find Orders.dbf table.")
endif
endmethod
```

See also

- reIndex, reIndexAll, setIndex

TableView



A TableView object displays the data in a table in its own window. A TableView object is distinct from a table frame, which is a UIObject placed in a form, and from a TCursor, which points to the data in a table.

When you declare a TableView variable, then open a table using that variable, you create a handle to the table window window, something you can refer to in your code to manipulate the table window.

TableView methods are a subset of the methods for the Form type. You can use them to control the table window's size, position, and appearance. Although you can start and end Edit mode for a table window, you cannot use ObjectPAL to directly edit the data in a table window.

You can use ObjectPAL to manipulate TableView properties in three main areas:

- The table window as a whole—for example, background color, grid style, and number of records
- The field-level data in the table —for example, font, color, display format, and the value of the current record
- The table window heading—for example, font, color, and alignment

See the TableView, TVData, and TVHeading categories in the Properties dialog box for a list of table window properties.

For more information and examples, refer to Chapters 8 and 10 in the *ObjectPAL Developer's Guide.* The TableView type includes several methods defined for the Form type.

action

TableView

Method

Performs an action command.

Syntax

action (const ***actionID*** SmallInt) **Logical**

Description

Performs the action represented by the constant *actionId*. ObjectPAL provides constants for actionID; see one of the Action categories (for example, ActionSelectCommands) in the Constants dialog box.

Example

This example opens a table view for the *Orders* table, moves the cursor to the end of the table, starts Edit mode, and inserts a new blank record. This code is attached to the **pushButton** method for a button named *startEditInsert*.

```
; startEditInsert::pushButton
method pushButton(var eventInfo Event)
var
  orderTV  TableView
endVar
if orderTV.open("Orders") then
  orderTV.action(DataEnd)              ; move to the end of the table
  orderTV.action(DataBeginEdit)        ; start Edit mode
  orderTV.action(DataInsertRecord)     ; insert a new blank record
  orderTV.wait()                       ; wait until TableView object is closed
  orderTV.close()                      ; close when return
else
  msgStop("Status", "Could not find Orders table.")
endif
endmethod
```

See also

- The discussion of the built-in **action** method in Chapter 2

close

TableView

Method

Closes a table window.

Syntax

close ( )

Description

Closes a table window; it is equivalent to choosing Close from the Control menu.

Example

In this example, the **open** method for a form opens a TableView object for the *Customer* table to the global variable *custTV*. When the form closes, the **close** method for the form closes the *custTV* TableView. This code is attached to the **close** method for the form:

```
; thisForm::close
method close(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    custTV.close()      ; close the Customer table that was
                        ; opened by thisForm's open method
endif
endmethod
```

This is the code for the form's Var window.

```
; thisForm::Var
Var
  custTV  TableView    ; global to form, the TableView object is opened by
                       ; form's open method
endVar
```

This is the code for the form's **open** method.

```
; thisForm::open
method open(var eventInfo Event)
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
  else
    ; code here executes just for form itself
    custTV.open("Customer")  ; open the Customer table view
endif
endmethod
```

See also

- open

---

open

TableView

Method

Opens a table window.

Syntax

1. **open** (const ***tvName*** String [, const ***windowStyle*** LongInt]) Logical
2. open (const ***tvName*** String, const ***windowStyle*** LongInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt) Logical

Description

Displays the table specified in *tvName* in a table window. Optional arguments specify (in twips) the location of the upper left corner of the form (*x* and *y*), the width and height (*w* and *h*), and style (*windowStyle*). The *windowStyle* argument is ignored, but required for syntax 2. If you want to specify a size and position, you can use a window style constant of *WinStyleDefault*.

Example

In the following example, the **pushButton** method for a button named *openWaitOrders* opens the *Orders* table, then waits until the user closes the table window.

```
; openWaitOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordersTV  TableView
endVar
if ordersTV.open("Orders", WinStyleDefault, 100, 100,
                 1440*5, 1440*4) then
  ordersTV.wait()    ; wait for user to close
  ordersTV.close()   ; close Orders table
endif
endmethod
```

See also

- close

wait

TableView

Method

Suspends execution of a method.

Syntax

wait ( )

Description

Suspends execution of a method. Execution resumes when the TableView object is closed. Note that you must follow a **wait** with a **close**. When a TableView object has been called by **wait**, the calling method suspends execution until the TableView object is closed by the user.

Example

See the example for open.

See also

- close, open

TCursor

TCursor

add	enumIndexStruct	locatePriorPattern
atFirst	enumLocks	lock
atLast	enumRefIntStruct	lockRecord
attach	enumSecStruct	lockStatus
attachToKeyViol	enumTableProperties	moveToRecNo
bot	eot	moveToRecord
cancelEdit	familyRights	nextRecord
cAverage	fieldName	nFields
cCount	fieldNo	nKeyFields
close	fieldRights	nRecords
cMax	fieldSize	open
cMin	fieldType	postRecord
cNpv	fieldUnits2	priorRecord
compact	fieldValue	qLocate
copy	getLanguageDriver	recNo
copyFromArray	getlanguageDriverDesc	recordStatus
copyRecord	home	reIndex
copyToArray	initRecord	reIndexAll
cSamStd	insertAfterRecord	seqNo
cSamVar	insertBeforeRecord	setFieldValue
cStd	insertRecord	setFilter
cSum	isAssigned	setFlyAwayControl
currRecord	isEdit	showDeleted
cVar	isEmpty	skip
delete	isEncrypted	sortTo
deleteRecord	isRecordDeleted	subtract
didFlyAway	isShared	switchIndex
dropIndex	isShowDeletedOn	tableName
edit	isValid	tableRights
empty	locate	type
end	locateNext	undeleteRecord
endEdit	locateNextPattern	unlock
enumFieldNames	locatePattern	unlockRecord
enumFieldNamesinIndex	locatePrior	updateRecord
enumFieldStruct		

A TCursor is a pointer to the data in a table, enabling you to manipulate data without having to display the table. It is not a clone or a copy of the table—editing records in a TCursor changes the underlying table, and any locks on the table affect the TCursor.

For more information and examples, refer to Chapter 10 in the *ObjectPAL Developer's Guide.*

For information about related objects, refer to the Table and TableView types.

add

TCursor

Method

Adds the records of one table to another.

Syntax

1. **add** (const ***destTable*** String [, const ***append*** Logical [, const ***update*** Logical]]) Logical
3. **add** (const ***destTable*** Table [, const ***append*** Logical [, const ***update*** Logical]]) Logical
2. **add** (const ***destTable*** TCursor [, const ***append*** Logical [, const ***update*** Logical]]) Logical

Description

Adds the records pointed to by a TCursor to the destination table specified in *destTable.* If the destination does not exist, this method creates it. The source table and the destination table can be the same type or different types; in any case, the tables must have compatible field structures.

Arguments *append* and *update* can be True or False. When True, *append* adds records at the end of a non-indexed table, or at the appropriate places in an indexed table. When True, *update* compares records in both tables, and where key values match, replaces the data in the destination table. When both are True, records with matching key values are updated, and others are appended. These arguments are optional, but if you specify *update,* you must also specify *append.* If omitted, both are True. Here are some example statements:

```
myTCursor.add(yourTable, False, True) ; specifies update
myTCursor.add(yourTable) ; specifies update and append by default
```

When tables are indexed, **add** uses the indexed fields to determine which records to update and which to append. When the destination table is not indexed, **add** fails if *update* is True. Key violations, if any, are listed in KEYVIOLS.DB in the user's private directory. This method overwrites an existing KEYVIOLS.DB or creates one, if necessary. **add** does not respect the limits of restricted views displayed in a linked table frame or multi-record object.

This method tries, for the duration of the retry period, to place write locks on the source table and the destination table. If either lock cannot be placed, the method fails.

Example

In this example, assume the *OldCust* and *NewCust* tables exist in the current directory. The following code associates a TCursor with each of the tables, adds *NewCust* records to *OldCust,* then adds all records to a table named *MyCust.* If *MyCust* does not exist in the current directory, **add** creates it. This code is attached to a button's **pushButton** method.

```
; getMyCust::pushButton
method pushButton(var eventInfo Event)
var
  dTC, sTC TCursor
endVar

if sTC.open("OldCust.db") and
   dTC.open("NewCust.db") then  ; if both TCursors can be associated
   dTC.add(sTC, True)           ; append oldCust records to newCust table
                                ; now sTC has records from both tables

   sTC.add("MyCust.db", True)   ; add sTC records to MYCUST.DB table

   sTC.close()                  ; close both TCursors
   dTC.close()
else
   msgStop("Stop!", "Could not open one or more tables.")
endif

endmethod
```

See also

- copy, subtract

atFirst

TCursor

Method

Reports whether the TCursor is pointing to the first record of a table.

Syntax

atFirst ( ) Logical

Description

Returns True if the TCursor is pointing to the first record of a table; otherwise, it returns False.

Example

This example assumes a form has a button named *moveToFirst*, and a multi-record object bound to ORDERS.DB. The code attached to the **pushButton** method for *moveToFirst* uses **atFirst** to determine if the TCursor is at the first record. If it isn't, this code moves to the first record.

```
; moveToFirst::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.attach(ORDERS)          ; orders is a multi-record object
if not tc.atFirst() then   ; if not at the first record
  tc.home()                ; move to the first record
  orders.moveToRecord(tc)  ; move highlight to first record
else
  msgStop("Currently on record " + String(tc.recNo()),
          "You're already on the first record!")
endif
endmethod
```

See also ❒ atLast, bot, eot

atLast

TCursor

Method Reports whether the TCursor is pointing to the last record of a table.

Syntax **atLast ( )** Logical

Description Returns True if the TCursor is pointing to the the last record of a table; otherwise, it returns False.

Example This example assumes a form has a button named *moveToLast*, and a multi-record object bound to ORDERS.DB. The code attached to the **pushButton** method for *moveToLast* uses **atLast** to determine if the TCursor is on the last record. If it isn't, this code moves to the last record.

```
; moveToLast::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.attach(ORDERS)
if not tc.atLast() then   ; if not on the last record
  tc.end()                ; move TCursor to the last record
  orders.moveToRecord(tc) ; move highlight to the last record
else
  msgStop("Currently on record " + String(tc.recNo()),
          "You're already at the last record!")
endif
endmethod
```

See also ❒ atFirst, bot, eot

attach

TCursor

Method Binds a TCursor to a UIObject.

Syntax

1. **attach** (const ***object*** UIObject) Logical
2. **attach** (const ***srcTCursor*** TCursor) Logical
3. **attach** (const ***tv*** TableView) Logical

Description Binds a TCursor to a UIObject (*object* in syntax 1), another TCursor (*srcTCursor* in syntax 2) or a TableView object (*tv* in syntax 3). The

TCursor

data comes from the underlying table—the TCursor gets no data from records that have not been committed (for example, because the record is being edited or has just been added). **attach** returns True if successful; otherwise, it returns False.

For more information and examples, refer to Chapter 10 in the *ObjectPAL Developer's Guide*.

Example

In this example, assume a form contains a table frame bound to ORDERS.DB, and another table frame bound to LINEITEM.DB. The *Orders* table has a one-to-many link to *LineItem*. A button named *findDetails* is also on the form. Suppose you want the user to be able to search through the entire *LineItem* table—not just those records linked to the current Order. In this case, the **pushButton** method for *findDetails* searches for orders that include the current part number.

This code is attached to the Var window for the *findDetails* button:

```
; findDetails::Var
Var
  lineTC TCursor  ; instance of LINEITEM for searching
endVar
```

The code that follows is attached to the **open** method for the *findDetails* button. This code associates the *lineTC* TCursor with LINEITEM.DB.

```
; findDetails::open
method open(var eventInfo Event)
lineTC.open("LineItem.db")
endmethod
```

The following code is attached to the **pushButton** method for *findDetails*:

```
; findDetails::pushButton
method pushButton(var eventInfo Event)
var
  stockNum  Number
  orderTC   TCursor
  OrderNum  Number
endVar

stockNum = LINEITEM.Stock_No  ; get Stock No from current LineItem
; lineTC was declared in Var window and opened by open method
if NOT lineTC.locateNext("Stock No", stockNum) then
  lineTC.locate("Stock No", stockNum)
endif
orderTC.attach(ORDERS)       ; attach TCursor to table frame
orderTC.locate("Order No", lineTC."Order No")
ORDERS.moveToRecord(orderTC) ; move to CUSTOMER and
                             ; resynchronize with TCursor
LINEITEM.moveTo()            ; move TCursor to LINEITEM detail
; move TCursor to matching record
LINEITEM.locate("Stock No", stockNum)
endmethod
```

This code is attached to the **close** method for *findDetails*:

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close()    ; close the TCursor to LineItem
endmethod
```

See also

- isValid

attachToKeyViol

TCursor

Method

Attaches a TCursor to the existing record that has the same key as the record you attempted to post.

Syntax

attachToKeyViol (const ***oldTC*** TCursor) Logical

Description

After a key violation occurs, **attachToKeyViol** attaches a TCursor to the existing record—the record that existed before the key violation occurred. Specify in *oldTC* the TCursor that points to the record that caused the key violation (the new, unposted record).

This method gives you a way to compare conflicting records before replacing or discarding a change to an existing record. *oldTC* must already be pointing to the new (yet unposted) record.

Example

This example demonstrates how **attachToKeyViol** can be used after a key violation occurs. The code declares two TCursors: *keyViolTC* and *originalRecTC*. The code opens *keyViolTC* for the *Orders* table, then deliberately inserts a record whose key value conflicts with another record. Then, the example attempts to post the new record to the table, which forces a key violation. At this point, if the user chooses to view the existing record, the code calls **attachToKeyViol**, attaches the second TCursor (*originalRecTC*) to the original record, and displays the record in a **view** dialog box. If the user chooses to update the original record with data from the new record, this example calls **updateRecord** method to do so; otherwise, the code makes no changes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  keyViolTC, originalRecTC TCursor
  rec DynArray[] AnyType
endvar

keyViolTC.open("Orders.db")          ; open TCursor for Orders
keyViolTC.edit()                     ; put TCursor in Edit mode
keyViolTC.insertRecord()             ; insert a new record
keyViolTC."Order No" = 1011          ; 1011 is a duplicate key

; if this attempt to post the new record fails
if NOT keyViolTC.postRecord() then
```

```
  ; attach originalRecTC to the existing record
  originalRecTC.attachToKeyViol(keyViolTC)

  ; give user the option to see the existing record
  if msgQuestion("Key Exists!",
     "Do you want to see the existing record?") = "Yes" then

    originalRecTC.copyToArray(rec)   ; copy existing record to rec
    rec.view("Original Record")      ; display rec in a dialog box

  endif

  ; give user the option to replace the existing record
  if msgQuestion("Confirm Update",
     "Do you want to replace existing record?") = "Yes" then

    ; force the new record to post
    keyViolTC.updateRecord(True)
  else
    message("Original record left intact.")
    sleep(1500)
  endif
else
  message("Posted order number 1011.")
endif

endmethod
```

See also

- postRecord, updateRecord

---

bot

TCursor

Method

Tests for a move past the beginning of a table.

Syntax

bot ( ) Logical

Description

Returns True if a command attempts to move past the first record of a table; otherwise, it returns False. **bot** is reset by the next move operation.

Example

This example moves a TCursor backwards through a table then displays a message. This code is attached to a button's **pushButton** method.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("sites.db")
tc.end()                    ; moves to end of table
while tc.bot() = False      ; loop until we hit the top
  tc.priorRecord()          ; move backwards through table
endWhile
```

```
msgInfo("The Top", "TCursor is on the first record.")

endmethod
```

See also

- atFirst, atLast, end, eot, home

---

cancelEdit

TCursor

Beginner

Method

Ends Edit mode without saving changes to the current record.

Syntax

cancelEdit ( ) Logical

Description

Takes a TCursor out of Edit mode without saving changes to the current record. You must use **cancelEdit** before moving the TCursor from the current record or otherwise committing or unlocking the record. Once you move the TCursor, changes to the record are committed.

Example

The code for this example is attached to the **pushButton** method for the *changeKey* button. This example associates a TCursor with the *Customer* table then attempts to change a value in a keyed field. If the record can not be successfully posted (for example, because of a key violation) this example displays an error message, then calls **cancelEdit** to restore the record to the original values and end Edit mode.

```
; changeKey::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  rec Array[] AnyType
endVar

tc.open("Customer.db")                     ; open TCursor for Customer
if tc.locate("Customer No", 1231) then     ; if 1231 exists in Customer No field
  tc.edit()                                ; go into edit mode
  tc."Customer No" = 1221                  ; attempt to change key value
  if not tc.endEdit() then                 ; if endEdit fails
    msgStop("Error", "Can't complete operation.")
    tc.cancelEdit()                        ; restore record and leave edit mode
    message("Record left intact.")
  else
    message("Key value changed.")
  endif
else
  msgStop("Error", "Can't find Customer 1231")
endif

endmethod
```

See also

- edit, endEdit

TCursor

cAverage

TCursor

Method

Returns the average value of a field (column) in a table.

Syntax

1. **cAverage** (const ***fieldName*** String) Number
2. **cAverage** (const ***fieldNum*** SmallInt) Number

Description

Returns the average of values in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cAverage** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

This example uses **cAverage** to calculate the average order size in the *Orders* table. This code is attached to the **pushButton** method for the *getAvgSales* button.

```
; getAvgSales::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
  avgSales, avgOrders Number
endVar

; open TCursor for ORDERS table
ordTC.open("Orders.db")
; store average invoice total in avgSales variable
avgSales = ordTC.cAverage("Total Invoice")
; display avgSales in a dialog
msgInfo("Average Order size", avgSales)

endmethod
```

See also

- cCount, cMax, cMin, cStd, cSum
- blankAsZero in the Session type

cCount

TCursor

Method

Returns the number of values in a field (column) of a table.

Syntax

1. **cCount** (const ***fieldName*** String) Number
2. **cCount** (const ***fieldNum*** SmallInt) Number

Description

Returns the number of values in the column (field) specified by *fieldName* or *fieldNum*. **cCount** works for all field types. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. If the field is non-numeric, **cCount** returns the number of non-blank values in the column of fields.

This method respects the limits of restricted views displayed in a linked table frame or multi-record object.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

cCount is useful for returning the number of entries used by another column function.

Example

This example opens a TCursor for a table, then uses **cCount** to display the number of records in the TCursor. This code is attached to the **pushButton** method for the *lineItemInfo* button.

```
; lineItemInfo::pushButton
method pushButton(var eventInfo Event)
var
  numbersTC TCursor
  avgQty, numRecs Number
endVar
numbersTC.open("Lineitem.db")
avgQty = numbersTC.cAverage("Qty")
numRecs = numbersTC.cCount(4)        ; assumes Quantity is field 4
msgInfo("Average quantity", "Average quantity: " +
String(avgQty) + " \nbased on " + String(numRecs) + " records.")

endmethod
```

See also

- cAverage, cMax, cMin, cStd, cSum
- blankAsZero in the Session type

close

Beginner

TCursor

Method

Closes a TCursor.

Syntax

close ( ) Logical

Description

Closes a TCursor, and makes the TCursor variable unassigned. If the current record cannot be committed, **close** still closes the TCursor, but discards any changes to the record.

Example

This example opens a TCursor for a table, displays information found in the last record, then closes the TCursor. In this example, the code displays a message indicating whether the TCursor variable is still assigned after the TCursor is closed. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.open("Orders.db")  ; open TCursor for the Orders table
tc.end()              ; move to the end of the table

; display information in the last record
msgInfo("Last Order", "Order number: " + String(tc."Order No") +
       " \nOrder date: " + String(tc."Sale Date"))

tc.close()                                 ; close tc TCursor
msgInfo("Is tc Assigned?", tc.isAssigned())  ; displays False

endmethod
```

See also

- isAssigned, open

cMax

TCursor

Method

Returns the maximum value of a field (column) in a table.

Syntax

1. cMax (const ***fieldName*** String) Number

2. cMax (const ***fieldNum*** SmallInt) Number

Description

Returns the maximum value in the column of numeric fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cMax** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

In the following example, assume a form has a button, *getMaxBalance*, and a table frame bound to the *Orders* table. In this code, the **pushButton** method for *getMaxBalance* associates the table frame with a TCursor, then locates the highest balance due in the *Orders* table:

```
; getMaxBalance::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
```

```
endVar

ordTC.attach(ORDERS)   ; ORDERS is a table frame on the form

; now locate the maximum value in the "Balance Due" field
ordTC.locate("Balance Due", ordTC.cMax("Balance Due"))
; syncronize the table frame to the TCursor
ORDERS.moveToRecord(ordTC)

endmethod
```

See also

- cAverage, cCount, cMin, cStd, cSum
- blankAsZero in the Session type

cMin

TCursor

Method

Returns the minimum value in a field (column) of a table.

Syntax

1. cMin (const ***fieldName*** String) Number

2. cMin (const ***fieldNum*** SmallInt) Number

Description

Returns the minimum value in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cMin** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

This example uses both forms of the syntax to calculate minimum values in the ORDERS.DB table:

```
; showMinimums::pushButton
method pushButton(var eventInfo Event)
var
  OrdTC TCursor
  minBalDue, minOrder Number
endVar
OrdTC.open("Orders.db")
minBalDue = ordTC.cMin("Balance Due")  ; get minimum balance due
minOrder  = ordTC.cMin(6)              ; assumes "Total Invoice" is field 6

; display results in a dialog box
msgInfo("Minimums", "Minimum balance due: " + String(minBalDue) + "\n" +
                    "Minimum order : " + String(minOrder))
endmethod
```

See also

- cAverage, cCount, cMax, cStd, cSum
- blankAsZero in the Session type

cNpv

TCursor

Method

Returns the net present value of a field (column), based on a specified discount or interest rate.

Syntax

1. **cNpv** (const ***fieldName*** String, const ***discRate*** Number) Number
2. **cNpv** (const ***fieldNum*** SmallInt, const ***discRate*** Number) Number

Description

Returns the net present value of the nonblank entries in a column of fields. The calculation is based on the interest or discount rate *discRate*, where *discRate* is a decimal number (for example, 12 percent is expressed as .12). This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a lock on the table. If a lock cannot be placed, the method fails.

This method calculates net present value using the following formula:

$$CPNV = \sum_{p=1}^{N} \frac{Vp}{(1+i)^p}$$

cNpv = sum(p=1 to n) of $Vp/(1+i)^p$

where n = number of periods, Vp = cash flow in *pth* period, and i = interest rate per period.

Example

The following example associates a TCursor with the *GoodFund* table, then calculates the net present value for the *Expected Return* field. In this example, the net present value is calculated based on a monthly interest rate. This code is attached to the **pushButton** method for the *calcNPV* button.

```
; calcNPV::pushButton
method pushButton(var eventInfo Event)
var
  SavingsTC TCursor
  goodFundNPV, apr Number
endVar
SavingsTC.open("GoodFund.db")  ; associate TCursor with Savings table
apr = .125                     ; annual percentage rate

; now calculate net present value based on monthly interest rate
goodFundNPV = SavingsTC.cNpv("Expected Return", (apr / 12))
msgInfo("Net present value", goodFundNPV)
```

```
endmethod
```

See also

- cAverage, cMax, cMin, cStd, cSum, cVar
- blankAsZero in the Session type

compact

TCursor

Method

Removes deleted records from a table.

Syntax

compact ([const ***regIndex*** Logical]) Logical

Description

Removes deleted records from a table. Deleted records are not immediately removed from a table. Instead, they are flagged as deleted and kept in the table. The optional argument *regIndex* is used to specify whether to regenerate indexes associated with the table, or simply to fix them. When *regIndex* is True, this method regenerates all maintained indexes associated with the TCursor and frees any unused space in the indexes. If omitted, *regIndex* is True by default.

If the table you're working with has maintained indexes, this method requires exclusive access; otherwise it requires a write lock.

When records are deleted from a Paradox table, they can no longer be retrieved—they are permanently deleted. However, the table file (and associated index files) contain "dead" space where the record was originally stored. **compact** removes dead space from Paradox files.

Example

The following example inspects each record in a table and deleted records where the value of the Date field is more than 30 days old. It compacts the table after appropriate records have been deleted. This code is attached the **pushButton** method for the *purgeTable* button.

```
; purgeTable::pushButton
method pushButton(var eventInfo Event)
var
  tb Table
  tc TCursor
  recsDeleted SmallInt
endVar
tb.attach("OldData.dbf")
tb.setExclusive()          ; get exclusive rights to table
tc.open(tb)                ; associate TCursor with OldData table
tc.edit()
scan tc for tc.Date  (today() - 30) :
  tc.deleteRecord()        ; delete old records
endScan
tc.compact()               ; remove all deleted records
tc.close()                 ; close TCursor
```

```
message("Old records purged.")
endmethod
```

See also

- deleteRecord, showDeleted

---

copy

TCursor

Method

Copies a table.

Syntax

1. copy (const ***destTableName*** String) Logical
2. copy (const ***destTableName*** Table) Logical

Description

Copies a table to the destination table *destTableName*. If *destTableName* does not exist, **copy** creates it. If *destTableName* already exists, **copy** overwrites it without asking for confirmation.

This method tries, for the duration of the retry period, to place a write lock on the source table and a full (exclusive) lock on the destination table. This method fails if either lock cannot be placed, or if the destination table is open.

This method does not respect the limits of restricted views displayed in a linked table frame or multi-record object.

Example

This example copies the *Customer* table to the *NewCust* table.

This code uses the **isTable** method (from the DataBase type) to test whether *NewCust* exists; if it does, the user is prompted to confirm the action before *NewCust* is overwritten:

```
; copyCust::pushButton
method pushButton(var eventInfo Event)
var
  sourceTC TCursor
  destTb Table
endVar
destTb.attach("NewCust.db")
sourceTC.open("Customer.db")

; if NewCust.db exists, ask for confirmation
if isTable(destTb) then
  if msgYesNoCancel("Copy table", "Overwrite Newcust.db?") = "Yes" then

    ; copy Customer.db records to NewCust.db
    sourceTC.copy(destTb)
  endIf
endIf

endmethod
```

See also

- add, copyRecord

copyFromArray

TCursor

Method

Copies data from an array to the fields of the current record.

Syntax

1. **copyFromArray** (const ***ar*** Array[] AnyType) Logical
2. **copyFromArray** (const ***ar*** DynArray[] AnyType) Logical

Description

Copies the elements of the array *ar* to the record pointed to by a TCursor, which must be in Edit mode. The first element of the array is copied to the first field, the second element to the second field, and so on until the array is exhausted or the record is full.

The method fails if an attempt is made to copy an unassigned array element or if the structures do not match. (This can never happen if the array was created by **copyToArray**, because **copyToArray** assigns a blank value if a field is blank.) If there are more elements in the array than fields in the record, the extra elements are ignored. To copy a new record into an empty table, use **insertRecord** to insert a blank record before using **copyFromArray**.

Example

In this example, suppose CUSTNAME.DB has three fields: Last Name, A20; First name, A20, and Telephone, A12. This method associates a TCursor with the *CustName* table, creates an array with three elements, inserts a new record in the table, then uses **copyFromArray** to copy data from the array to the new record.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("CustName.db") then  ; open table
  tc.edit()                     ; copyFromArray only works in Edit mode
  tc.insertRecord()             ; insert new record
  tc.copyFromArray(aa)          ; copy from array to table
  tc.endEdit()
else
  msgStop("Stop", "Couldn't open CustName.db.")
endIf
endmethod
```

See also

- copyRecord, copyToArray, insertRecord, insertAfterRecord, insertBeforeRecord

TCursor

copyRecord

TCursor

Method

Copies a record pointed to by one TCursor into the record pointed to by another TCursor.

Syntax

copyRecord (const ***pointer*** TCursor) Logical

Description

Copies the record pointed to by one TCursor into the record pointed to by another TCursor. For example, the following code copies the record pointed to by the *copyFromThisTC* TCursor into the record pointed to by the *pointerTC* TCursor:

```
copyToThisTC.copyRecord(pointerTC)
```

The TCursor specified in *pointer* does not have to be in Edit mode; the TCursor you're copying to does.

You can not use **copyRecord** to copy a record into an empty table. To copy a new record into an empty table, use **insertRecord**.

Example

This example uses a TCursor to scan the *Orders* table for sales posted in the last 10 days and copies them to the *NewOrdrs* table in the current directory. This code is attached to the **pushButton** method for the *getNewOrders* button.

```
; getNewOrders::pushButton
method pushButton(var eventInfo Event)
var
  ordTC, newOrdTC TCursor
  ui TableView
endVar

ordTC.open("Orders.db")
newOrdTC.open("NewOrdrs.db")
newOrdTC.edit()                 ; copyRecord only works in Edit mode

; scan Orders.db table for records dated ten days ago and later
scan ordTC for ordTC."Sale Date" = today() - 10:
  newOrdTC.insertRecord()       ; insert a new record in NewOrdrs.db
  newOrdTC.copyRecord(ordTC)    ; copy record from Orders.db into NewOrdrs.db
endScan
newOrdTC.endEdit()              ; end Edit mode for TCursor

ui.open("NewOrdrs.db")
endmethod
```

See also

- copyToArray, insertAfterRecord, insertBeforeRecord, insertRecord

copyToArray

TCursor

Method

Copies the fields of the current record to an array.

Syntax

1. **copyToArray** (var ***ar*** Array[] AnyType) Logical
2. **copyToArray** (var ***ar*** DynArray[] AnyType) Logical

Description

Copies the fields of the current record to the elements of an array specified in *ar*. You must declare the array to be of type AnyType, or of a type that matches every field in the table.

In syntax 1, where *ar* is a fixed or resizeable array, the value of the first field is copied to the first element of the array, the value of the second field to the second element, and so on. If the array is resizeable, it grows automatically to hold the number of fields in the record. If the array is not resizeable, it holds as many fields as it can, and the rest are discarded.

If syntax 2, where *ar* is a DynArray, index values correspond to the field names and DynArray values correspond to field values:

ar [*fieldName*] = *fieldValue*

The size of the array is equal to the number of fields in the record (unless *ar* is a fixed array). The record number field and any display-only or calculated fields that appear in a form window of the table are not copied to the array.

Example

In this example, assume a form has a table frame, CUSTOMER, bound to CUSTOMER.DB. When the user attempts to delete a CUSTOMER record, this code (attached to the built-in **action** method) uses **copyToArray** and **copyFromArray** to copy the record to an archive table, CUSTARC.DB. If CUSTARC.DB cannot be opened, this method informs the user and does not delete the record.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcOrig, tcArc TCursor
  arcRec Array[] AnyType
endVar

if eventInfo.id() = DataDeleteRecord then ; when user to deletes a record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't delete the record

                                          ; ask for confirmation
    if msgQuestion("Confirm", "Delete record?") = "Yes" then

      tcOrig.attach(CUSTOMER)             ; sync TCursor to UIObject
      tcOrig.copyToArray(arcRec)          ; store the record in arcRec
      if tcArc.open("CustArc.db") then    ; True if tcArc can open CustArc
```


TCursor

```
      tcArc.edit()                    ; copyFromArray requires Edit
      tcArc.insertAfterRecord()       ; create a new record
      tcArc.copyFromArray(arcRec)     ; copy arcRec to new record
      enableDefault                   ; delete the record in Customer
    else                              ; can't open Customer TCursor
      msgStop("Stop!", "Sorry, Can't archive record.")
    endif
  else                                ; user didn't confirm dialog box
    message("Record not deleted.")
  endif

else                                  ; not in Edit mode
  msgStop("Stop!", "Press F9 to edit data.")
endif
endif
endmethod
```

See also

- copyFromArray

cSamStd

TCursor

Method

Returns the sample standard deviation of a field (column) of a table.

Syntax

1. **cSamStd** (const ***fieldName*** String) Number
2. **cSamStd** (const ***fieldNum*** SmallInt) Number

Description

Returns the sample standard deviation of values in a column of numeric fields. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. The returned value is based on the sample variance. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a lock on the table. If a lock cannot be placed, the method fails.

The sample standard deviation (as opposed to population) is calculated using this formula:

sqrt(*TCursor*.cVar(*FieldName*) * (n/(n - 1)))

where

variance = *TCursor*.cVar(*fieldName*)

n = *TCursor*.cCount(*fieldName*)

Example

The following example uses both forms of the syntax to calculate the sample standard deviation of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamStd*:

```
; showSamStd::pushButton
method pushButton(var eventInfo Event)
```

```
var
  empTC TCursor
  tblName String
  CalcSalary, CalcYears Number
endVar
tblName = "Answer"
if empTC.open(tblName) then
  CalcSalary = empTC.cSamStd("Salary")  ; get sample std deviation for salaries
  CalcYears  = empTC.cSamStd(2)         ; assume "Years in service" is field 2
  msgInfo("Sample Std Deviation",       ; display info in a dialog box
          "Salaries : " + String(CalcSalary) + "\n" +
          "Years in service : " + String(CalcYears))
else
  msgInfo("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamVar, cStd, cSum, cVar

---

cSamVar

TCursor

Method

Returns the sample variance of a field (column) in a table.

Syntax

1. **cSamVar** (const ***fieldName*** String) Number
2. **cSamVar** (const ***fieldNum*** SmallInt) Number

Description

Returns the sample variance of the values in a column of fields. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

The sample variance (as opposed to population) is calculated using this formula:

$$\sqrt{(variance) * \left(\frac{n}{n-1}\right)}$$

where

variance = TCursor.cVar(fieldName)

n = *TCursor*.cCount(*fieldName*)

Example

The following example uses both forms of the syntax to calculate the sample variance of two different fields in the *Answer* table. This code is attached to the **pushButton** method for *showSamVar*.

TCursor

```
; showSamVar::pushButton
method pushButton(var eventInfo Event)
var
  empTC TCursor
  tblName String
  CalcSalary, CalcYears Number
endVar
tblName = "Answer"
if empTC.open(tblName) then
  CalcSalary = empTC.cSamVar("Salary") ; get sample variance for salaries
  CalcYears  = empTC.cSamVar(2)        ; assume "Years in service" is field 2
  msgInfo("Sample Variance",           ; display info in a dialog box
          "Salaries : " + String(CalcSalary) + "\n" +
          "Years in service : " + String(CalcYears))
else
  msgInfo("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cStd, cSum, cVar

cStd

TCursor

Method

Returns the population standard deviation of a field (column) in a table.

Syntax

1. **cStd** (const ***fieldName*** String) Number
2. **cStd** (const ***fieldNum*** SmallInt) Number

Description

Returns the population standard deviation of the values in a column of numeric fields. The calculation is based on the variance. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

In this example, the **pushButton** method for *thisButton* calculates the population standard deviation for two separate fields and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  test1, test2 Number
endVar
tc.open("scores.dbf")
test1 = tc.cStd("Test1")
test2 = tc.cStd(2)                ; assumes Test2 is field 2

; show results in a dialog
```

```
msgInfo("Standard Deviation",
        "Test1 results : " + String(test1) + "\n" +
        "Test2 results : " + String(test2))

endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cSum, cVar

cSum

TCursor

Method

Returns the sum of the value in a field (column) of a table.

Syntax

1. **cSum** (const ***fieldName*** String) Number
2. **cSum** (const ***fieldNum*** SmallInt) Number

Description

Returns the sum of the values in a column of numeric fields. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. This method handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

In this example, the **pushButton** method for *sumOrders* uses both forms of **cSum** syntax to calculate totals for two fields in ORDERS.DB:

```
; sumOrders::pushButton
method pushButton(var eventInfo Event)
var
  orderTC TCursor
  orderTotal, amtPaid Number
  tblName String
endVar
tblName = "Orders"
if orderTC.open(tblName) then
  orderTotal = orderTC.cSum("Total Invoice")  ; get sum for Total Invoice field
  amtPaid    = orderTC.cSum(7)                ; assumes Amount Paid is field 7
  msgInfo("Order Totals",
         "Total Orders : " + String(orderTotal) + "\n" +
         "Total Receipts : " + String(amtPaid))
else
  msgInfo("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cStd, cVar

currRecord

TCursor

Method

Reads the current record into the record buffer.

Syntax

currRecord ( ) Logical

Description

Reads values of the current record into the record buffer. This method ensures you're working with the most recently updated version of the record, particularly on a network.

Example

This example copies a record from the *CustBak* table and inserts it into the *Customer* table. Before posting the new *Customer* record, the code gives the user a chance to confirm the changes or cancel them.

```
; updateFromBackup::pushButton
method pushButton(var eventInfo Event)
var
  custBakTC, custTC TCursor
endVar

custBakTC.open("CustBak.db")
custTC.open("Customer.db")

if custBakTC.locate("Customer No", 6312) then
  custTC.edit()
  custTC.insertRecord(custBakTC)
  if msgYesNoCancel("Confirm",
     "Do you want to overwrite Customer information?") = "Yes" then
    ; user confirmed, so overwrite the existing record
    custTC.updateRecord()
    custTC.endEdit()
  else
    ; user did not confirm, so cancel changes to record
    custTC.currRecord()
  endif
else
  msgStop("Error", "Can't find Customer 6312")
endif

endmethod
```

See also

- end, home, moveTo, nextRecord, priorRecord, skip,

cVar

TCursor

Method

Returns the variance of a field (column) in a table.

Syntax

1. **cVar** (const ***fieldName*** String) Number
2. **cVar** (const ***fieldNum*** SmallInt) Number

Description

Returns the population variance of values in a column of numeric fields. This method respects the limits of restricted views displayed in a linked table frame or multi-record object. **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

This method tries, for the duration of the retry period, to place a write lock on the table. If a lock cannot be placed, the method fails.

Example

In this example, the **pushButton** method for *thisButton* calculates the population variance deviation for two separate fields and displays the results in a dialog box:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  myTable TCursor
  test1, test2 Number
endVar
myTable.open("scores.dbf")
test1 = myTable.cVar("Test1")      ; get Test1 cVar
test2 = myTable.cVar(2)            ; assumes Test2 is field 2
msgInfo("Population Variance",
       "Test1 results : " + String(test1) + "\n" +
       "Test2 results : " + String(test2))
endmethod
```

See also

- cAverage, cCount, cMax, cMin, cNpv, cSamStd, cSamVar, cStd, cSum

deleteRecord

TCursor

Beginner

Method

Deletes the record pointed to by a TCursor.

Syntax

deleteRecord ( ) Logical

Description

Deletes the record pointed to by a TCursor without prompting for confirmation. The operation cannot be undone. The table must be in Edit mode.

If the record is locked or has already been deleted by another user (in a dBASE table), this method fails.

Example

In this example, the **pushButton** method for the *checkIOU* button determines whether a particular debt has been marked as paid; if it has, this code uses **deleteRecord** to delete the record:

```
; checkIOU::pushButton
method pushButton(var eventInfo Event)
var
  iou TCursor
```

```
  searchName String
endVar
searchName = "Hall"
iou.open("iou.db")
iou.edit()
if iou.locate("Name", searchName) then
  if iou."paid" = "Yes" then
    iou.deleteRecord()          ; delete the current record
    message(searchName + " deleted")
  else
    sendBill()                  ; run custom procedure
  endIf
else
   msgStop("Stop", "Couldn't find " + searchName)
endIf
endmethod
```

See also

- empty

didFlyAway

TCursor

Method

Reports whether the current record moved to a different position as the result of a key value change.

Syntax

didFlyAway ( ) Logical

Description

Returns True if the most recent call to **unlockRecord** caused the record to move to a different position in the table; otherwise, it returns False. This method is relevant only if the **setFlyAwayControl** method has been set to True (Yes); otherwise, **didFlyAway** returns False (even if the record moved to its sorted position).

Example

The following example demonstrates how **setFlyAwayControl** affects the position of a TCursor after a call to **unlockRecord** and under what circumstances **didFlyAway** returns True.

```
; demoButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

tc.open("MyTable.db")

; assume that MyTable.db has the following
; values in its only key field, "Customer No" :
;  Record#  Customer No
;     1        110
;     2        120   ; the code below changes this value to 145
;     3        130
;     4        140
                     ; which moves the record to this position
;     5        150
```

```
tc.setFlyAwayControl(Yes)
; now, a call to unlockRecord will make tc point
; to a record that "flies away"

if tc.locate("Key Field", 120) then
  tc.edit()

  ; change the key value so that the record
  ; changes relative position
  tc."Key Field" = 145

  ; Unlock the record. Because setFlyAwayControl
  ; is set to Yes, tc still points to the record
  tc.unlockRecord()

  ; this displays True because the new key value
  ; changes the record's relative position in the table
  msgInfo("Did 145 fly away?", tc.didFlyAway())

else
  message("120 not found.")
endif

endmethod
```

See also

- setFlyAwayControl, unLockRecord

dropIndex

TCursor

Method

Deletes an index file associated with a table.

Syntax

1. (Paradox tables) **dropIndex** ([const ***indexName*** String]) Logical
2. (dBASE tables) **dropIndex** (const ***indexName*** String [, const ***tagName*** String]) Logical

Description

Deletes a specified index file or index tag.

When working with a Paradox table, *indexName* is optional. If *indexName* is omitted, **dropIndex** deletes the table's primary index (unless the table has a secondary index, in which case the method fails).

When working with a dBASE table, you can use *indexName* to specify a .NDX file, or use *indexName* and *tagName* to specify a .MDX file and an index tag.

This method requires exclusive rights to the table if you're dropping a maintained index; otherwise, it requires a write lock.

Example

In this example, the **pushButton** method for *thisButton* deletes the *CustName* tag from a .MDX file and the primary index from a Paradox table:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc1, tc2 TCursor
  tbl Table
endVar

if isTable("Sales.dbf") then
  tbl.attach("Sales.dbf")                    ; Sales.dbf is a dBASE table
  tc1.open(tbl)
  tc1.dropIndex("index2.mdx", "CustName") ; delete CustName tag from index2 file
  msgInfo("", "custname dropped")
else
  msgStop("Stop!", "Could not find Sales.dbf table.")
endif

if isTable("Orders.db") then
  tc2.open("Orders.db")                      ; Orders.db is a Paradox table
  if tc2.dropIndex() then
    msgInfo("", "Primary index for Orders.db was deleted")
  else
    msgStop("", "Could not delete primary index for Orders.db")
  endif
else
  msgStop("Stop!", "Could not find Orders.db table.")
endif
endmethod
```

See also

- switchIndex

edit

TCursor

Beginner

Method

Puts a TCursor into Edit mode.

Syntax

edit ( ) Logical

Description

Puts a TCursor into Edit mode so changes can be made to the current record. After editing, if you want to stay in Edit mode, move off the record or use **postRecord** to accept changes to the record. If you want to leave Edit mode, use **cancelEdit** to cancel changes to the record or use **endEdit** to accept changes.

Example

The following example creates an array and uses **copyFromArray** to copy the contents of the array to a new record in the *CustName* table. Because the TCursor must be in Edit mode before the new record is inserted, this code uses **edit** to start editing the table. After the new

record is inserted, this code uses **endEdit** to end Edit mode and accept changes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("custname.db") then  ; open table
  tc.edit()                     ; put TCursor in Edit mode
  tc.insertRecord()             ; insert new record
  tc.copyFromArray(aa)          ; copy from array to table
  tc.endEdit()                  ; end Edit mode
else
  msgStop("Stop", "Couldn't open Custname.db.")
endIf
endmethod
```

See also

- cancelEdit, endEdit, postRecord

---

empty

TCursor

Method

Deletes all records from a table.

Syntax

empty ( ) Logical

Description

Deletes all records from a table without prompting for confirmation. The table does not have to be in Edit mode, but a write lock is required. This operation cannot be undone.

Example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table. If the user does not confirm the action, this code uses **nRecords** to indicate how many records exist in SCRATCH.DB.

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
  tblName String
  tc TCursor
endVar

tblName = "Scratch.db"
if isTable(tblName) then
  tc.open(tblName)
  if msgQuestion("Confirm", "Empty " + tblName + " table?") = "Yes" then
    tc.empty()
    message("All " + tblName + " records have been deleted.")
  else
    message(tblname + " has " + String(tc.nRecords()) + " records.")
```

TCursor

```
  endif
else
  msgInfo("Error", "Can't find " + tblName + " table.")
endif
endmethod
```

See also

- deleteRecord, nRecords

end

TCursor

Beginner

Method

Moves a TCursor to the last record in a table.

Syntax

end ( ) Logical

Description

Sets the current record (and the record buffer) to the last record in a table.

Example

This example uses **end** to move a TCursor to the last record in the *Orders* table, then displays in a dialog box information in the last record.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")         ; open tc for Orders table
tc.end()                     ; move to the last record in the table
                             ; display info in last record
msgInfo("Customer No " + tc."Customer No",
        "Oustanding balance: " + tc."Balance Due")

endmethod
```

See also

- currRec, home, moveToRecord, nextRec, priorRec, skip

endEdit

TCursor

Beginner

Method

Exits Edit mode and accepts changes made to the current record.

Syntax

endEdit ( ) Logical

Description

Accepts changes made to the current record and exits Edit mode. It does not close the TCursor. (Changes to previous records are committed or canceled as the user navigates through the table.)

Example

The following example creates an array and uses **copyFromArray** to copy the contents of the array to a new record in the *CustName* table. Because *CustName* must be in Edit mode before the new record is inserted, this code uses **edit** to start editing the table. After the new record is inserted, this code uses **endEdit** to exit Edit mode.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  aa Array[3] AnyType
endVar
aa[1] = "Borland"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("custname.db") then   ; open table
  tc.edit()                      ; put TCursor in Edit mode
  tc.insertRecord()              ; insert new record
  tc.copyFromArray(aa)           ; copy from array to table
  tc.endEdit()                   ; end Edit mode
else
  msgStop("Stop", "Couldn't open Custname.db.")
endIf
endmethod
```

See also

- cancelEdit, edit

enumFieldNames

TCursor

Method

Fills an array with the names of fields in a table.

Syntax

enumFieldNames (const ***fieldArray*** Array[] String **)** Logical

Description

Fills *fieldArray* with the names of the fields in a table. If *fieldArray* is resizeable, it grows automatically to hold the field names; if it is not resizeable, it holds as many as it can, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

Example

For this example, the **pushButton** method for the *enumFields* button stores field names in a resizeable array, then uses **view** to display the contents of the array.

```
; enumFields::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fieldNames Array[] String     ; field names for tables are always strings
endVar
if tc.open("orders.db") then
  tc.enumFieldNames(fieldNames) ; load fieldNames with names of Orders.db fields
  fieldNames.view()             ; display field names in a dialog box
```

```
else
  msgStop("Stop", "Couldn't open Orders.db.")
endIf

endmethod
```

See also

- enumFieldNamesInIndex

enumFieldNamesInIndex

TCursor

Method

Fills an array with the names of fields in a table's index.

Syntax

1. (Paradox tables) **enumFieldNamesInIndex**
   ([const ***indexName*** String,] ***fieldArray*** Array[] String) Logical
2. (dBASE tables) **enumFieldNamesInIndex**
   ([const ***indexName*** String [, const ***tagName*** String ,] ***fieldArray*** Array[] String) Logical

Description

Fills *fieldArray* with the names of the fields in a table's index, as specified in *indexName*. If *indexName* is omitted, this method uses the current index. If *fieldArray* is resizeable, it grows automatically to hold the field names; if it is not resizeable, it holds as many as it can, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

When working with a dBASE table, you can use the optional argument *tagName* to specify an index tag within a .MDX file.

Example

In this example, the **pushButton** method for the *enumIndex* button stores field names in a resizeable array, then uses **view** to display the contents of the array:

```
; enumIndex::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fieldNames Array[] String
endVar
if tc.open("Sales.dbf") then
  ; load fieldNames array with field names in the byDate index
  tc.enumFieldNamesInIndex("DateIndx", "byDate", fieldNames)
  ; display the index field names for byDate in DateIndx
  fieldNames.view()
else
  msgStop("Stop", "Couldn't open Sales.dbf.")
endIf
endmethod
```

See also

- enumIndexStruct

enumFieldStruct

TCursor

Method

Creates a Paradox table listing the structure of a TCursor.

Syntax

enumFieldStruct (const ***tableName*** String) Logical

Description

Creates the Paradox table specified in *tableName* listing the structure of a TCursor. If *tableName* exists, this method overwrites it without confirmation.

You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **struct** option in a **create** statement to borrow a table's field structure (including primary keys and validity checks) for use in the new table.

The structure of the Paradox table is listed in the following table:

Field Name	Type	Size
FieldName	A	31
Type	A	31
Size	S	
Dec	S	
Key	A	1
_Required Value	A	1
_Min Value	A	255
_Max Value	A	255
_Default Value	A	255
_Picture Value	A	175
_Table Lookup	A	81
_Table Lookup Type	A	1
_Invariant Field ID	S	

Example

For this example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you want all of the fields in *NewCust* to be required fields. To accomplish this, the following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans through *CustFlds* and modifies the field definitions so that each record describes a field that will be required. *CustFlds* is then supplied in the **struct** clause of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
```


TCursor

```
var
  newCustTbl Table
  tc   TCursor
  structName, sourceName String
endVar

structName = "CustFlds.db"
sourceName = "Customer.db"

if tc.open(sourceName) then

  ; include from Customer field name, ValChecks,
  ; and key fields in new table CustFlds
  tc.enumFieldStruct(structName)

  ; now point the TCursor to the CustFlds table
  tc.open(structName)
  tc.edit()

  ; this loop scans through the CustFlds table and
  ; changes ValCheck definitions for every field
  scan tc :
    tc."_Required Value" = 1    ; make all fields required
  endscan

  ; now create NEWCUST.DB and borrow field names,
  ; ValChecks and key fields from CUSTFLDS.DB
  newCustTbl = create "NewCust.db"
                 struct structName
               endCreate

  ; NEWCUST.DB requires that all fields be filled

else
  msgStop("Error", "Can't get field structure for Customer table.")
endif

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumIndexStruct, enumRefIntStruct, enumSecStruct
- The create keyword listed in the Table type

enumIndexStruct

TCursor

Method

Creates a Paradox table listing the structure of a TCursor's secondary indexes.

Syntax

enumIndexStruct (const ***tableName*** String) Logical

Description

Creates the Paradox table specified in *tableName* listing the structure of a table's secondary indexes. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes**

method. If *tableName* exists, this method prompts the user for confirmation before overwriting the table.

You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow secondary indexes for use in the new table.

The structure of *tableName* is listed in the following table.

Field Name	Field Type
infoHeader	A1
szName	A127
szTagName	A31
szFormat	A31
bPrimary	A1
bUnique	A1
bDescending	A1
bMaintained	A1
bCaseInsensitive	A1
bSubset	A1
bExpIdx	A1
bKeyExpType	N
szKeyExp	A220
szKeyCond	A220
FieldNo	N
FieldName	A31

Example

For this example, assume that you want a new table called *NewCust* that is similar to the *Customer* table. However, you don't want to borrow referential integrity or security information. To accomplish this, the following code uses **enumFieldStruct** and **enumIndexStruct** to generate two tables: CUSTFLDS.DB and CUSTINDX.DB. *CustFlds* and *CustIndx* are then supplied to the **struct** and **indexStruct** clauses of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
  newcustTC Table
  custTC    TCursor
endVar

if custTC.open("Customer.db") then

  ; write field level information to CUSTFLDS.DB
  custTC.enumFieldStruct("CustFlds.db")

  ; write secondary index information to CUSTINDX.DB
```

TCursor

```
  custTC.enumIndexStruct("CustIndx.db")

  ; now create NEWCUST.DB--
  ; borrow field names, ValChecks, and key fields from CUSTFLDS.DB
  ; borrow secondary indexes from CUSTINDX.DB
  newcustTC = create "NewCust.db"
                struct "CustFlds.db"
                indexStruct "CustIndx.db"
              endCreate

else
  msgStop("Error", "Can't find Customer table.")
endif

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct enumRefIntStruct, enumSecStruct
- The create keyword in the Table type

enumLocks

TCursor

Method

Creates a Paradox table listing the locks currently applied to a TCursor.

Syntax

enumLocks (const ***tableName*** String) LongInt

Description

Creates the Paradox table specified in *tableName*. *tableName* lists the locks currently applied to the table pointed to by a TCursor. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. For dBASE tables, this method lists only the lock you've placed (not all locks currently on the table). You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is listed below:

Field Name	Type	Size
UserName	A	15
Lock Type	A	32
Net Session	N	
Session	N	
Record Number	N	

Example

In this example, the built-in **pushButton** method for the *showOrdersLcks* button creates a table listing the locks currently applied to ORDERS.DB and opens the newly created table.

```
; showOrdersLcks::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tv TableView
endVar
if tc.open("Orders.db") then
  tc.enumLocks("OrderLck.db")  ; store Orders.db locks in OrderLck.db
  tv.open("OrderLck.db")       ; open OrderLck.db
else
  msgStop("Stop!", "Can't open Orders.db table")
endif

endmethod
```

See also

- lock, lockStatus

enumRefIntStruct

TCursor

Method

Creates a Paradox table listing referential integrity information for a TCursor.

Syntax

enumRefIntStruct (const ***tableName*** String) Logical

Description

Writes referential integrity information for a TCursor to the table specified in *tableName*. If *tableName* exists, this method prompts the user for confirmation before overwriting the table. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow referential integrity information for use in the new table.

The structure of *tableName* is listed in the following table:

TCursor

Field Name	Type	Size
infoHeader	A	1
RefName	A	31
Other Table	A	81
Slave	A	1
Modify	A	1
Delete	A	1
FieldNo	N	
aiThisTabField	A	31
Other FieldNo	N	
aiOthTabField	A	31

Example

This example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. Then, the code supplies *CustRef* to the **refIntStruct** clause in a **create** statement.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc  TCursor
  tbl Table
endVar

tc.open("Customer.db")

; write referential integrity information to CustRef
tc.enumRefIntStruct("CustRef.db")

; write field level information to CustFlds
tc.enumFieldStruct("CustFlds.db")

; now create NEWCUST.DB--
; borrow field level information from CUSTFLDS.DB
; borrow referential integrity information from CUSTREF.DB
tbl = create "NewCust.db"
        struct "CustFlds.db"
        refIntStruct "CustRef.db"
      endCreate

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct, enumIndexStruct, enumSecStruct
- enumTableLinks in the Form type

---

enumSecStruct

TCursor

Method

Writes table security information to a Paradox table.

Syntax

enumSecStruct (const ***tableName*** String)

Description

Creates the Paradox table specified in *tableName* listing security information (access rights) of a table. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

You can supply *tableName* to the **secStruct** option in a **create** statement to borrow security information for use in the new table.

The structure of *tableName* is listed in the following table.

Field Name	Type	Size
TableName	A	32
PropertyName	A	64
PropertyValue	A	255

Example

This example creates a new table based on the security information associated with the *Secrets* table. The code uses **enumSecStruct** to write security information to the *SecInfo* table, then uses the table to create the *MySecrts* table.

```
; getSecrets::pushButton
method pushButton(var eventInfo Event)
var
  tc  TCursor
  tbl Table
endVar

tc.open("Secrets.db")

; write security information to SECRETS.DB
tc.enumSecStruct("Secrets.db")

; now create MYSECRTS.DB--
; borrow field names and types from SECRETS.DB
; borrow security information from MYSECRTS.DB
tbl = create "MySecrts.db"
        like "Secrets.db"
        secStruct "Secrets.db"
      endCreate

endmethod
```

See also

- enumFieldNames, enumFieldNamesInIndex, enumFieldStruct, enumIndexStruct, enumRefIntStruct

enumTableProperties

TCursor

Method

Writes the properties of a TCursor to a Paradox table.

Syntax

enumTableProperties (const ***tableName*** String) Logical

Description

Writes the properties of a table associated with a TCursor to the table specified in *tableName*. If *tableName* exists, this method prompts the user for confirmation before overwriting the table. If *tableName* is open, this method fails. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory.

The structure of *tableName* is listed in the following table.

Field Name	Type	Size
TableName	A	32
PropertyName	A	64
PropertyValue	A	255

Example

The following example uses **enumTableProperties** to write ORDERS.DB properties to ORDPROPS.DB. If ORDPROPS.DB exists, this code prompts the user for confirmation before overwriting the table.

```
; showTblProps::pushButton
method pushButton(var eventInfo Event)
var
  tblName, propTbl String
  tc TCursor
  tv TableView
endVar
tblName = "Orders.db"
propTbl = "OrdProps.db"

if tc.open(tblName) then
  if isTable(propTbl) then
    if msgYesNoCancel("Confirm",
       propTbl + " exists. Overwrite it?") <> "Yes" then
      return
    endif
  endif
  ; write Orders.db properties to OrdProps.db
  tc.enumTableProperties(propTbl)
  ; open newly created OrdProps.db table
  tv.open(propTbl)
else
  msgStop("Stop!", "Can't open " + tblName + " table.")
endif

endmethod
```

See also

- enumFieldNames

eot

TCursor

Method

Tests for a move past the end of a table.

Syntax

eot ( ) Logical

Description

Returns True if a command attempts to move past the last record of a table; otherwise, it returns False. **eot** is reset by the next move operation.

eot (and **bot**) also return True if a command forces the TCursor to point to a nonexistent record. For example, suppose the *Customer* table has values in the first key field that range from 1000 to 10,000. If you call **setFilter** such that the TCursor points to key values from 1 to 10 (outside the possible range of *Customer* values), the TCursor points to a nonexistent record. The following code fragment demonstrates **setFilter** can affect **eot** and **bot**:

```
var tc TCursor endvar
tc.open("Customer.db")
tc.setFilter(1, 10)          ; filter ranges from 1 to 10
  ; tc.eot() and tc.bot() are True at this point
```

Similarly, if a call to **switchIndex** forces the TCursor to point to a nonexistent record, **eot** and **bot** methods return True.

Example

In this example, a **while** loop controls a TCursor's movement through the *Orders* table. When code within the loop attempts to move beyond the end of the table, **eot** returns True and the loop terminates.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
  fldVal AnyType
endVar
tblName = "Customer.db"
if tc.open(tblName) then
  while tc.eot() = False      ; while subsequent commands do not
                              ; move past end of the table
    message(tc."Customer No") ; display value in Customer No field
    sleep(250)                ; pause for the message
    tc.nextRecord()           ; move to the next record
  endWhile
  msgInfo("End", "That's all, folks!")
else
  msgStop("Stop!", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- bot, end, home

TCursor

familyRights

TCursor

Method

Tests for a user's ability to create or modify objects in a table's family.

Syntax

familyRights (const ***rights*** String) Logical

Description

Returns True if you have rights to the type of object specified in *rights;* otherwise, it returns False. *rights* is a single-letter string—either "F" (form), "R" (report), "S" (image settings), or "V" (validity checks)— that indicates the type of object you are interested in.

Example

This example indicates in a dialog box whether you have "F" rights to CUSTOMER.DB.

```
; showFRights::pushButton
method pushButton(var eventInfo Event)
var
  custTC TCursor
endVar

custTC.open("Customer.db")
msgInfo("Rights", "Form Rights: " + String(custTC.familyRights("F")))
; displays True if you have Form rights to Customer.db

endmethod
```

See also

- tableRights

fieldName

TCursor

Method

Returns the name of a field.

Syntax

fieldName (const ***fieldNum*** SmallInt) String

Description

Returns the name of field *fieldNum*. Fields are numbered from left to right, beginning with 1.

Example

The following example uses **fieldName** to display the name of field number two in the *Answer* table. This code is attached to the built-in **pushButton** method of a button.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldName, tblName String
  fldNum SmallInt
endVar
```

```
tblName = "Answer.db"

if tc.open(tblName) then
  fldName = tc.fieldName(2)       ; store name of field 2 in fldName
  msgInfo("Field Name",           ; display field 2 field name
          "Field name for field 2 is\n" + fldName)
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endIf

endmethod
```

See also

- fieldNo, fieldType, fieldValue

fieldNo

TCursor

Method

Returns the position of a field in a table.

Syntax

fieldNo (const ***fieldName*** String) SmallInt

Description

Returns the position of the field *fieldName* in a table. Fields are numbered from left to right, beginning with 1.

Example

The following code is attached to the **pushButton** method for *thisButton*. When you press *thisButton*, this example uses **fieldName** to display *Common Name*'s field position in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldNum SmallInt
endVar

if tc.open("biolife.db") then
  fldNum = tc.fieldNo("Common Name")   ; store field number in fldNum
  msgInfo("Field Number",
          "Common Name field is\n field number " + String(fldNum))
else
  msgInfo("Sorry", "Can't open BioLife.db table.")
endif

endmethod
```

See also

- fieldValue

fieldRights

TCursor

Method

Reports whether a user has rights to read or modify a field in a table.

TCursor

Syntax

1. **fieldRights** (const ***fieldName*** String, const ***rights*** String) Logical
2. **fieldRights** (const ***fieldNum*** SmallInt, const ***rights*** String) Logical

Description

Returns True if the user has *rights* to the field specified in *fieldName* or *fieldNum*; otherwise, it returns False. The value of *rights* must be an expression that evaluates to one of the following strings: ReadOnly, ReadWrite, or All. Rights are obtained using the Session type method **addPassword**; rights cannot be acquired after the table is opened.

Example

This example uses **fieldRights** to determine whether a TCursor has adequate field rights before attempting to modify the field's value.

```
; updateCust::pushButton
method pushButton(var eventInfo Event)
var
  custTC TCursor
endVar
custTC.open("Customer.db")
if custTC.locate("Name", "Unisco") then
  ; if we don't have sufficient rights to change the Name field
  if NOT custTC.fieldRights("Name", "ReadWrite") then
    ; display error message and abort operation
    msgStop("Error!", "Insufficient rights to change Name field")
  else
    ; otherwise, we have rights to make changes to the field
    custTC.edit()
    custTC.Name = "Unisco Worldwide, Inc."
    message("Changed Unisco to Unisco Worldwide, Inc.")
    custTC.endEdit()
  endif
else
  msgStop("Error", "Can't find Unisco")
endif

endmethod
```

See also

- tableRights

fieldSize

TCursor

Method

Returns the size of a field.

Syntax

1. **fieldSize** (const ***fieldName*** String) SmallInt
2. **fieldSize** (const ***fieldNum*** SmallInt) SmallInt

Description

Returns the size of a field as defined when the table was created. The return value can represent the maximum number of characters a field can contain; for example, given a field defined as Alpha20, **fieldSize** returns 20. Or, the return value can represent the maximum amount

of data to display. For example, when you create a table and define a Memo field, you can specify a number of characters to display. **fieldSize** would return that number.

Numeric fields in dBASE tables can specify the number of digits to display on either side of the decimal point; for example, a field defined as Number 8.2 could display up to 8 digits total, with 6 digits to the left of the decimal and 2 digits to the right. **fieldSize** returns the first part of the definition; that is, the number of digits to the left of the decimal. To get the second part, use **fieldUnits2**.

For field types that do not display characters or numbers (such as OLE, binary, graphic, and so on), this method returns 0.

Example

This example uses a dynamic array to store the size of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldSizes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldSizes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if tc.open(tblName) then
   ; this FOR loop loads the DynArray with BioLife.db field sizes
  for i from 1 to tc.nFields()
    fldSizes[tc.fieldName(i)] = tc.fieldSize(i)
  endFor
   ; now show the contents of the DynArray
  fldSizes.view(tblName + " field sizes.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- fieldName, fieldType, fieldUnits2

fieldType

TCursor

Method

Returns the data type of a field.

Syntax

1. **fieldType** (const ***fieldName*** String) String
2. **fieldType** (const ***fieldNum*** SmallInt) String

Description

Returns the data type of a field. It returns "Unknown" if the field is not found. The following table lists the possible return values:

Field type	Paradox table	dBASE table
Alphanumeric	ALPHA	(none)
Character	(none)	CHARACTER
Date	DATE	DATE
Integer	SHORT	(none)
Floating-point value	NUMERIC	FLOAT (IV)
		NUMERIC (III+ or IV)
Graphic	GRAPHIC	(none)
Logical	(none)	BOOLEAN
Money	MONEY	(none)
Memo	MEMO	MEMO
Formatted memo	FMTMEMO	(none)
Binary	BINARYBLOB	(none)
OLE object	OLEOBJ	(none)

Example

This example uses a dynamic array to store the type of each field in the *BioLife* table, then displays the contents of the dynamic array in a dialog box.

```
; showFldTypes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldTypes DynArray[] AnyType
  tblName String
endVar
tblName = "BioLife.db"

if tc.open(tblName) then
   ; this FOR loop loads the DynArray with BioLife.db field types
  for i from 1 to tc.nFields()
    fldTypes[tc.fieldName(i)] = tc.fieldtype(i)
  endFor
   ; now show the contents of the DynArray
  fldTypes.view(tblName + " field types.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- fieldNo

fieldUnits2

TCursor

Method

Returns the number of decimal places defined for a numeric field in a dBASE table.

Syntax

1. fieldUnits2 (const ***fieldName*** String) SmallInt
2. fieldUnits2 (const ***fieldNum*** SmallInt) SmallInt

Description

Returns the number of decimal places defined for a numeric field in a dBASE table. Numeric fields in dBASE tables can specify the number of digits to display on either side of the decimal point; for example, a field defined as Number 8.2 could display up to 8 digits total with 6 characters to the left of the decimal and 2 digits to the right. **fieldUnits2** returns the second part of the definition; that is, the number of digits to the right of the decimal. To get the first part, use **fieldSize**. This method returns 0 for non-numeric field types such as alphanumeric, boolean, date, and so on.

Example

For this example, the **pushButton** method for *thisButton* concatenates values returned from **fieldSize** and **fieldUnits2** such that both sides of the decimal point are expressed in a single number. For example, if a field's size is 11 and is defined with 2 decimal places, this method concatenates the values to 11.2. This code uses a DynArray to store concatenated values for each field in SCORES.DBF then displays the contents of the array in a dialog box.

```
; showFldSizes::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  i SmallInt
  fldSizes DynArray[] AnyType
  tblName String
  totalSize Number
endVar
tblName = "Scores.dbf"

if tc.open(tblName) then
   ; This FOR loop loads the DynArray with the full field spec.
   ; For example if fieldSize(1) = 11 and fieldUnits2(1) = 2,
   ; one value in the DynArray would be 11.2
  for i from 1 to tc.nFields()
    totalSize = numVal(String(tc.fieldsize(i)) + "." +
                       String(tc.fieldUnits2(i)))
    fldSizes[tc.fieldName(i)] = totalSize
  endFor
   ; now show the contents of the DynArray
  fldSizes.view(tblName + " total field sizes.")
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

❒ fieldName, fieldNo, fieldSize, fieldType

fieldValue

TCursor

Method

Reads the value of a specified field.

Syntax

1. **fieldValue** (const ***fieldName*** String, var ***result*** AnyType) Logical
2. **fieldValue** (const ***fieldNum*** SmallInt, var ***result*** AnyType) Logical

Description

Gets the value of a specified field (*fieldName* or *fieldNum*) in the current record and assigns it to the variable *result*. This method returns True if successful; otherwise, it returns False.

You can get the same information using dot notation. For example, this statement uses dot notation to assign the *myPrice* variable with data from the Last Bid field:

```
myCost = tcVar."Last Bid"
```

The following statement uses **fieldValue** to achieve the same results:

```
tcVar.fieldValue("Last Bid", myCost)
```

Example

For this example, assume a form has at least one field, one of which is named *paymentField*. When you right-click on the *paymentField*, the code in this example presents a pop-up menu listing possible values for the field. When you choose a menu item from the list, that value is inserted into the field.

The following code is attached to the field's Var window:

```
; paymentField::Var
Var
  lkupTbl String
  menuArray Array[] String
  fldVal AnyType
  pl PopUpMenu
  tc TCursor
endVar
```

The following code is attached to the field's **open** method. When the field opens, this code scans through the *PayMethd* table and loads the *menuArray* array with values from the *Pay Method* field.

```
; paymentField::open
method open(var eventInfo Event)

lkupTbl = "PayMethd.db"
tc.open(lkupTbl)
scan tc :                             ; scan through table
  tc.fieldValue("Pay Method", fldVal) ; store field value in fldVal
  menuArray.addLast(fldVal)           ; add new element to menuArray
endScan
pl.addStaticText("Possible Values")   ; put static text at top of menu
pl.addSeparator()                     ; add a vertical bar below static text
```

```
pl.addArray(menuArray)              ; add array to the menu

endmethod
```

The following code is attached to the field's **mouseRightUp** method. When you right-click the field, this code presents a pop-up menu and the field takes the value of your menu choice.

```
; paymentField::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)

disableDefault              ; don't show the default menu
choice = pl.show()          ; show the pop-up menu
if NOT isBlank(choice) then ; if user did not press Esc
  self.value = choice       ; enter choice into the field
endif

endmethod
```

See also

- setFieldValue

getLanguageDriver

TCursor

Method

Returns the name of the current language driver for a table.

Syntax

getLanguageDriver ( ) String

Description

Returns a String value indicating the name of the current language driver for a table.

Example

This example displays in a dialog box the language driver for the *Customer* table:

```
; getDriver::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")
msgInfo("", tc.getLanguageDriver())  ; displays "ascii"
endmethod
```

See also

- getLanguageDriverDesc

getLanguageDriverDesc

TCursor

Method

Returns name of the current language driver description for a table.

Syntax **getLanguageDriverDesc ( )** String

Description Returns a String value indicating the language driver description for a table.

Example This example displays in a dialog box the language driver description for the *Customer* table.

```
; getDriverDesc::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")
msgInfo("", tc.getLanguageDriverDesc())  ; displays "Paradox ascii"
endmethod
```

See also

- getLanguageDriver

home

Beginner

TCursor

Method Moves to the first record of a table.

Syntax **home ( )** Logical

Description Sets the current record (and the record buffer) to the first record in a table.

Example For this example, the **pushButton** method associates a TCursor with the *Orders* table, then loads an array with field values in a **scan** loop. After the loop terminates, the TCursor is positioned at the last record in the table. This method uses **home** to move the TCursor back to the first record of the table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  fldArray Array[] AnyType
  fldVal AnyType
endVar
tc.open("Orders.db")
fldArray.grow(tc.nRecords())
; scan table and store order numbers in fldArray
scan tc:
  tc.fieldValue(1, fldVal)
  fldArray[tc.recNo()] = tc."Order No"
endScan
; TCursor is on the last record after the scan loop

fldArray.view()          ; display contents of array
```

```
tc.home()                  ; move TCursor to the first record
endmethod
```

See also

❒ end, moveToRecord, moveToRecNo, nextRecord, priorRecord, skip

initRecord

TCursor

Method

Empties the record buffer.

Syntax

initRecord ( ) Logical

Description

Initializes the record buffer by filling it with blanks (*not* spaces). If default values have been set for fields in the table, **initRecord** initializes those fields with the default.

See also

❒ copyRecord, currRecord

insertAfterRecord

TCursor

Method

Inserts a record into a table after the current record.

Syntax

insertAfterRecord ([const ***pointer*** TCursor]) Logical

Description

Inserts a record after the current record. This method is useful for inserting a new record after the last record of a table. You can use the optional argument *pointer* to insert the record pointed to by a different TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise it is inserted after the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

Example

For this example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in **action** method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertAfterRecord** you don't have to store the

record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by a TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a record
  if thisForm.Editing = True then        ; if form is in Edit mode
    disableDefault                        ; don't process DataDeleteRecord yet

    if msgYesNoCancel("Confirm",          ; if user confirms delete
      "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)             ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.end()                       ; move to end of table
        tcArc.insertAfterRecord(tcCust)   ; insert current CUSTOMER record
                                          ; after last record in CustArc.db
        doDefault                         ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can't archive record.")
      endif
    else                                  ; else user didn't confirm delete
      message("Record not deleted.")
    endif
  else                                    ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endif
endif
endmethod
```

See also

❒ insertBeforeRecord, insertRecord

insertBeforeRecord

TCursor

Method

Inserts a record into a table before the current record.

Syntax

insertBeforeRecord ([const ***pointer*** TCursor]) Logical

Description

Inserts a record before the current record (the same as **insertRecord**). You can use the optional argument *pointer* to insert the record pointed to by another TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise, it is inserted before the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

Example

For this example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in **action** method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertBeforeRecord** you don't have to store the record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by a second TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't process DataDeleteRecord yet

    if msgYesNoCancel("Confirm",          ; if user confirms delete
       "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)             ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.insertBeforeRecord(tcCust)  ; insert current CUSTOMER record
                                          ; before current record in CustArc.db
        doDefault                         ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can't archive record.")
      endif
    else                                  ; else user didn't confirm delete
      message("Record not deleted.")
    endif
  else                                    ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endif
endif
endmethod
```

See also

- insertAfterRecord, insertRecord

insertRecord

TCursor

Beginner

Method

Inserts a record into a table.

Syntax

insertRecord ([const ***pointer*** TCursor]) Logical

Description

Inserts a record into a table before the current record (the same as **insertBeforeRecord**). You can use the optional argument *pointer* to insert the record pointed to by another TCursor, or omit the argument to insert a blank record.

If the table is indexed, the record is placed in its sorted position when the record is committed; otherwise, it is inserted before the current record.

This method fails if the table is not in Edit mode. Also, this method fails if the current record cannot be committed (for example, because of a key violation).

Example

For this example, assume a form has a table frame, *CUSTOMER*, bound to CUSTOMER.DB. When the user attempts to delete a record, the built-in **action** method for *CUSTOMER* moves the record to CUSTARC.DB before deleting the record from *CUSTOMER*. You could use **copyFromArray** and **copyToArray** to accomplish the same thing, but if you use **insertRecord** you don't have to store the record in an array in order to copy it. As this code demonstrates, you can use the optional argument *pointer* to insert the record pointed to by another TCursor.

```
; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
  tcCust, tcArc TCursor
endVar
if eventInfo.id() = DataDeleteRecord then ; if user attempts to delete a record
  if thisForm.Editing = True then         ; if form is in Edit mode
    disableDefault                        ; don't process DataDeleteRecord yet

    if msgYesNoCancel("Confirm",          ; if user confirms delete
       "Delete the current record?") = "Yes" then
      tcCust.attach(CUSTOMER)             ; sync TCursor to CUSTOMER pointer
      if tcArc.open("CustArc.db") then
        tcArc.edit()
        tcArc.insertRecord(tcCust)        ; insert current CUSTOMER record
                                          ; before current record in CustArc.db
        doDefault                         ; process DataDeleteRecord now
      else
        msgStop("Stop!", "Sorry, Can't archive record.")
      endif
    else                                  ; else user didn't confirm delete
      message("Record not deleted.")
    endif
  else                                    ; else form is not in Edit mode
    msgStop("Stop!", "Press F9 to edit data.")
  endif
endif
endmethod
```

See also

- insertAfterRecord, insertBeforeRecord

isAssigned

TCursor

Method

Reports whether a TCursor variable has been assigned a value.

Syntax

isAssigned ( ) Logical

Description

Returns True if a TCursor variable has a value assigned using **open** or **attach**; otherwise, it returns False.

Example

This example associates a TCursor with a table, displays information found in the last record, then closes the TCursor. In this example, the code displays a message indicating whether the TCursor variable is still assigned after the TCursor is closed. This code is attached to the built-in **pushButton** method for *thisButton*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")          ; open a TCursor for Orders.db
tc.end()                      ; move to end of the table

; display information in last record
msgInfo("Last Order", "Order number: " +
String(tc."Order No") + " \nOrder date: " + String(tc."Sale Date"))

tc.close()                    ; attempt to close TCursor

; if close is successful, this displays False (tc is no longer assigned)
; otherwise, it displays True (tc is still assigned if close fails)
msgInfo("Is tc Assigned?", tc.isAssigned())

endmethod
```

See also

- close, open

TCursor

isEdit

TCursor

Method

Reports whether a TCursor is in Edit mode.

Syntax

isEdit ( ) Logical

Description

Returns True if the TCursor is in Edit mode; otherwise, it returns False. If you attach a TCursor to a display manager (such as a UIObject, or a TableView), and that object is in Edit mode, the TCursor will be in Edit mode as well.

Example

For this example, assume a form has a table frame bound to the *Customer* table and a button. The code attached to the **pushButton** method for *thisButton* attaches a TCursor to the table frame, then uses **isEdit** to determine whether the TCursor is in Edit mode. If the table

frame was in Edit mode when the TCursor was attached, the TCursor will also be in Edit mode.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

; attach to the table frame
tc.attach(CUSTOMER)

; if CUSTOMER was in Edit mode, tc will be in Edit mode too

if NOT tc.isEdit() then    ; test whether tc is in Edit mode
  tc.edit()
endif

if tc.locate("Name", "Action Club") then
  tc.phone = "808-555-1234"
else
  msgStop("Sorry", "Can't find Action club")
endif

endmethod
```

See also

- attach

isEmpty

TCursor

Method

Determines whether a table contains any records.

Syntax

isEmpty ( ) Logical

Description

Returns True if there are no records in the table associated with the TCursor; otherwise, it returns False.

Example

For this example, the **pushButton** method for the *rptRecNo* button displays the number of records in the *Orders* table. If the table is empty, this code alerts the user that the table is empty.

```
; rptRecNo::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
endVar
tblName = "Orders.db"

if tc.open(tblName) then
  if tc.isEmpty() then                        ; if Orders.db is empty
    msgStop("Hey!",
            tblName + " table is empty!")
```

```
    else
      msgInfo(tblName + " table has",            ; report number of records
              String(tc.nRecords()) + " records")
    endIf
  else
    msgStop("Sorry", "Can't open " + tblName + " table.")
  endif
  endmethod
```

See also

- empty, insertRecord, nRecords

isEncrypted

TCursor

Method

Reports whether a table is password-protected.

Syntax

isEncrypted ( ) Logical

Description

Returns True if a table is password-protected; otherwise, it returns False. A TCursor can't be opened on an encrypted table until you use the Session type method **addPassword** to present the required password. This method does not report whether a user has access rights to the table—use **tableRights** for that.

Example

This example tests whether the *Customer* table is encrypted.

```
; thisButton::pushButton
method open(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Customer.db") then
  if tc.isEncrypted() then
    msgInfo("Table is protected", "An acceptable password has been presented.")
else
  msgStop("Error", "Can't open the Customer table.")
endif

endmethod
```

See also

- tableRights
- addPassword in the Session type
- protect in the Table type

isRecordDeleted

TCursor

Method

Reports whether the current record has been deleted (dBASE tables only).

Syntax

isRecordDeleted ( ) Logical

Description

Reports whether the current record has been deleted. **isRecordDeleted** works only for dBASE tables because deleted Paradox records can't be displayed. This method returns True if the current record has been deleted; otherwise, it returns False.

Deleted records in a dBASE table are not shown by default. For **isRecordDeleted** to work correctly, you must call **showDeleted** to show deleted records in the table; otherwise, deleted records are not visible to **isRecordDeleted**.

Example

This example opens a TCursor for the SCORES.DBF dBASE table, then uses **showDeleted** to display all deleted records. Then, the code attempts to locate a specific record in the table. This example uses **isRecordDeleted** to determine whether the record has been deleted; if it has, it is undeleted with **undeleteRecord**. The following code is attached to the **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Scores.dbf")                  ; open TCursor on a dBASE table
tc.showDeleted()                       ; show deleted records
if tc.locate("Name", "Jones") then     ; if locate finds Jones in Name field
  if tc.isRecordDeleted() then         ; if the record has been deleted
    tc.edit()                          ; begin Edit mode
    tc.undeleteRecord()                ; undelete the record
    message("Jones record undeleted")
  endif
else
  msgStop("Error", "Can't find Jones.")
endif
endmethod
```

See also

- isShowDeletedOn, showDeleted

isShared

TCursor

Method

Reports whether a table is currently shared.

Syntax

isShared ( ) Logical

Description

Returns True if another user has opened the table pointed to by a TCursor; otherwise, it returns False.

Example

For this example, a form's built-in **open** method determines whether CUSTOMER.DB is currently being shared by another user; if it is, the user is warned and given the option to continue or abort.

```
; thisPage::open
method open(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Customer.db")              ; open a TCursor for Customer
if tc.isShared() then               ; if table is currently shared
  if msgYesNoCancel("Continue?",    ; ask for confirmation
     "Customer table is currently being shared.\n" +
     "Continue anyway?") <> "Yes" then

    close()                         ; close this form
  endif
endif
endmethod
```

See also

- isAssigned, isValid

isShowDeletedOn

TCursor

Method

Reports whether deleted records in a dBASE table are shown.

Syntax

isShowDeletedOn ( ) Logical

Description

Reports whether the table pointed to by a TCursor currently shows deleted records. You can use the **showDeleted** method to specify whether or not to show deleted records, then use **isShowDeletedOn** to determine states. **isShowDeletedOn** is valid only for dBASE tables.

Example

In this example, if **isShowDeletedOn** returns False, the code calls **showDeleted** to show deleted records in ORDERS.DBF.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  dbfTC TCursor
endVar
if dbfTC.open("Orders.dbf") then
  if NOT dbfTC.isShowDeletedOn() then   ; if deleted records are not shown
    dbfTC.showDeleted(Yes)              ; show deleted records
  endif
else
```

```
  msgStop("Sorry", "Can't open Orders.dbf table.")
endIf
endmethod
```

See also

- compact, isRecordDeleted, showDeleted

isValid

TCursor

Method

Reports whether the contents of a field are legitimate and complete.

Syntax

1. **isValid** (const ***fieldName*** String, const ***value*** AnyType) Logical
2. **isValid** (const ***fieldNum*** SmallInt, const ***value*** AnyType) Logical

Description

Reports whether the value specified in *value* conforms with field type and validity checks for the field specified in *fieldNum* or *fieldName.* This method gives you an opportunity to check whether a new value is valid for a field before you attempt to post the record.

isValid returns True if *value* conforms to field type and validity checks; otherwise, it returns False.

Example

This example uses **isValid** to test whether a given value is valid for a Date field. If the value is not valid, this code warns the user of the error; otherwise the value is entered into the field. The following code is attached to the **pushButton** method for *thisButton.*

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tryValue Number
endVar
tryValue = 100
tc.open("Orders.db")
if NOT tc.isValid("Sale Date", tryValue) then  ; 100 is not a valid date
  msgStop("Error",
  String(tryValue) + " is not valid for this field.")
else                                           ; this condition is never met
  tc."Sale Date" = tryValue
  tc.postRecord()
endif

endmethod
```

See also

- postRecord, setFieldValue

locate

TCursor

Beginner

Method

Searches for a specified field value.

Syntax

1. **locate** (const ***fieldName*** String,
   const ***exactMatch*** AnyType [, const ***fieldName*** String,
   const ***exactMatch*** AnyType]*) Logical
2. **locate** (const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType [, const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType]*) Logical

Description

Searches a table for records whose values match the criteria specified in one or more field/value pairs. Specify the value to search for in *searchValue* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the first value matching *searchValue* is found, given the current view of the records. If the TCursor is using a secondary index, **locate** finds the first record in secondary index order—regardless of that record's primary index order.

The search always starts from the beginning of the table, but if no match is found, the TCursor returns to the original record. If a match is found, the TCursor moves to that record. This operation fails if the current record cannot be posted (for example, because of a key violation).

Example

In the following example, the **pushButton** method for the *fixSpelling* button searches for a value in the *Name* field of the *Customer* table. If **locate** is successful, this method replaces the name with a new value and informs the user of the change.

```
; fixSpelling::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
endVar

ordTC.open("Customer.db")
; if locate finds "Profesional Divers, Ltd." in the Name field
if ordTC.locate("Name", "Profesional Divers, Ltd.") then
  ; begin Edit mode
  ordTC.edit()
  ; correct spelling (Professional)
  ordTC.Name = "Professional Divers, Ltd."
  msgInfo("Success", "Corrected spelling error.")
else
  msgInfo("Search Failed",
          "Couldn't find \nProfesional Divers, Ltd.")
endIf
ordTC.endEdit()
endmethod
```

See also

- locateNext, locateNextPattern, locatePattern

locateNext

TCursor

Beginner

Method

Searches for a specified field value.

Syntax

1. **locateNext** (const ***fieldName*** String,
   const ***exactMatch*** AnyType [, const ***fieldName*** String,
   const ***exactMatch*** AnyType]*) Logical
2. **locateNext** (const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType [, const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType]*) Logical

Description

Searches a table for records whose values match the criteria specified in one or more field/value pairs. Specify the value to search for in *searchValue* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the next value matching *searchValue* is found, given the current view of the records. If the TCursor is using a secondary index, **locateNext** finds the next record in secondary index order—regardless of that record's primary index order.

The search begins with the record after the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be posted (for example, because of a key violation).

Example

This example uses **locate** and **locateNext** to count the number of records that have "FL" in the State/Prov field of the *Customer* table. The following code is attached to the *findFL* **pushButton** method.

```
; findFL::pushButton
method pushButton(var eventInfo Event)
var
  CustTC TCursor
  numFound LongInt
endVar
custTC.open("Customer.db")

if custTC.locate("State/Prov", "FL") then
  numFound = 1
  while custTC.locateNext("State/Prov", "FL")
    numFound = numFound + 1
  endWhile
  msgInfo("Records Found", String("Found ", numFound, " companies in FL"))
else
  msgInfo("Sorry", "Can't find FL in State/Prov field.")
```

```
endIf

endmethod
```

See also

❐ locate, locateNextPattern, locatePattern

locateNextPattern

TCursor

Method

Locates the next record containing a field that has a specified pattern of characters.

Syntax

1. **locateNextPattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locateNextPattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds sub-strings (for example, "comp" in "computer"). The search begins with the record after the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. If the TCursor is using a secondary index, **locateNextPattern** finds the next record in secondary index order—regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

For example, the following statement checks values the in the first field of each record. If a value is anything except "Borland", **locateNextPattern** returns True.

```
tc.locateNextPattern(1, "[^Borland]")
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database"):

```
tc.locateNextPattern("Name", "Borland", "Product", "Paradox", "Keywords",
"data*")
```

Example

In this example, assume the SOFTWARE.DB table exists in the current directory. Assume further that two of the fields are named Product and Name. This code (attached to the **pushButton** method) searches for records whose Name field contains "Borland" and whose Product field begins with "Par". This code keeps track of the matches found and stores field values in a resizeable array. When the method can't locate any more records that match the criteria, the results are displayed in a dialog box.

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
  myNames TCursor
  searchFor String
  numFound SmallInt
  productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Borland"

; this searches for records with "Borland" in the Name field
; and values starting with "Par" in the Product field
if myNames.locatePattern("Name", searchFor, "Product", "Par..") then
  numFound = 1
  productNames.grow(1)
  productNames[numFound] = myNames.Product

  ; now continue searching through fields with same criteria and
  ; store Product values in myNames array
  while myNames.locateNextPattern("Name", searchFor, "Product", "Par..")
    numFound = numFound + 1
    productNames.addLast(myNames.product)
  endWhile
endIf
if productNames.size() > 0 then
  productNames.view()
endIf
endmethod
```

See also

- locateNext, locatePattern
- advMatch in the String type

- advancedWildcardsInLocate and isAdvancedWildcardsInLocate in the Session type

locatePattern

TCursor

Method

Locates a record containing a field that has a specified pattern of characters.

Syntax

1. **locatePattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locatePattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds sub-strings (for example, "comp" in "computer"). The search always starts at the beginning of the table, but if no match is found, the TCursor returns to the original record. If a match is found, the TCursor moves to that record. If the TCursor is using a secondary index, **locate** finds the first record in secondary index order—regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search after the current record, use **locateNextPattern**. To start a search before the current record, use **locatePriorPattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

For example, the following statement checks values in the first field of each record. If a value is anything except "Borland", **locatePattern** returns True.

```
tc.locatePattern(1, "[^Borland]")
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, database).

```
tc.locatePattern("Name", "Borland", "Product", "Paradox", "Keywords", "data*")
```

Example

In this example, assume the SOFTWARE.DB table exists in the current directory. Assume further that two of the fields are named Product and Name. This code (attached to the **pushButton** method) searches for records whose Name field contains "Borland" and whose Product field begins with "Par". This code keeps track of the matches found and stores field values in a resizeable array. When the method can't locate any more records that match the criteria, the results are displayed in a dialog box.

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
  myNames TCursor
  searchFor String
  numFound SmallInt
  productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Borland"

; this searches for records with "Borland" in the Name field
; and values starting with "Par" in the Product field
if myNames.locatePattern("Name", searchFor, "Product", "Par..") then
  numFound = 1
  productNames.grow(1)
  productNames[numFound] = myNames.Product

  ; now continue searching through fields with same criteria and
  ; store Product values in myNames array
  while myNames.locateNextPattern("Name", searchFor, "Product", "Par..")
    numFound = numFound + 1
    productNames.addLast(myNames.product)
  endWhile
endIf
if productNames.size() > 0 then
  productNames.view()
endIf
endmethod
```

See also

- locate, locateNextPattern
- advMatch in the String type
- advancedWildcardsInLocate and isAdvancedWildcardsInLocate in the Session type

locatePrior

TCursor

Method

Searches for a specified field value.

Syntax

1. **locatePrior** (const ***fieldName*** String,
   const ***exactMatch*** AnyType [, const ***fieldName*** String,
   const ***exactMatch*** AnyType]*) Logical
2. **locatePrior** (const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType [, const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType]*) Logical

Description

Searches a table for records whose values match the criteria specified in one or more field/value pairs. Specify the value to search in *searchValue* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the previous value matching *searchValue* is found, given the current view of the records. If the TCursor is using a secondary index, **locatePrior** finds the previous record in secondary index order—regardless of that record's primary index order.

The search starts from the current record, and searches backwards in the table for the previous match. If a match is found, the TCursor moves to that record; otherwise, it returns to the original record. This operation fails if the current record cannot be committed (for example, because of a key violation). This method returns True if a successful match was made; otherwise, it returns False.

Example

In this example, the **pushButton** method for *showPrior* searches backwards through the *Lineitem* table for records with a certain order number. The *lineTC* variable is declared in the page's Var window, and opened to the *Lineitem* table in the **open** method for the page.

The following code goes in the Var window for *thisPage*:

```
; thisPage::var
Var
  lineTC TCursor
endVar
```

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
  lineTC.open("Lineitem")        ; open a TCursor for LineItem.db
endmethod
```

The following code is attached to the **pushButton** method for the *showPrior* button:

```
; showPrior::pushButton
method pushButton(var eventInfo Event)
var
```

```
  rec Array[] AnyType
endVar

if lineTC.locatePrior("Order No", 1005) then
  lineTC.copyToArray(rec)
  rec.view("Record #" + String(lineTC.recNo()))
else
  msgStop("Sorry", "No more records.")
endif
endmethod
```

See also

- locateNext, locateNextPattern, locatePattern

locatePriorPattern

TCursor

Method

Locates the previous record containing a field that has a specified pattern of characters.

Syntax

1. **locatePriorPattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locatePriorPattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds sub-strings (for example, "comp" in "computer"). The search begins with the record before the current record. If a match is found, the TCursor moves to that record. If no match is found, the TCursor returns to the original record. If the TCursor is using a secondary index, **locatePriorPattern** finds the previous record in secondary index order—regardless of that record's primary index order.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern

operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

For example, the following statement checks values in first field of each record. If a value is anything except "Borland," **locatePriorPattern** returns True.

```
tc.locatePriorPattern(1, "[^Borland]")
```

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the following statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database"):

```
tc.locatePriorPattern("Name", "Borland", "Product", "Paradox", "Keywords",
"data*")
```

Example

In this example, the **pushButton** method for *showPriorPtrn* searches backwards through the *Software* table for records with a certain company and product name. The *tc* variable is declared in the page's Var window, and opened to the *Software* table in the **open** method for the page.

The following code goes in the Var window for *thisPage*:

```
; thisPage::var
Var
  tc        TCursor
  searchFor String
endVar
```

The following code is attached to the **open** method for *thisPage*:

```
; thisPage::open
method open(var eventInfo Event)
  tc.open("Software.db")  ; open TCursor for Software.db
  tc.end()                ; move TCursor to the last record
  searchFor = "Borland"
endmethod
```

The following code is attached to the **pushButton** method for the *showPriorPtrn* button:

```
; showPrior::pushButton
method pushButton(var eventInfo Event)
var
  rec Array[] AnyType
endVar

; search for the previous pattern
if tc.locatePriorPattern("Name", searchFor, "Product", "Par..") then
  tc.copyToArray(rec)
  rec.view("Record #" + String(tc.recNo()))
else
  msgStop("Sorry", "No more records.")
endif
endmethod
```

See also

- locateNextPattern, locatePattern
- advMatch in the String type
- advancedWildcardsInLocate and isAdvancedWildcardsInLocate in the Session type

lock

TCursor

Beginner

Method

Places specified locks on a specified table.

Syntax

lock (const ***lockType*** String) Logical

Description

Attempts to place a lock on the TCursor, where *lockType* is one of the following String values: Write, Read, Full, or Any. If successful, this method returns True; otherwise, it returns False.

Example

The following example opens a TCursor for *Customer*, places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if tc.open(pdoxTbl) then
  if tc.lock("Full") then     ; attempt to place Full lock
    tc.reIndex("Phone_Zip")   ; rebuild Phone_Zip index
    tc.unLock("Full")         ; unlock the table
    message("Phone_Zip rebuilt.")
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endif
endif
endmethod
```

See also

- lockStatus, unlock

lockRecord

TCursor

Beginner

Method

Puts a write lock on the current record.

Syntax

lockRecord ( ) Logical

Description

Paradox places a write lock on a record when you begin to make changes (an implicit record lock). **lockRecord** attempts to place a write lock on the record pointed to by a TCursor (an explicit record lock) and if successful, returns True; otherwise, it returns False.

Example

In the following example, the **pushButton** method for *thisButton* searches for a record in the *Customer* table. If the search is successful, this example attempts to lock the record with **lockRecord**. When the record has been locked, a custom procedure is called to get new customer information from the user. If **lockRecord** is not successful, the user is asked to try again later.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC, myCustTC TCursor
endVar
custTC.open("Customer.db")

; attempt to locate record in Customer.db
if custTC.locatePattern("Name", "Jamaica..") then
  custTC.edit()
  if custTC.lockRecord() then         ; attempt to lock the record
    custTC.initRecord()               ; initialize record to the defaults
    getCustInfo()                     ; call a custom procedure
  else                                ; otherwise record couldn't be locked
    msgStop("Sorry", "Can't lock record. \n Try again later.")
  endif
else
  msgStop("Sorry", "Can't find record.")
endif

endmethod
```

See also

- unlockRecord

lockStatus

TCursor

Method

Returns the number of times you have placed a lock on a table.

Syntax

lockStatus (const ***lockType*** String) SmallInt

Description

Returns the number of times you have placed a lock of type *lockType* on a table, where *lockType* is one of the following String values: Write, Read, Full, or Any.

If you haven't placed any locks of a given type, **lockStatus** returns 0.

If you specify "Any" for *lockType*, **lockStatus** returns the total number of locks you've placed on the table. **lockStatus** reports only on locks you've placed explicitly, not on locks placed by Paradox or by other users or applications.

Example

This example uses **lockStatus** to determine whether you've placed any write locks on the *Customer* table; if so, all write locks are removed.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  tblName String
endVar

tc.open("Customer.db")

; loop until all write locks are removed from Customer
while tc.lockStatus("Write") > 0
  tc.unlock("Write")
endWhile
message("All write locks removed from Customer.db")

endmethod
```

See also

- lock, unlock

---

moveToRecNo

TCursor

Method

Moves a TCursor to a specific record in a table.

Syntax

moveToRecNo (const ***recordNum*** LongInt) Logical

Description

Sets the current record to the record specified in *recordNum*. It returns an error if *recordNum* is not in the table. Use the **nRecords** method or examine the NRecords property to find out how many records a table contains. This method is recommended only for dBASE tables. When used for a Paradox table, **moveToRecNo** behaves exactly like the **moveToRecord** method.

Example

This example moves a TCursor to the sixth record in a dBASE table.

```
; gotoRecSix::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.dbf")     ; suppose Orders.db has 10 records
if not tc.moveToRecNo(6) then
  msgStop("Sorry", "Can't post current record.")
```

```
endif
endmethod
```

See also

- end, home, moveToRecord, nextRecord, nRecords, priorRecord, skip

moveToRecord

TCursor

Method

Moves a TCursor to a specific record in a table.

Syntax

moveToRecord (const ***recordNum*** LongInt) Logical

Description

Sets the current record (and the record buffer) to the record specified in *recordNum*. It returns an error if *recordNum* is greater than the number of records in the table. Use the method **nRecords** or examine the NRecords property to find out how many records a table contains. This method can be very slow for dBASE tables; use **moveToRecNo** instead.

This operation fails if the current record cannot be committed (for example, because of a key violation).

Example

In this example, the **pushButton** method attempts to move a TCursor to the sixth record in the *Orders* table. If the TCursor can't be moved (because the current record can't be posted) this method displays a warning message.

```
; gotoRecSix::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")      ; suppose Orders.db has 10 records
if not tc.moveToRecord(6) then
  msgStop("Sorry", "Can't post current record.")
endif
endmethod
```

See also

- end, home, nextRecord, nRecords, priorRecord, skip

nextRecord

Beginner

TCursor

Method

Moves to the next record in a table.

Syntax

nextRecord ( ) Logical

Description

Sets the current record to the next record in the table. If the table is in Edit mode, **nextRecord** commits changes to the current record before moving. This operation fails if the current record cannot be committed (for example, because of a key violation).

nextRecord returns False if you try to move past the end of the table. Also, the last record of the table becomes the current record, and **eot** returns True.

Example

In this example, the **pushButton** method for *showNextCust* uses **nextRecord** to move a TCursor through the *Customer* table. Each time the TCursor lands on a new record, the code uses **copyToArray** to copy the contents of the record to a DynArray, then displays field values in a dialog box. When **nextRecord** attempts to move beyond the last record in the table, **eot** returns True and the **pushButton** method terminates.

```
; showNextCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  scratch DynArray[] AnyType
  tblName String
endVar
tblName = "Customer.db"

if tc.open(tblName) then

  while NOT tc.eot()          ; True until nextRecord attempts to move
                              ; beyond the end the table
    tc.copyToArray(scratch)   ; copy the record to scratch DynArray
    scratch.view("Record " + String(tc.recNo()))
    if msgQuestion("",
       "Do you want to see the next record?") = "Yes" then
      tc.nextRecord()         ; move down one record
    else
      return
    endif
  endWhile

  msgStop("That's it!", "No more records.")

else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- end, home, moveToRecord, priorRecord, skip

nFields

TCursor

Method

Returns the number of fields in a table.

Syntax **nFields ( )** LongInt

Description Returns the number of fields in the table associated with a TCursor.

Example In this example, the **pushButton** method for *thisButton* displays the number of fields in the *BioLife* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
if tc.open("BioLife.db") then
  msgInfo("Number of BioLife fields", tc.nFields())
else
  msgStop("Sorry", "Can't open BioLife.db table")
endif

endmethod
```

See also
- nKeyFields, nRecords

---

nKeyFields

TCursor

Method Returns the number of fields in the current index of a table.

Syntax **nKeyFields ( )** LongInt

Description Returns the number of fields in the current index of the table associated with a TCursor. When used on a Paradox table, this method works with the primary index; when used on a dBASE table, it works with the index specified by the Table type method **setIndex**.

Example This example reports the number of key fields in a Paradox table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  pdoxTC, dBASETC TCursor
  nkf LongInt
  pdoxTbl, dBASETbl String
  tb Table
endVar
pdoxTbl = "Orders.db"
dBASETbl = "Scores.dbf"

if pdoxTC.open(pdoxTbl) then
  nkf = pdoxTC.nKeyFields()       ; number of key fields in the primary index
  msgInfo(pdoxTbl, pdoxTbl + " has " + String(nkf) + " key fields.")
else
  msgInfo("Sorry", "Can't open " + pdoxTbl + " table.")
endif
```

TCursor

```
endmethod
```

See also

- nFields, nRecords

nRecords

TCursor

Beginner

Method

Returns the number of records in a table.

Syntax

nRecords ( ) LongInt

Description

Returns the number of records in the table associated with a TCursor. This operation can take a long time for large tables.

When working with a dBASE table, **nRecords** counts deleted records if **showDeleted** is turned on. If **showDeleted** is turned off, deleted records are not counted.

Example

In this example, the **pushButton** method for *thisButton* runs a custom method if there are more than 10000 records in ORDERS.DB; otherwise, the code displays the current number of records in *Orders*.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  ordTC TCursor
  nOrders LongInt
endVar
if ordTC.open("Orders.db") then
  nOrders = ordTC.nRecords()
  if nOrders > 10000 then    ; if Orders has more than 10000 records
    archiveOldOrders()       ; run a custom method
  else
    msgInfo("Status", "Orders table has " + String(nOrders) + " records.")
  endif
else
  msgStop("Sorry", "Can't open Orders table.")
endif
endmethod
```

See also

- nFields, nKeyFields

open

TCursor

Beginner

Method

Opens a table.

Syntax

1. open (const ***tableName*** String [, const ***db*** DataBase]
[, const ***indexName*** String]) Logical

2. open (const ***tableVar*** Table) Logical

Description

Associates a TCursor with the table named in *tableName*. If you use syntax 1, where *tableName* is a String, you can use arguments *db* and *indexName* to specify a database and an index. If you use syntax 2, where *tableVar* is the name of a Table variable, you can use the Table method **setIndex** to specify an index, and you can specify the database using the Table method **attach**.

Example

The following example uses the first syntax to open a TCursor on the *Customer* table in the "SampleTables" database. This code uses the optional *indexName* clause, so the TCursor uses the "NameAndState" index. The following code is attached to the **pushButton** method for *firstButton*:

```
; firstButton::pushButton
method pushButton(var eventInfo Event)
var
  tcl  TCursor
  samp DataBase
endVar

; create the SampleTables alias for the default sample directory
addAlias("SampleTables", "Standard", "c:\\pdoxwin\\sample")

; associate the samp DataBase var with SampleTables database
samp.open("SampleTables")

; associate tcl to the Customer table in samp database,
; and use the NameAndState index
tcl.open("Customer.db", samp, "NameAndState")

endmethod
```

The next example achieves the same as the previous example, but uses the second form of the syntax where a *tableVar* is used. The following code is attached to the **pushButton** method for *secondButton*:

```
; secondButton::pushButton
method pushButton(var eventInfo Event)
var
  tcl  TCursor
  samp DataBase
  tbl  Table
endVar

; create the SampleTables alias for the default sample directory
addAlias("SampleTables", "Standard", "c:\\pdoxwin\\sample")

; associate the samp DataBase var with SampleTables database
samp.open("SampleTables")

; attach the tbl Table handle to Customer in the samp database
tbl.attach("Customer.db", samp)
; set the tbl index to the NameAndState index
```

```
tbl.setIndex("NameAndState")

; now associate tcl TCursor to Customer table in samp database
tcl.open(tbl)

endmethod
```

See also

- close

postRecord

TCursor

Beginner

Method

Posts changes to a record.

Syntax

postRecord ( ) Logical

Description

Posts changes to a record immediately. The record remains locked. If the record is in an indexed table, it moves to its appropriate position, and remains the current record. This method returns True if successful; otherwise, it returns False.

Example

For this example, the **pushButton** method for the *fixName* button attempts to find a misspelled name in the *Customer* table. If the erroneous name is found, the code corrects it, then posts changes with **postRecord**.

```
; fixName::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  badName String
endVar
badName = "Usco"
goodName = "Unisco"

tc.open("Customer.db")
if tc.locate("Name", badName) then ; if the erroneous name is found
  tc.edit()                        ; put TCursor in Edit mode
  tc.Name = goodName               ; correct misspelled name
  if tc.postRecord() then          ; True if record is posted
    message("Changes posted.")
  else                             ; record is not posted (Key violation?)
    msgStop("PostRecord", "Can't post these changes.")
  endIf
  tc.endEdit()                     ; end Edit mode
  ; If the record was committed, endEdit simply ends Edit mode--the Name
  ; field now stores "Unisco". If the record was not committed, the field
  ; retains its original value ("Usco").

else                               ; can't find "Usco" in Name field
   message("Can't find " + badName)
endif
endmethod
```

See also

- unLockRecord

priorRecord

Beginner

TCursor

Method

Moves to the previous record in a table.

Syntax

priorRecord ( ) Logical

Description

Sets the current record to the previous record in a table. If the table is in Edit mode, **priorRecord** commits changes to the current record before moving. It returns False if the TCursor is already at the first record. Also, the first record of the table becomes the current record, and **bot** returns True.

priorRecord may not be appropriate in all databases, because some may not be bi-directional. This operation fails if the current record cannot be committed (for example, because of a key violation).

Example

In this example, the **pushButton** method for *showPrevCust* uses **priorRecord** to move a TCursor backwards through the *Customer* table. Each time the TCursor lands on a new record, this code uses **copyToArray** to copy the contents of the record to a DynArray and display field values in a dialog box. When **priorRecord** attempts to move beyond the beginning of the table, **bot** returns True and the **pushButton** method terminates.

```
; showPrevCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  scratch DynArray[] AnyType
  tblName String
endVar
tblName = "Customer.db"

if tc.open(tblName) then

  tc.end()                     ; move to end of table
  while NOT tc.bot()           ; True until priorRecord attempts to move
                               ; beyond the beginning of the table
    tc.copyToArray(scratch)    ; copy the record to scratch DynArray
    scratch.view("Record " + String(tc.recNo()))
    if msgQuestion("",
       "Do you want to see the next record?") = "Yes" then
      tc.priorRecord()         ; move up one record
    else
      return
    endif
  endWhile

  msgStop("That's it!", "No more records.")
```

```
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

- end, home, moveToRecord, nextRecord, skip

qLocate

TCursor

Method

Searches an indexed table for a specified field value.

Syntax

qLocate (const ***searchValue*** AnyType [, const ***searchValue*** AnyType]*) Logical

Description

Searches an indexed table for records whose values exactly match the criteria specified in *searchValue*. **qLocate** searches for values in the active index; the first value corresponds to the first field in the index, the second value corresponds to the second field in the index, and so on.

The search always starts from the beginning of the table, but if no match is found, the TCursor returns to the original record. If a match is found, the TCursor moves to that record. This operation fails if the current record cannot be posted or if the number of search values exceeds the number of fields in the current index.

Example

This code uses **qLocate** to find a key value in the *Lineitem* table:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

if tc.open("Lineitem.db") then

  ; if qLocate can find 1002 in the first field of the
  ; index and 1316 in the second field of the index
  if tc.qLocate(1002, 1316) then

    ; make some changes to the record
    tc.edit()
    tc.Qty = 10
    tc.Total = tc."Selling Price" * tc.Qty
    tc.close()
  else
    msgStop("Sorry", "Can't find specified record.")
  endif
else
  msgStop("Error", "Can't open Lineitem.db")
endif

endmethod
```

See also

- locate, locateNext, locateNextPattern, locatePattern, locatePrior, locatePriorPattern

recNo

TCursor

Method

Returns the record number of the current record.

Syntax

recNo ( ) LongInt

Description

Returns an integer representing the current record's position in the table. For a dBASE table, **recNo** returns the physical position of the record in the table; for an indexed Paradox table, it returns the record's sorted position.

Example

In this example, the **pushButton** method for *thisButton* searches the *Customer* table for customers residing in Oregon. If any are found, this code stores record numbers in an array, then displays the contents of the array in a dialog box.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  ar Array[] SmallInt
  tblName String
endVar
tblName = "Customer.db"

tc.open(tblName)
if tc.locate("State/Prov", "OR") then
  ar.addLast(tc.recNo())                       ; add record number to array
  while tc.locateNext("State/Prov", "OR")  ; find the next "OR"
    ar.addLast(tc.recNo())                     ; add more array elements
  endWhile
  ar.view("Record Numbers")                    ; display ar array
else
  msgInfo("Nothing to do!", "Can't find \"OR\" in \"State/Prov\" field")
endif
endmethod
```

See also

- nRecords

recordStatus

TCursor

Method

Reports about the status of a record.

Syntax

recordStatus (const ***statusType*** String **)** Logical

TCursor

Description

Returns True or False to a question to report about the status of a record. Use the argument *statusType* to specify the status to ask about, where *statusType* is one of the following String values: New, Locked, or Modified.

"New" means the record has just been inserted into the table and is not yet posted to the table. "Locked" means a lock (implicit or explicit) has been placed on the record. "Modified" means at least one of the field values has been changed and is not yet posted to the table.

Example

This example tests whether the current record is locked. If the record is not locked, this method uses **lockRecord** to lock the record; otherwise this example informs the user that the record has previously been locked.

```
; lockThisRecord::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("orders.db")
tc.edit()

; if the current record is NOT locked
if tc.recordStatus("Locked") = False then
  ; lock the current record
  tc.lockRecord()

  ; if record is locked, this statement will display True
  msgInfo("Record Status", "recordStatus(\"Locked\") = " +
                        String(tc.recordStatus("Locked")))
else
  message("Current record is already locked.")
endIf

endmethod
```

See also

- lockRecord, unlockRecord

---

reIndex

TCursor

Method

Rebuilds specified index files.

Syntax

reIndex (const ***IndexName*** String [, const ***TagName*** String]) Logical

Description

Rebuilds an index (or index tag) that is not automatically maintained. When working with a Paradox table, use *indexName* to specify an index (the field name, for a single-field index, or the full name of a composite index). When working with a dBASE table, use *indexName*

to specify a .NDX file, or *indexName* and *tagName* to specify an index tag in a .MDX file. This method requires exclusive access to the table.

Example

The following example opens a TCursor for *Customer* (a Paradox table), gains exclusive access to the table, then uses **reIndex** to rebuild the *Phone_Zip* index.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc      TCursor
  pdoxTbl String
  tb      Table
endVar
pdoxTbl = "Customer.db"

tb.attach(pdoxTbl)
tb.setExclusive(Yes)

if tc.open(tb) then
  tc.reIndex("Phone_Zip")         ; rebuild Phone_Zip index
  message("Phone_Zip reindexed.")
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endif

endmethod
```

See also

❐ reindexAll

reIndexAll

TCursor

Method

Rebuilds all index files for a table.

Syntax

reIndexAll ( ) Logical

Description

Rebuilds all indexes for the table associated with a TCursor. This method requires exclusive rights to the table to rebuild a maintained index, and it requires a write lock to rebuild a non-maintained index. **reIndexAll** only works with Paradox tables, because any index opened for a dBASE table is maintained automatically.

Example

For this example, the **pushButton** method for a button attempts to place a full lock on the *Customer* table. If **lock** is successful, this code rebuilds all indexes for the *Customer* table then unlocks the table.

```
; reindexAllCust::pushButton
method pushButton(var eventInfo Event)
var
  tc      TCursor
  pdoxTbl String
  tb      Table
```

```
endVar
pdoxTbl = "Customer.db"

tb.attach(pdoxTbl)
tb.setExclusive(Yes)

if tc.open(tb) then
  tc.reIndexAll()              ; rebuild all Customer indexes
  message("Indexes rebuilt.")
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endif
endmethod
```

See also

- reIndex

seqNo

TCursor

Method

Returns the record number of the current record.

Syntax

seqNo ( ) LongInt

Description

Returns an integer representing the current record's position in a table. For dBASE tables, **seqNo** returns the sequential position of a record as viewed by the current index. For Paradox tables, **seqNo** and **recNo** always return the same value.

Example

In this example, assume SCORES.DBF has three records and the second record has been deleted. The code attached to the **pushButton** method for *testSeqNo* demonstrates the difference between **seqNo** and **recNo** methods.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

  ; Scores.dbf has 3 records and the second record is deleted
tc.open("Scores.dbf")

  ; do not show deleted records
tc.showDeleted(No)

  ; this displays recNo() = 1
  ;               seqNo() = 1
msgInfo("tc Status", "recNo() = " + String(tc.recNo()) + "\n" +
                     "seqNo() = " + String(tc.seqNo()))

  ; move to the last record in the table
tc.end()

  ; this displays   recNo() = 3
  ;                 seqNo() = 2   (record number 2 is deleted)
```

```
msgInfo("tc Status", "recNo() = " + String(tc.recNo()) + "\n" +
                     "seqNo() = " + String(tc.seqNo()))

endmethod
```

See also

- moveToRecNo, moveToRecord, recNo

setFieldValue

TCursor

Method

Assigns a value to a specified field.

Syntax

setFieldValue (const ***fieldName*** String, const ***value*** AnyType) Logical

setFieldValue (const ***fieldNum*** SmallInt, const ***value*** AnyType) Logical

Description

Sets the value of a field (*fieldName* or *fieldNum*) to *value*. This method returns True if successful; otherwise, it returns False.

You can achieve the same results using dot notation. For example, this statement uses dot notation to change the value in the Last Bid field:

```
tcVar."Last Bid" = 32.25
```

The following statement uses **setFieldValue** to achieve the same results:

```
tcVar.setFieldValue("Last Bid", 32.25)
```

Example

In this example, the **pushButton** method for *correctName* locates a misspelled name in the Name field, then uses **setFieldValue** to replace the original name.

```
; correctName::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  badName, goodName String
endVar

badName = "Usco"
goodName = "Unisco"
tc.open("Customer.db")
if tc.locate("Name", badName) then
  tc.edit()
  tc.setFieldValue("Name", goodName)    ; correct misspelled name
  tc.postRecord()                       ; post record to the table
  tc.endEdit()                          ; end Edit mode
  message("Usco replaced with Unisco.")
else                                    ; can't find "Usco" in Name field
  message("Can't find " + badName)
```

TCursor

```
endif
endmethod
```

See also

- fieldValue

setFilter

TCursor

Method

Sets the range of records a TCursor can point to.

Syntax

setFilter ([const ***exactMatchVal*** AnyType,] *
const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description

Specifies conditions for including a range of records. Records that meet the conditions are included, records that don't are filtered out. This operation fails if the current record cannot be committed or if the TCursor does not point to a keyed table.

This method compares the criteria you specify with values in the corresponding fields of a table's index. To filter records based on the value of a single field, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record. If a value is less than 14 or greater than 88, that record is filtered out.

```
tcVar.setFilter(14, 88)
```

To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55:

```
tcVar.setFilter(55, 55)
```

You can filter records based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tcVar.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table. If you don't call this method before opening a TCursor for a dBASE table, deleted records are not shown.

Example

For this example, assume that the first field in *Lineitem*'s key is "Order No." and you want to know the total for order number 1005. When you press the *getDetailSum* button, the **pushButton** method

opens a TCursor for *Lineitem*, then limits the number of records included in the TCursor to those with 1005 in the first key field. After the call to **setFilter**, this example uses **cSum** to display the sum of the Total field. Because the TCursor is pointing only to order number 1005, **cSum** reports summary information only for that order.

```
; getDetailSum::pushButton
method pushButton(var eventInfo Event)
var
  lineTC TCursor
  tblName String
endVar
tblName = "LineItem.db"
if lineTC.open(tblName) then

  ; this limits TCursor's view to records that have
  ; 1005 as their key value (Order No. 1005).
  lineTC.setFilter(1005, 1005)

  ; now display the total for Order No. 1005
  msgInfo("Total for Order 1005", lineTC.cSum("Total"))
else
  msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endmethod
```

See also

❐ reIndex, reIndexAll, switchIndex

setFlyAwayControl

TCursor

Method

Controls whether the TCursor points to a record whose position has changed as the result of an **unlockRecord**.

Syntax

setFlyAwayControl ([const ***yesNo*** Logical]) Logical

Description

Specifies in *yesNo* whether the TCursor will stay on the record after a successful call to **unlockRecord**.

When you're working with indexed tables, the **didFlyAway**, **setFlyAwayControl**, and **unlockRecord** methods are closely related. When you call **unlockRecord**, the record is posted (if no key violation exists) to the table and moved to its sorted position. Depending on whether the record moved to a new position, the TCursor may not continue to point to the posted record. This behavior is referred to as *record flyaway*.

You can use the **setFlyAwayControl** method to control the behavior of **unlockRecord** and record flyaway. If the optional argument *yesNo* is Yes, **setFlyAwayControl** guarantees that the TCursor will point to the posted record after a call to **unlockRecord**; if set to No, the TCursor points to the record following the original position of the

record. You can use the **didFlyAway** method to test whether the record did, in fact, fly away.

When **setFlyAwayControl** is set to Yes, Paradox performs record-level checking for many cursor level operations. Because this extra work can slow an application down, **setFlyAwayControl** should be used with caution. The **postRecord** method is usually preferred over **setFlyAwayControl** and **unlockRecord**. See the entry for **postRecord** earlier in this section for more information.

Example

See the example for the didFlyAway method.

See also

- didFlyAway, postRecord, unLockRecord

showDeleted

TCursor

Method

Specifies whether to show deleted records in a dBASE table.

Syntax

showDeleted ([*yesNo*]) Logical

Description

Specifies whether to show deleted records in a dBASE table. You can use *yesNo* to specify Yes to show deleted records, or No if you don't want to show them. If omitted, *yesNo* is Yes by default. **showDeleted** is valid only for dBASE tables because deleted records in a Paradox table cannot be shown.

Example

In this example, the **pushButton** method attached to *showDeletedRecs* calls **showDeleted** to show deleted records in ORDERS.DBF.

```
; showDeletedRecs::pushButton
method pushButton(var eventInfo Event)
var
  dbfTC TCursor
endVar
if dbfTC.open("Orders.dbf") then
  dbfTC.showDeleted(Yes)
else
  msgStop("Sorry", "Can't open Orders.dbf table.")
endIf
endmethod
```

See also

- compact, isRecordDeleted, isShowDeletedOn

skip

TCursor

Method

Moves forward or backward a specified number of records in a table.

Syntax

skip ([const ***nRecords*** LongInt]) Logical

Description

Sets the current record (and the record buffer) to the record *nRecords* from the current record. If **skip** tries to move beyond the limits of the table, you'll get an error, and the current record will be the first or last record of the table, as appropriate. This operation fails if the current record cannot be committed (for example, because of a key violation).

Positive values for *nRecords* move forward through the table (**skip**(1) is the same as **nextRecord**), negative values move backward (**skip**(-1) is the same as **priorRecord**), and a value of 0 doesn't move (**skip**(0) is the same as **currRecord**). If omitted, *nRecords* is 1 by default.

Example

This example demonstrates how **skip** affects a TCursor's record position in a table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Orders.db")

tc.skip(5)      ; ahead 5 records. tc.recNo() = 6
tc.skip(-3)     ; back 3 records. tc.recNo() = 3
tc.skip(-5)     ; fails--attempted to move beyond the
                ; beginning of the table.
                ; tc.recNo() = 1
                ; tc.bot() = True

endmethod
```

See also

- currRecord, end, home, moveToRecord, nextRecord, priorRecord

sortTo

TCursor

Method

Sorts a table.

Syntax

1. **sortTo** (const ***destTable*** String, const ***numFields*** SmallInt,
   const ***sortFields*** Array[] String,
   const ***sortOrder*** Array[] SmallInt) Logical

TCursor

2. sortTo (const *destTable* Table, const *numFields* SmallInt, const *sortFields* Array[] String, const *sortOrder* Array[] Smallint) Logical

Description

Sorts a table based on values of fields, and puts the results into *destTable*.

sortFields is an array of strings or integers specifying the fields on which to sort. The size of the *sortFields* array is specified in *numFields*. *sortOrder* is an array of integers, where a value of 0 specifies a sort in ascending order, and a value of 1 specifies descending order. The two arrays must be the same size, specified in *numFields*. Element 1 of *sortOrder* specifies how to sort the field named in element 1 of *sortFields*, and so on.

This method requires at least a read only lock on the source table, and a full lock on the destination table. If *destTable* exists, it will be overwritten without asking for confirmation. If *destTable* is open, this method fails. You cannot use **sortTo** to sort a table onto itself.

Example

This example sorts the *Customer* table to the CUSTSORT.DB table, then opens the sorted table. If the *Customer* table cannot be write-locked, this example informs the user of the error and aborts the operation. If the *CustSort* destination table exists, the user is given an opportunity to continue or abort.

The following code goes in the Var window for the *sortCustButton* button:

```
; sortCustButton::var
var
  sortFlds Array[2]  String
  sortOrder Array[2] SmallInt
  tc                 TCursor
  srcTbl, destTbl    String
  noSort             Logical
  sortTbl            TableView
endVar
```

The following code is attached to the button's **open** method. This code assigns **open** a TCursor for the *Customer* table and initializes the array elements. These assignments determine the sort criteria for **sortTo**.

```
; sortCustButton::pushButton
method open(var eventInfo Event)
srcTbl = "Customer.db"
destTbl = "CustSort.db"
if tc.open(srcTbl) then
  noSort = False                  ; flag for pushButton method
  sortFlds[1] = "First Contact"   ; sort by First Contact
  sortOrder[1] = 0                ; in ascending order

  sortFlds[2] = "Country"         ; then by Country
  sortOrder[2] = 0                ; in descending order
```

```
else
  noSort = True
endif

endmethod
```

The following code is attached to the **pushButton** method for the *sortCustButton* button. When the button is pressed, the code attempts to place a write lock on the source table (CUSTOMER.DB), prompts the user if the destination table exists (CUSTSORT.DB), then sorts *Customer* to *CustSort* based on the values in the *sortFlds* and *sortOrder* arrays. After CUSTSORT.DB is created (or overwritten), this example opens it as a TableView.

```
; sortCustButton::pushButton
method pushButton(var eventInfo Event)
if noSort = False then
  if tc.lock("Write") then
    if isTable(destTbl) then
      if msgQuestion("Overwrite?",
         "Do you want to replace " + destTbl + " table?")  "Yes" then
        msgInfo("Canceled", "Operation canceled.")
        return
      endif
    endif
    tc.sortTo(destTbl, 2, sortFlds, sortOrder)
    sortTbl.open(destTbl)
  else
    msgStop("Stop!", "Can't write-lock " + srcTbl + " table.")
  endif
else
  msgStop("Sorry", "Can't open " + srcTbl + " table.")
endif
endmethod
```

See also

❐ add, copy, subtract

subtract

TCursor

Method

Subtracts the records in one table from another table.

Syntax

1. subtract (const ***destTable*** String) Logical

2. subtract (const ***destTable*** Table) Logical

3. subtract (const ***destTable*** TCursor) Logical

Description

Checks whether any records in the source table are also in *destTable*. If so, **subtract** deletes them from *destTable* without asking for confirmation.

If *destTable* is indexed, **subtract** deletes all records with indexes that exactly match values in corresponding index fields in the source table. If *destTable* is not indexed, **subtract** deletes all records that

exactly match any record in the source table. Whether tables are indexed or not, this method considers only fields that *could* be keyed. For example, numeric fields are considered, but formatted memos are not. This method requires read/write access to both tables.

Note If the destination table is not indexed, this operation can be expensive.

Example

In this example, the **pushButton** method for *subtractCust* deletes records from the *Customer* table that exactly match those in the *Answer* table.

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
  ansTC, custTC TCursor
endVar

if ansTC.open("Answer.db") and
   custTC.open("Customer.db") then

  ansTC.subtract(custTC)           ; subtract Answer records from Customer

else
  msgStop("Stop!", "Can't open tables.")
endif

endmethod
```

See also

❒ add, copy

switchIndex

TCursor

Beginner

Method

Specifies another index to use to view the records in a table.

Syntax

1. **switchIndex** ([const ***indexName*** String]
   [, const ***stayOnRecord*** Logical]) Logical
2. **switchIndex** ([const ***indexFileName*** String
   [, const ***tagName*** String]] [, const ***stayOnRecord*** Logical])
   Logical

Description

Specifies in *indexName* an index file to use with a table. In syntax 1, *indexName* specifies an index to use with a Paradox table. Syntax 2 is for dBASE tables, where *indexFileName* can specify a .NDX file or a .MDX file, and optional argument *tagName* specifies an index tag in a production index (.MDX) file.

In both syntaxes, if optional argument *stayOnRecord* is Yes, this method maintains the current record after the index switch; if it is

No, the first record in the table becomes the current record. If omitted, *stayOnRecord* is No by default.

Example

In this example, assume that the *Customer* is a keyed Paradox table that has a secondary index named "NameAndState". This example opens a TCursor for *Customer*, calls **switchIndex** to switch from the pimary index to the "NameAndState" index, then displays the first value in the Name field. Since the TCursor is sorted on Name and State fields in ascending order, the field value displayed is the first name in ascending sort order.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endvar

tc.open("Customer.db")              ; open TCursor for Customer
tc.switchindex("NameAndState")      ; switch to index NameAndState
tc.home()                           ; make sure we're on the first record
msgInfo("First Record", tc.Name)    ; display value in Name field

endmethod
```

See also

- reIndex, reIndexAll
- setIndex in the Table type

tableName

TCursor

Method

Returns the name of the table associated with a TCursor.

Syntax

tableName ( ) String

Description

Returns the name of the table associated with a TCursor. This method is useful when you're passing variables to the TCursor **open** method.

Example

In this example, the **pushButton** method for *thisButton* uses **findFirst** and **findNext** methods from the FileSystem type to locate Paradox tables in the current working directory. This example searches each table for a value in the *Name* field of the current table. This example opens all of the tables in the current directory that have "Unisco" in the "Name" field.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fs FileSystem
  tc TCursor
  tb TableView
```

TCursor

```
endVar
if fs.findFirst("*.db") then
  while fs.findNext()
    tc.open(fs.Name())                    ; open TCursor for a .db file
    if tc.locate("Name", "Unisco") then ; if we find Unisco in Name field
      tb.open(tc.tableName())             ; open table associated with TCursor
    endIf
    tc.close()
  endWhile
endIf

endmethod
```

See also

- tableRights

tableRights

TCursor

Method

Reports about the operations you can perform on a table.

Syntax

tableRights (const ***rights*** String) Logical

Description

Reports about a user's rights to a table, where *rights* is one of:

- "ReadOnly" (read from the table, but don't change it)
- "Modify" (enter or change data)
- "Insert" (add new records)
- "InsDel" (add and delete records)
- "All" (perform all operations)

Example

This example reports whether the user has "InsDel" rights to the *Orders* table.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
   myRights Logical
   ordersTC TCursor
endVar
ordersTC.open("orders.db")
ordersTC.edit()
myRights = ordersTC.tableRights("InsDel")

  ; this displays True if you have InsDel rights to Orders.db
msgInfo("Rights to Enter?", myRights)

endmethod
```

See also

- fieldRights

type

TCursor

Method

Returns the type of a table.

Syntax

type ( ) String

Description

Returns a string describing the type of a table, either Paradox or dBASE.

Example

This example compacts (removes deleted records) from the *Orders* table if **type** returns dBASE; otherwise a message indicates that *Orders* is a Paradox table.

```
; compact::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar

tc.open("Orders.db")

; if Orders.db is a dBASE table
if tc.type() = "dBASE" then
  ; remove deleted records
  tc.compact()
else
  ; otherwise, display the type of table
  msgStop("Stop!", "Orders.db is a " + tc.type() + " table.")
endif

endmethod
```

See also

- isAssigned, tableName

unDeleteRecord

TCursor

Beginner

Method

Undeletes the current record from a dBASE table.

Syntax

unDeleteRecord ( ) Logical

Description

Undeletes the current record of a dBASE table. This operation can only be successful if **showDeleted** has been set to True, the current record is a deleted record, and the TCursor is in Edit mode.

Example

This example opens a TCursor for SCORES.DBF (a dBASE table), then uses **showDeleted** to display all deleted records. Then, the code attempts to locate a specific record in the table. This example uses

isRecordDeleted to determine whether the record has been deleted; if it has, it is undeleted with **undeleteRecord**. The following code is attached to the **pushButton** method for *thisButton*:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
tc.open("Scores.dbf")                  ; open TCursor on a dBASE table
tc.showDeleted()                       ; show deleted records
if tc.locate("Name", "Jones") then     ; if locate finds Jones in Name field
  if tc.isRecordDeleted() then         ; if the record has been deleted
    tc.edit()                          ; begin Edit mode
    tc.undeleteRecord()                ; undelete the record
    message("Jones record undeleted")
  endif
else
  msgStop("Error", "Can't find Jones.")
endif
endmethod
```

See also

- deleteRecord, isRecordDeleted, isShowDeletedOn, showDeleted

---

unlock

TCursor

Beginner

Method

Removes specified locks from a TCursor.

Syntax

unlock (const ***lockType*** String) Logical

Description

Attempts to remove locks explicitly placed on the table pointed to by a TCursor. *lockType* must be an expression that evaluates to one of the following string values: Write, Read, Full, or Any. If successful, this method returns True; otherwise, it returns False.

Example

The following example opens a TCursor for *Customer* (a Paradox table), places a full lock on the table, then uses **reIndex** to rebuild the *Phone_Zip* index. Once the index is rebuilt, this code unlocks *Customer* so other users on a network can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if tc.open(pdoxTbl) then
  if tc.lock("Full") then    ; attempt to gain exclusive access
    tc.reIndex("Phone Zip")  ; rebuild Phone Zip index
    tc.unLock("Full")        ; unlock the table
```

```
  else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
  endif
else
  msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endif
endmethod
```

See also

- lock, lockStatus

unlockRecord

Beginner

TCursor

Method

Unlocks the current record.

Syntax

unlockRecord ( ) Logical

Description

Unlocks the current record if it is locked. If you try to unlock a record that isn't locked, you'll get an error. This operation fails if the current record cannot be committed (for example, because of a key violation).

If the table is indexed, the record is posted to the table and moved to its sorted position. Depending on whether the record moved to a new position, the TCursor may not continue to point to the posted record. This behavior is referred to as record *flyAway.*

You can use the **setFlyAwayControl** method to control the behavior of **unlockRecord** and record flyaway. If **setFlyAwayControl** is set to True, the TCursor will continue to point to the posted record after a call to **unlockRecord**. You can also use the **didFlyAway** method to test whether the record did, in fact, fly away. Refer to the entry for **didFlyAway** method for more information regarding record flyaway and the **unlockRecord** method.

Example

In the following example, the **pushButton** method for *thisButton* attempts to locate a misspelled value in the *Name* field of the *Customer* table. If the value is found, this code locks the record, corrects the value in the field, then unlocks the record with **unLockRecord**.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
endVar
if tc.open("Customer.db") then
  if tc.locate("Name", "Usco") then
    tc.edit()
    tc.lockRecord()          ; lock current record
    tc.Name = "Unisco"       ; change field value
    tc.unlockRecord()        ; unlock current record
```

```
    message("Name changed to \"Unisco\"")
  else
    msgStop("Sorry", "Can't find \"Usco\" in \"Name\" field.")
  endif
else
  msgStop("Sorry", "Can't open Customer.db table.")
endif

endmethod
```

See also

- ❐ didFlyAway, lockRecord, postRecord, setFlyAwayControl

updateRecord

TCursor

Beginner

Method

Updates the existing record with data from the new record when a key violation exists.

Syntax

updateRecord ([const ***moveTo*** Logical]) Logical

Description

Overwrites the existing record with values from the unposted new record when a key violation exists. The record is posted to the table and does not remain locked. If optional argument *moveTo* is True, the TCursor will point to the record after it is posted to the table; if False, the TCursor points to the record following the position of the original record.

Example

See the example for attachToKeyViol.

See also

- ❐ attachToKeyViol, lockRecord, postRecord

TextStream

TextStream

advMatch
close
commit
create
end
eof
home
open
position
readChars
readLine
setPosition
size
writeLine
writeString

A TextStream is a sequence of characters read from (or written to) a text file. TextStreams contain only ANSI characters; formatting information such as font, alignment, and margins is not included. However, nonprinting characters, such as carriage returns and line feeds (CR/LF) are included.

Paradox maintains a file position pointer that behaves like an insertion point cursor in a word processor. The pointer tells you how far (how many characters) you are from the beginning of the file. Counting begins with 1 (not with 0, as in some other languages).

advMatch

TextStream

Method

Searches for a pattern of characters in a text file.

Syntax

advMatch (var ***startIndex*** LongInt, var ***endIndex*** LongInt, const ***pattern*** String) Logical

Description

Searches a text file for a pattern of characters represented by the variable *pattern*. If *startIndex* is assigned a value, the search starts at the startIndex position; otherwise, the search starts at the beginning of the file. The position in *endIndex* does *not* indicate the end of the range to search. If the pattern is found, the position of the first matching character is stored in *startIndex*, and the position of the last matching character is stored in *endIndex*.

advMatch returns True if *pattern* is found in the file; otherwise, it returns False. This method is case sensitive by default, but you can use the System procedure **ignoreCaseInStringCompares** to change the case behavior.

If you supply *pattern* from within a method, you need to use two backslashes when you want to tell **advMatch** to treat a special character as a literal; for example, **\\(** tells **advMatch** to treat the parenthesis as a literal character. Two backslashes are required in this situation because of the ambiguity between the compiler's interpretation of a backslash (used in escape sequences such as "\t" for a tab) and **advMatch**'s understanding of a backslash. When the compiler sees a string with an embedded escape sequence, such as a "\tstart", it interprets the "\t" as a tab, followed by the word "start". The backslash character has a special meaning to the compiler, but it also has a special meaning to **advMatch**. (See the entry for **advMatch** in the String type.)

If you supply *pattern* from a field in a table or a TextStream, special **advMatch** symbols are recognized without a preceding backslash, and one backslash and plus symbol (\+) yields a literal character.

To specify *pattern*, use a string with the optional symbols listed in the table below.

Symbol	Matches
\	Use backslash to include any of the above as regular characters. (Remember to use two backslashes in quoted strings.)
[ ]	Match the enclosed set. For instance, [aeiou0-9] match a, e, i, o, u, and 0 through 9.
[^]	Do *not* match the enclosed set. For instance, [^aeiou0-9] matches anything except a, e, i, o, u, and 0 through 9.
()	Grouping.
^	Beginning of line (do not confuse this with [^], where the ^^ acts as a logical NOT).
$	End of string.
..	Match anything.
@	Match any single character.
*	Zero or more of the preceding character or expression.
+	One or more of the preceding character or expression.
?	None or one of the preceding character or expression.
|	OR operation.

Example

This example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The call to **advMatch** specifies "@o@e" as the pattern to search. This pattern matches any character followed by an **o** followed by any character followed by an **e**. When this pattern is found, the variables *firstChar* and *lastChar* store the positions of the first and last matching characters. The calls to **setPosition** and **readChars** read the matching characters and store them in the variable *theMatch*.

```
; findSome::pushButton
method pushButton(var eventInfo Event)
var
  pdq                 TextStream
  firstChar, lastChar LongInt
  theMatch            String
endvar
if pdq.open("pdxquote.txt", "R") then
  if pdq.advMatch(firstChar, lastChar, "@o@e") then
    msgInfo("The position found", firstChar); 59
    pdq.setPosition(firstChar)
    pdq.readChars(theMatch, lastChar - firstChar)
    message(theMatch)               ; displays "some"
  else
    msgInfo("Sorry", "Match not found.")
  endIf
  pdq.close()
else
  msgInfo("Sorry", "Couldn't open the requested text file.")
endif
endmethod
```

See also

- end, home, position, setPosition
- advMatch in the String type

close

TextStream

Method

Closes a text file.

Syntax

close ( ) Logical

Description

Closes a text file and writes the contents of all text buffers to disk. It also ends the association between a TextStream variable and the underlying text file.

Example

This example declares one TextStream variable, *ts*, and calls **open** to associate *ts* with the text file PDXQUOTE.TXT, then calls **close** to end the association.

```
; quoteALine::pushButton
method pushButton(var eventInfo Event)
```

```
var
  ts        TextStream
  firstLine String
endvar
ts.open("pdxQuote.txt", "R")
ts.readLine(firstLine)
firstLine.view("Line 1 of PDXQUOTE.TXT")
ts.close()
endmethod
```

See also

- commit, open

commit

TextStream

Method

Writes the contents of the text buffer to disk.

Syntax

commit ( )

Description

Empties the text buffer and writes the contents to disk. The file stays open and the position of the file pointer does not change.

Example

In this example, the *createText* button creates a new file called MYTEXT.TXT, writes a line to it, commits the current version of the TextStream, then closes the file.

```
; createText::pushButton
method pushButton(var eventInfo Event)
var
  ts TextStream
endVar

ts.create("myText.txt")
msgInfo("TextStream position is now", ts.position()) ; displays 1

ts.writeLine("This is some text.")
msgInfo("TextStream position is now", ts.position()) ; displays 21

ts.commit()
msgInfo("TextStream position is now", ts.position()) ; still 21

endmethod
```

See also

- writeString

create

TextStream

Method

Creates a text file.

Syntax

create (const ***fileName*** String) Logical

Description

Creates the text file *fileName* and opens it for reading and writing. If *fileName* exists, **create** overwrites it without prompting for confirmation. You can specify a directory in which to create the file using a full DOS path or an alias. If you don't specify a path or an alias, Paradox creates the file in the working directory (:WORK:).

This method returns True if successful; otherwise, it returns False. If the file is successfully created, it is opened for reading and writing.

Note The following statments are equivalent:

```
ts.create("newText.txt")
ts.open("newText.txt, "NW")
```

Example

The following code is attached to a button's **pushButton** method. It consists of a variable declaration block, a procedure declaration, and the body of the method. In the body of the method, the call to the FileSystem method **findFirst** checks for the existence of a file named RICK.TXT. If it doesn't exist, the custom procedure **addLine** creates it and adds a line to it. If the file does exist, a dialog box confirms the decision to overwrite the file.

```
; createFile::pushButton
var
  ts             TextStream
  firstLine      String
  allLines Array[] String
  fs             FileSystem
endvar

proc addLine()
  ts.create(":PRIV:rick.txt")  ; creates file, open for writing and reading
  ts.writeLine("Here's looking at you, kid.")
  ts.home()
  ts.readLine(allLines)
  allLines.view("Rick says:")
endProc

method pushButton(var eventInfo Event)
if not fs.findFirst(":PRIV:rick.txt") then
  addLine()
else
  if msgYesNoCancel(":PRIV:RICK.TXT", "Overwrite this file?") = "Yes" then
    addLine()
  endif
endif
endmethod
```

See also

❐ close, open

TextStream

end

TextStream

Method

Sets the file pointer to the end of a text file.

Syntax

end ( )

Description

Sets the file pointer to the last character of a text file.

Example

This example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The code in this example is attached to the built-in **newValue** method of a field object displayed as two radio buttons. The values of the radio buttons are "Overwrite" and "Append." Choose one to specify whether to insert text at the beginning of the file (which overwrites existing text) or append it to the end of the file. If you choose "Overwrite," the call to **home** moves the pointer to position 1. If you choose "Append," the call to **end** moves the pointer to position 103 (the end of this particular file).

```
; insertAppendField::changeValue
method newValue(var eventInfo Event)
var
  ts TextStream
  allLines Array[] String
endVar
if eventInfo.reason() = EditValue then
  ts.open(":PRIV:pdxquote.txt", "W")
  switch
    case self.value = "Overwrite" :
      ts.home()
      ts.writeLine(DateTime())   ; time stamp the file at beginning
      ; file will read:
      ; DateTimeStamp (depends on date/time)
      ; have met with Paradox.
      ; Now we have some hope of making progress.
      ; Niels Bohr
    case self.value = "Append" :
      ts.end()
      ts.writeLine(DateTime())   ; time stamp the file at end
      ; file will read:
      ; How wonderful that we have met with Paradox.
      ; Now we have some hope of making progress.
      ; Niels Bohr
      ; DateTimeStamp (depends on date/time)
  endSwitch
  ts.home()
  ts.readLine(allLines)
  allLines.view()
  ts.close()
endif
endmethod
```

See also ❒ eof, home, setPosition

eof

TextStream

Method

Tests for a move past the end of a text file.

Syntax

eof ( ) Logical

Description

Returns True if an operation tries to move the file pointer past the end of a text file; otherwise, it returns False.

Example

This example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The **while** loop reads each of the three lines from the file and displays it in a dialog box. Then **eof** returns True, and a dialog box tells the user that there's no more text in the file.

```
; lineAtATime::pushButton
method pushButton(var eventInfo Event)
var
  pdq      TextStream
  textLine String
endVar

pdq.open(":PRIV:pdxquote.txt", "r")
while not pdq.eof()        ; quit loop when you hit the end of the file
  pdq.readLine(textLine)   ; read the next line
  msgInfo("Position " + String(pdq.position()), textLine)
endWhile
msgInfo("Finished", "No more text")
endmethod
```

See also

❒ end, position, setPosition

home

TextStream

Method

Sets the pointer to the beginning of a text file.

Syntax

home( )

Description

Sets the file pointer to the first character of a text file.

Example See the example for end.

See also

- end, eof, setPosition

open

TextStream

Method Opens a text file in a specified mode.

Syntax

open (const ***fileName*** String, const ***mode*** String) Logical

Description

Opens *fileName* in the mode specified in *mode*, and associates a FileSystem variable with the underlying file. The modes are

Mode	Rights
a	Append/Read
r	Read Only
w	Write/Read
nw	New/Write/Read

If the file exists, the New/Write/Read mode overwrites the file without asking for confirmation.

Note The following statements are equivalent:

```
ts.open("new.txt", "NW")
ts.create("new.txt")
```

Opening a file in any mode *except* Append/Read sets the pointer to the beginning of the file.

You can specify a directory from which to open the file using a full DOS path or an alias. If you don't specify a path or an alias, Paradox looks for the file in the working directory.

This method returns True if successful; otherwise, it returns False.

Example

This example uses an alias with **open** to create a text file in the private directory and write a line of text to it:

```
var
   ts TextStream
endVar
if ts.open(":PRIV:memo14.txt", "NW") then
   ts.writeLine("This is private!")
endIf
```

You can associate more than one TextStream variable with the same file. Both variables have equal rights to the file, and Paradox maintains separate pointers for each variable. The following example

declares two TextStream variables, *ts1* and *ts2*, and calls **open** to associate each of them with the text file NEWTEXT.TXT. As statements are written to the file, messages display the pointer position for each variable.

```
; openStreams::pushButton
method pushButton(var eventInfo Event)
var
  tsl, ts2  TextStream
  firstLine String
  allLines  Array[] String
endvar
tsl.open("newText.txt", "nw")      ; open a new file Write/Read
tsl.writeLine("Written by tsl.")
tsl.writeLine("This is line 2.")
msgInfo("Text stream one", tsl.position()) ; displays 35
tsl.commit()                       ; write it out to disk, so that
                                   ; ts2 will get most current version

ts2.open("newText.txt", "w")       ; open existing file Write/Read
msgInfo("Text stream one", tsl.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 1

ts2.writeLine("Written by ts2.")
msgInfo("Text stream one", tsl.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 18

tsl.home()
tsl.readLine(allLines)       ; reads all lines into an array
allLines.view("tsl")         ; displays:
                             ; Written by ts2.
                             ; This is line 2.

endmethod
```

See also

- close, create

position

TextStream

Method

Returns the pointer's position in a text file.

Syntax

position ( ) LongInt

Description

Returns an integer representing the pointer's position in a text file. **position** counts both printing and nonprinting characters. Counting begins with 1 (not with 0).

Example

It may be helpful to think of **position** as returning the number of the next character in the file. As this example shows, when you create a new text file and call **position**, it returns 1. The call to **writeLine** adds 14 characters to the file: 12 printing characters and the carriage return

and line feed (CR/LF) pair. The next character will be 15, so **position** returns 15.

```
var newFile TextStream endVar
newFile.open("newmemo.txt", "nw")
message(newFile.position()) ; displays 1
sleep(1000)
newFile.writeLine("Don't panic.")
message(newFile.position()) ; displays 15
                            ; 12 printing characters + CR/LF = 14
                            ; next character will be 15
sleep(1000)
```

See also

❐ setPosition, size

readChars

TextStream

Method

Reads a specified number of characters from a text file.

Syntax

readChars (var ***string*** String, const ***nChars*** SmallInt) Logical

Description

Reads the number of characters specified in *nChars* and stores them in *string*. **readChars** starts reading from the current pointer position. This method returns True if successful; otherwise, it returns False.

Example

This example assumes that a file named PDXQUOTE.TXT exists in the current working directory. The file contains the following text:

```
How wonderful that we have met with paradox.
Now we have some hope of making progress.
Niels Bohr
```

The call to **readChars** reads the first 100 characters from the file:

```
; getLetters::pushButton
method pushButton(var eventInfo Event)
var
  letter  TextStream
  myChars String
endVar
letter.open("pdxquote.txt", "r")
if letter.readChars(myChars, 100) then
  msgInfo("The first 100 characters are:", myChars)
endIf
endmethod
```

See also

❐ readLine, setPosition

❐ The example for advMatch

readLine

TextStream

Method

Reads a line or lines from a text file.

Syntax

readLine (var ***value*** String) Logical

readLine (var ***stringArray*** Array[] String) Logical

Description

Reads characters from a line of text from a file until a carriage return and line feed pair (CR/LF) is encountered, and moves the file pointer to the first position after the CR/LF pair. **readLine** begins reading from the current pointer position.

You can store a single line in *value*, or store the entire file in *stringArray*, where *stringArray* is a resizable array of strings and each array item stores one line from the file. In either case, the CR/LF pair is not stored.

This method returns True if successful; otherwise, it returns False.

Example

The first example creates a 2-line text file, then calls **readLine** to read the first line into a String variable. **readLine** reads the four characters before the CR/LF in the first line, then skips over the CR/LF characters, and sets the pointer.

```
method pushButton(var eventInfo Event)
var
   ts TextStream
   oneLine String
endvar

ts.create("newtext.txt")
ts.writeLine("1234")
ts.writeline("5678")
ts.home()

ts.readLine(oneLine)
message(oneLine.size()) ; displays 4 (doesn't include CR/LF)
sleep(1234)
message(ts.position()) ; displays 7 (skips over CR/LF)
sleep(1234)
endmethod
```

The next example creates a 3-line text file, then calls **readLine** to read the entire file into an array, then displays the array in a dialog box.

```
var
   letter TextStream
   allLines Array[] String
endVar

letter.open("letter.txt", "nw")
letter.writeLine("Dear Customer,")
letter.writeLine("Thank you for your interest in our new product.")
letter.writeLine("A representative will call you next week.")
```

```
letter.home() ; move the pointer to the beginning of the file

letter.readLine(allLines)
allLines.view("Entire letter")     ; displays the entire letter
letter.close()
```

See also

- readChars, writeLine

setPosition

TextStream

Method

Positions the pointer in a text file.

Syntax

setPosition (const ***offset*** LongInt)

Description

Positions the file pointer *offset* characters from the beginning of a text file. Carriage return and line feed characters are considered part of the file, and can be overwritten. Specifying a position before the beginning or after the end of the file moves the pointer to the beginning or end of the file.

Example

In this example, the *showPositions* button first writes a line to a new text file, MEMO.TXT. The method then moves back to the fourth character, overwrites that character with "4", then rereads and displays the line.

```
; showPositions::pushButton
method pushButton(var eventInfo Event)
var
  myFile  TextStream
  lineOne String
endVar
myFile.open(":PRIV:memo.txt", "nw")        ; open new file as read/write
myFile.writeLine("1235")                   ; 4 characters plus CR/LF
msgInfo("Where am I?", myFile.position())  ; displays 7

myFile.setPosition(4)                 ; move to character 4
myFile.writeString("4")               ; now, line is "1234"
myFile.home()                         ; same as setPosition(1)
myFile.readLine(lineOne)
msgInfo("This is line one", lineOne) ; displays  "1234"
endmethod
```

This example shows what happens when you attempt to move the pointer beyond the end of a file or before the beginning of a file.

```
; showPositions::pushButton
method pushButton(var eventInfo Event)
var
  myFile  TextStream
endVar

myFile.open(":PRIV:memo.txt", "r")  ; open existing file for read
myFile.setPosition(100)             ; beyond end of file
```

```
msgInfo("End", myFile.position())    ; displays 7 -- the real end
myFile.setPosition(-100)             ; before beginning of file
msgInfo("Home", myFile.position())   ; displays 1 -- the beginning
endmethod
```

See also

- eof, position, size

size

TextStream

Method

Returns the number of characters in a text file.

Syntax

size ( ) LongInt

Description

Returns the number of characters in a text file, including nonprinting characters such as carriage returns and line feeds (CR/LF).

Example

This example creates a TextStream, writes a line to it, then shows the size of the file.

```
; showSize::pushButton
method pushButton(var eventInfo Event)
var
  myText TextStream
endVar
myText.create("short.txt")
myText.writeLine("1234")
msgInfo("What size am I?", myText.size()) ; displays 6
; 4 printing characters "1234", and 2 nonprinting characters CR/LF
myText.close()
endmethod
```

See also

- eof, position, setPosition

writeLine

TextStream

Method

Writes a string to a text file.

Syntax

writeLine (const ***value*** AnyType [, const ***value*** AnyType]*) Logical

Description

Writes a comma-separated list of *value*s to a text file, and appends a carriage return and line feed character pair (CR/LF). Compare this method to **writeString**, which doesn't append a CR/LF pair.

Example

See the example for create.

See also ❐ readLine, writeString

writeString

TextStream

Method

Writes a character string to a text file.

Syntax

writeString (const ***value*** AnyType [, const ***value*** AnyType]*) Logical

Description

Writes a comma-separated list of *values* to a text file, but does not append a carriage return and line feed pair (CR/LF). Compare this method to **writeLine**, which does append a CR/LF pair.

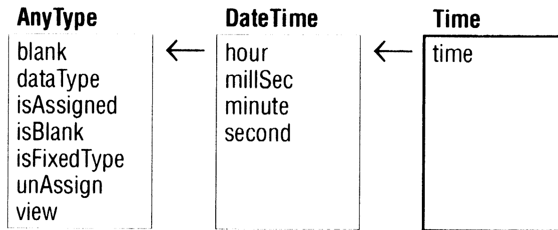
Example

The following example assigns strings to the variables *lo* and *hi,* then uses **writeString** to write them to an open TextStream.

```
; goodAdvice::pushButton
method pushButton(var eventInfo Event)
var
   myText TextStream
   lo, hi String
endVar
lo = "Buy low. "
hi = "Sell high."
myText.open(":PRIV:advice.txt", "nw")        ; open a new file
myText.writeString(lo, hi)
msgInfo("File size:", string(myText.size()))  ; displays 19
; Buy low.  = 9, Sell High. = 10
mytext.close()
endmethod
```

See also ❐ writeLine

Time



Time variables store times in the form hour-minute-second-millisecond. You can use the following characters as separators: blank, tab, space, comma (,), hyphen (-), slash (/), period (.), colon (:), and semicolon (;).

Time values must be cast (explicitly declared). For example, the following statements assign to the Time variable *ti* a time of 10 minutes and 40 seconds past eleven o'clock in the morning:

```
var ti Time endVar
ti = Time("11:10:40 am")
```

The quotes around the value are required. Whether a time is valid depends on the current Paradox time format. For example, if the current time format is set to 12-hour format (such as hh:mm:ss), methods in the Time type consider hh:mm:ss a valid time format. Use **formatSetTimeDefault** and **formatSetDateTimeDefault** from the System type to set Paradox time formats with ObjectPAL.

The Time type includes several methods defined for the AnyType and DateTime types. See Chapter 9 in the *ObjectPAL Developer's Guide* for more information and examples.

time

Time

Beginner

Procedure

Casts a value as a time, or returns the current time.

Syntax

time ([const ***value*** AnyType]) Time

Description

Casts (converts) *value* as a time, or returns the current time according to the system clock. *value*, if given, must match the current Paradox time format. For more information, refer to the System type procedure **formatSetTimeDefault**.

Example

This example calls **time** to convert a string value to a time value:

```
var
   st String
   ti Time
endVar

st = "12:21:33 am"
ti = time(st)
```

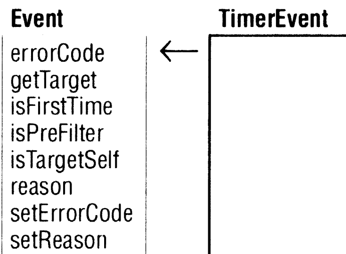
The next example displays the current time in a dialog box. The display format varies according to the user's current time format. This code is attached to a button's **pushButton** method.

```
; timeButton::pushButton
method pushButton(var eventInfo Event)

  ; displays the current time in a dialog box
   msgInfo("Current Time", time())

endmethod
```

See also

- hour, milliSec, minute, second in the DateTime type

TimerEvent



Methods in the TimerEvent type provide information used by the built-in **timer** methods. The TimerEvent type includes several methods defined for the Event type. Refer to the "Event" section for more information. For more information about built-in methods, see Chapter 2.

Use **setTimer**, from the UIObject type, to specify when to send Timer Events to an object, then modify the object's built-in **timer** method to control how the object responds when a timer goes off. Use **killTimer**, defined for the UIObject type, to turn off an object's timer. The following example shows how to use these methods with TimerEvents. For more information and examples, refer to Chapter 6 in the ObjectPAL Developer's Guide.

In this example, assume that a form contains a multi-record object bound to the *Customer* table. The record container in the multi-record object is named *custRecord.*

Suppose you have a data-entry program, and you want to give the user 60 seconds to edit a record. After 60 seconds, you want to alert the user. To accomplish this, the **action** method for *custRecord* tests every action. If the action is DataArriveRecord, the method stops any old timers with **killTimer** and sets a new timer for the record object with **setTimer**. When the timer goes off, a message pops up alerting the user. To make it easy to change the timer interval, a constant is defined in the Const window for *custRecord,* as follows:

```
; custRecord::Const
const
  alertTime = 60000      ; data-entry alert at 60 seconds
endConst
```

The following code is for the **action** method for *custRecord.*

```
; custRecord::action
method action(var eventInfo ActionEvent)
if eventInfo.id() = DataArriveRecord then ; when opening to a new record
  self.killTimer()        ; just in case it hasn't expired
                          ; yet, kill the old timer
  self.setTimer(alertTime) ; start timer for this record
```

```
endif
endmethod
```

This code is attached to the **timer** method for *custRecord*.

```
; custRecord::timer
method timer(var eventInfo TimerEvent)
beep()
msgInfo("Alert", "You have been processing this record for " +
                 "one minute now.")
self.killTimer()
endmethod
```

See the **killTimer**, **setTimer**, and **action** methods from the UIObject type for more information.

UIObject

UIObject

action	home	mouseRightDouble
atFirst	insertAfterRecord	mouseRightDown
atLast	insertBeforeRecord	mouseRightUp
attach	insertRecord	mouseUp
broadcastAction	isContainerValid	moveTo
cancelEdit	isEdit	moveToRecNo
convertPointWithRespectTo	isEmpty	moveToRecord
copyFromArray	isLastMouseClickedValid	nextRecord
copyToArray	isLastMouseRightClickedValid	nFields
create	isRecordDeleted	nKeyFields
currRecord	keyChar	nRecords
delete	keyPhysical	pixelsToTwips
deleteRecord	killTimer	postAction
edit	locate	postRecord
empty	locateNext	priorRecord
end	locateNextPattern	pushButton
endEdit	locatePattern	recordStatus
enumFieldNames	locatePrior	reSync
enumLocks	locatePriorPattern	rgb
enumObjectNames	lockRecord	setFilter
enumSource	lockStatus	setPosition
enumSourceToFile	menuAction	setProperty
enumUIClasses	methodDelete	setTimer
enumUIObjectNames	methodGet	skip
enumUIObjectProperties	methodSet	switchIndex
execMethod	mouseClick	twipsToPixels
getBoundingBox	mouseDouble	undeleteRecord
getPosition	mouseDown	unlockRecord
getProperty	mouseEnter	view
getPropertyAsString	mouseExit	wasLastClicked
getRGB	mouseMove	wasLastRightClicked
hasMouse		

UIObjects (the UI stands for User Interface) create the user interface for an application: anything you can place in a form is a UIObject. Only UIObjects have built-in methods. The different UIObjects are the bitmap, box, button, crosstab, ellipse, field object, form, graph, line, multi-record object, OLE object, page, record object, table frame, and text box.

Note The form behaves like a UIObject: a form has built-in methods, you can attach code to those built-in methods, and a form responds to events. There is also a separate type, Form, for methods and procedures that work only with forms.

Many UIObject methods duplicate TCursor methods. The UIObject methods that work with tables work on the underlying table through

the visible object. Actions directed to the UIObject that affect the table are immediately visible in the object the table is bound to. TCursor methods, by contrast, work with a table behind the scenes; actions that affect the table are not necessarily visible in any object, even if the TCursor is acting on the same table to which a visible object is bound.

Some table operations are considerably faster with TCursors than with UIObjects. For instance, if you need to perform a table-oriented operation that will cause a high volume of screen refreshes—which might be slow—you can use this technique: declare a TCursor, attach it to the object the table is already bound to (such as a table frame), do the operation with the TCursor, then resynchronize the display object to the TCursor. When you attach a TCursor to an object bound to a table, the TCursor's record pointer is set to the current record for the object. After you perform a TCursor operation, such as a **locate**, the TCursor might point to a different record than the object. To make the object point to the same record as the TCursor, use **resync** or **moveToRecord**; to make the TCursor point to the same record as the object, use the **attach** method. See the example for **insertRecord**, later in this section.

For more information about working with UIObjects, refer to Chapter 7 of the *ObjectPAL Developer's Guide.*

action

UIObject

Beginner

Method

Performs a specified action.

Syntax

action (const ***actionID*** SmallInt) Logical

Description

Specifies an *actionID* to perform in response to an event. ObjectPAL provides a constants for *actionID*; see one of the categories beginning with Action (for example, ActionDataCommands), in the Constants dialog box.

Example

The code in this example is attached to a button's **mouseUp** method and does the following: if you press and hold *Shift* and click the button, the pointer moves to the next set of records. If you click the button without pressing *Shift*, the pointer moves to the next record.

The action constants DataFastForward and DataNextRecord behave like the Fast Forward and Next Record SpeedBar buttons. Assume that *CUSTOMER* refers to a table frame on the form and that *nextRecordOrFast* is a button on the same form. The *nextRecordOrFast*

button is not in the same containership hierarchy as *CUSTOMER*, so the action won't bubble up to *CUSTOMER* automatically. Thus, the action must be sent to the *CUSTOMER* object explicitly.

```
; nextRecordOrFast::mouseUp
method mouseUp(var eventInfo MouseEvent)
; if Shift key is down, go to next set of records,
;  otherwise go to next record
if eventInfo.isShiftKeyDown() then
   CUSTOMER.action(DataFastForward)
else
   CUSTOMER.action(DataNextRecord)
endif
endmethod
```

See also

- menuAction
- id, setId in the ActionEvent type
- The discussion of the built-in **action** method in Chapter 2

atFirst

UIObject

Method

Reports if the pointer is at the first record of a table.

Syntax

atFirst ( ) Logical

Description

Returns True if the pointer is at the first record of a table; otherwise, it returns False. **atFirst** respects the limits of restricted views displayed in a linked table frame or multi-record object.

Example

In the following example, assume that *CUSTOMER* refers to a table frame on the form and *goToFirstButton* is a button on the same form. The method checks the pointer position. If the pointer is not on the first record of *CUSTOMER*, the method moves it to that record.

```
; goToFirstButton::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atFirst() then
  CUSTOMER.home()
  ; this has the same effect as:  CUSTOMER.action(DataBegin)
endif
endmethod
```

See also

- atLast

atLast

UIObject

Method

Reports if the pointer is at the last record in a table.

Syntax

atLast ( ) Logical

Description

Returns True if the pointer is at the last record of a table; otherwise, it returns False. **atLast** respects the limits of restricted views displayed in a linked table frame or multi-record object.

Example

In the following example, assume that *CUSTOMER* refers to a table frame on the form and *goToLastButton* is a button on the same form. The method checks the pointer position. If the pointer is not on the last record of *CUSTOMER*, the method moves it to that record.

```
; goToLastButton::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atLast() then
  CUSTOMER.end()
  ;this has the same effect as:  CUSTOMER.action(DataEnd)
endif
endmethod
```

See also

- atFirst

attach

UIObject

Method

Binds a UIObject variable to a specified design object.

Syntax

1. **attach** ( ) Logical
2. **attach** (const ***object*** UIObject) Logical
3. **attach** (const ***form*** Form [, const ***objectName*** String]) Logical
4. **attach** (const ***form*** Report [, const ***objectName*** String]) Logical

Description

Binds a UIObject variable to self (syntax 1), to another UIObject (*object* in syntax 2), to a Form (*form* in syntax 3), or to a UIObject in another Form (*objectName* in syntax 3). You can also use attach to bind a UIObject to an open report, or to an object in an open report (syntax 4).

Note Some of the methods in the UIObject type can be used for forms, but only if you attach a UIObject variable to the form. Syntax 3 of the **attach** method lets you attach a UIObject variable to a form so that you can access those methods. For instance, to send a **mouseUp**

event to another form's form-level **mouseUp** built-in method, you need to attach a UIObject (a Form variable won't work) to an open form.

Example

The following example shows all three forms of the syntax. For syntax 1, the method attaches the variable *objBox* to the current object (self), then changes its color. For syntax 2, the method attaches *objBox* to another object, then changes that object's color via *objBox*. A second example for syntax 2 opens another form, attaches *objBox* to a box on the second form, and changes the color of the other form's object via *objBox*.

In this example, assume the current form contains two boxes, *thisBox* and *thatBox*. The method is attached to *thisBox*. The secondary form contains one box, called *otherBox*.

```
; thisBox::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  objBox,
  objForm    UIObject
  otherForm  Form
endVar

; syntax 1
objBox.attach()          ; binds objBox to thisBox
objBox.color = DarkMagenta

; syntax 2
objBox.attach(thatBox)    ; binds objBox to thatBox
objBox.color = Magenta

; assume the form uiattch2.fsl exists and it has
; one object called otherBox
if otherForm.open("uiattch2.fsl") then
  objBox.attach(otherForm.otherBox)
  objBox.color = DarkBlue
  sleep(2000)
  otherForm.close()
endif

; syntax 3
if otherForm.open("uiattch2.fsl") then
  ; notice that the object name is given as a string
  objBox.attach(otherForm, "otherBox")
  objBox.color = LightBlue
  sleep(2000)
  otherForm.close()
endif

endmethod
```

See also

- moveToRecord

UIObject

broadcastAction

UIObject

Method

Broadcasts an action to an object.

Syntax

broadcastAction (const ***actionID*** SmallInt)

Description

Notifies an object and all of the objects contained by that object that an event has occurred.

See also

- action, menuAction

cancelEdit

Beginner

UIObject

Method

Cancels record changes without ending Edit mode.

Syntax

cancelEdit ( ) Logical

Description

Leaves a table in Edit mode but cancels changes to the current record. It returns True if successful; otherwise, it returns False. To abort changes to the current record, you must use **cancelEdit** before moving the pointer from the current record; once you move the pointer, changes to the record are committed.

cancelEdit has the same effect as the action constant DataCancelEdit, so the following statements are equivalent:

```
obj.cancelEdit()
```

```
obj.action(DataCancelEdit)
```

Example

The following method attaches a UIObject variable, *noChange*, to a table frame, *CUSTOMER*. (From then on *noChange* is used as a handle to the table frame.) The method searches for a value in the *Customer* table, and, if found, changes the value. Before leaving the record, the change is cancelled with the **cancelEdit** method. In this example, assume that you have one page on the form, called *pageOne*; a table frame attached to the *Customer* table; and a button named *CancelEditButton*.

```
; CancelEditButton::pushButton
method pushButton(var eventInfo Event)
var
  noChange UIObject
endVar

noChange.attach()
```

```
noChange.attach(pageOne.CUSTOMER)
noChange.edit()
if noChange.locate("Name", "Unisco") then
  noChange."Name" = "Jones"  ; prepare to change the record
  msgInfo("noChange.'Name'", noChange."Name".value)
  noChange.cancelEdit()       ; belay that order!
                             ; record not changed.
endif
noChange.endEdit()           ; exit Edit mode

endmethod
```

See also

- currRecord, edit, endEdit

convertPointWithRespectTo

UIObject

Method

Changes the frame of reference for calculating the coordinates of a point.

Syntax

convertPointWithRespectTo **(** const ***otherUIObject*** UIObject, const ***oldPoint*** **Point,** var ***convertedPoint*** Point **)**

Description

Changes the frame of reference for calculating the position of a point. Normally, coordinates are calculated relative to the upper left corner of the object's container (or the container's frame, in the case of an ellipse). This method instead calculates a point's position relative to the upper left corner of the object specified in *otherUIObject*.

Example

This example gets and shows the position of an object called *innerBox*. *innerBox* is contained by *outerBox*, and is on a page called *pageOne*. First, the position of *outerBox* relative to the upper left corner of the page is obtained and displayed. Next, the position of *innerBox* is taken, relative to the upper left corner of *outerBox*. Finally, the position of *innerBox* is converted with respect to the page, so you can see how far *innerBox* is from the top and left edges of the page.

```
; alignInnerBox::pushButton
method pushButton(var eventInfo Event)
var
  innerPos,
  outerPos,
  convertedPos  Point
  x, y, w, h    LongInt
  innerObj      UIObject
endVar

outerBox.getPosition(x, y, w, h)
outerPos = point(x, y) ; convert x and y from
outerPos.view("Outer box position") ; outerBox to a point
;outerBox.innerBox.getPosition(x, y, w, h)
innerObj.attach("outerBox.innerBox")
innerObj.getPosition(x, y, w, h)
```

```
innerPos = point(x, y)
innerPos.view("Inner box position unconverted")
; how far is innerPos from the upper left corner of the page?
innerObj.convertPointWithRespectTo(pageOne, innerPos, convertedPos)
convertedPos.view("Inner box position converted")
endmethod
```

See also

- The description of the Point type in Chapter 9 of the *ObjectPAL Developer's Guide*

copyFromArray

UIObject

Method

Copies data from an array to a record of a table.

Syntax

copyFromArray (const ***ar*** Array[] AnyType) Logical

Description

Copies data from an array *ar* to a UIObject (typically a table frame or multi-record object). The first element of the array is copied to the first field of the table, the second element to the second field, and so on until the array is exhausted or the record is full.

The method fails if an attempt is made to copy an unassigned array element or if the structures do not match. (This can never happen if the array was created by **copyToArray**, which assigns a blank value if a field is blank.) In addition, the method fails if the form is not in Edit mode. If there are more elements in the array than fields in the record, the extra elements are ignored.

Example

In this example, suppose a form contains a table frame named *CUSTNAME*. The *CUSTNAME* table has three fields: Last name, A20; First name, A20; and Middle Initial, A1. This method starts editing *CUSTNAME*, creates an array with three elements, creates a new record in *CUSTNAME*, then copies data from the array to the record.

```
; createRecord::pushButton
method pushButton(var eventInfo Event)
var
  nameArray Array[3] String
endvar
CUSTNAME.edit()            ; start Edit mode
nameArray[1] = "Hall"      ; fill the array with the record to insert
nameArray[2] = "Robert"
nameArray[3] = "A"
CUSTNAME.action(DataInsertRecord) ; insert a blank record first
CUSTNAME.copyFromArray(nameArray) ; then copy the array into the new record
CUSTNAME.endEdit()
endmethod
```

See also

- copyToArray

copyToArray

UIObject

Method

Copies data from a record to an array.

Syntax

copyToArray (var ***ar*** Array [] AnyType) Logical

Description

Copies the fields of the current record of a UIObject (typically a table frame or a multi-record object) to the elements of an array specified in *ar*. You must declare the array to be of type AnyType, or of a type that matches every field in the table. If the array is resizeable, it grows automatically to hold the number of fields in the record. If the array is not resizeable, it holds as many fields as it can, and the rest are discarded.

The value of the first field is copied to the first element of the array, the value of the second field to the second element, and so on. The size of the array is equal to the number of fields in the record. The record number field and any display-only or calculated fields that appear in a form view of the table are not copied to the array.

Example

The following example assumes that there are two table frames on a form, *CUSTOMER* and *CUSTARC*, and one button, named *archiveButton*. The form itself is renamed *thisForm*. When *archiveButton* is pushed, the current record in *CUSTOMER* is moved to *CUSTARC*.

First, the method looks at the Editing property of the form; if it's False, the method starts Edit mode. The method then copies the current record in *CUSTOMER* to the *arcRecord* array and attempts to delete the current record. If the current record can't be locked and deleted, the record is not copied to the target table *CUSTARC*. If the record delete is successful, the method adds a new blank record to the target table, and writes the contents of the array to the record.

```
; archiveButton::pushButton
method pushButton(var eventInfo Event)
var
  arcRecord Array[] String
endVar

; check to see if form is in edit mode
if thisForm.Editing = False then ; if not, then start
  CUSTOMER.action(DataBeginEdit)
endif

; move the current record from CUSTOMER to archive in CUSTARC
CUSTOMER.copyToArray(arcRecord)
arcRecord.view()          ; take a look at the array
; if the record can't be locked, it won't be deleted
if CUSTOMER.deleteRecord() = True then
  ; if it is deleted, then copy it to the archive table
  CUSTARC.insertRecord()          ; insert blank record
  CUSTARC.copyFromArray(arcRecord)  ; copy array to blank record
```

UIObject

```
endif

endmethod
```

See also ❒ copyFromArray

create

UIObject

Method Creates an object.

Syntax

create (const ***objectType*** SmallInt, const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt [, const ***container*** UIObject])

Description

Creates the object specified in *objectType* at a position specified in *x* and *y*, with a width specified in *w* and a height specified in *h*. (*x*, *y*, *w*, and *h* are assumed to be twips.) The optional argument *container* specifes a container object for the object you're creating.

ObjectPAL provides constants (like ButtonTool) for objectType; see UIObjectTypes in the Constants dialog box.

Note When you use **create** to create an object for running a form or report, the object is invisible. To make it visible, set its Visible property to True. Objects can also be deleted at run time with the **delete** method.

Example

The following code is attached to the **mouseUp** method for *pageOne* on a form. This example creates a box, names it *Fred*, colors it blue, and sets it visible. An ellipse is then created with a size position in *Fred*, and its container is set to *Fred*.

```
; pageOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  ui UIObject
endvar

; create a Blue box, called Fred, and make it visible
ui.create(BoxTool, 144, 144, 2880, 2880)
ui.Name = "Fred"
ui.Color = Blue
ui.Visible = True
; create a Green ellipse inside Fred, called Bill
ui.create(EllipseTool, 288, 288, 1440, 1440, self.Fred)
ui.Name = "Bill"
ui.Color = Green
ui.Visible = True

endmethod
```

See also ❒ delete, methodSet

currRecord

UIObject

Method

Reads the current record into the record buffer.

Syntax

currRecord ( ) Logical

Description

Cancels changes to the current record, refreshing the current record from saved data. Any changes to the record are not committed. **currRecord** leaves a locked record locked. It returns True if successful; otherwise, it returns False.

currRecord has the same effect as the action constant DataRefresh, so the following statements are equivalent:

```
obj.currRecord()

obj.action(DataRefresh)
```

Example

In this example, assume that a form contains a table frame bound to *Orders*.

```
; refreshRecord::pushButton
method pushButton(var eventInfo Event)
ORDERS.edit()              ; start edit
ORDERS.Amount Paid = 321.45   ; make a change
message("Watch closely now.")
sleep(2000)
ORDERS.currRecord()        ; refreshes record from disk,
                           ; any changes are lost, record
                           ; is not locked
if ORDERS.recordStatus("Locked") then
  msgInfo("FYI", "The record is still locked.")
endif
endmethod
```

See also

❐ cancelEdit

delete

UIObject

Method

Deletes an object from a form.

Syntax

delete ( )

Description

Deletes an object from a form at run time.

Example

In the following example, assume that a form contains a method that creates a box named *Fred* and a ellipse inside *Fred* called *Bill*. These objects are created at run time and thus can't be referenced directly

by this method, because they don't exist yet. The technique used here attaches to the object using a string evaluated at run time. See the **create** example for details about the **mouseUp** method (on the same form) that creates the objects to be deleted.

```
; pageOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  ui   UIObject
endVar

;  Fred and Bill are objects created by the mouseUp method
;  for pageOne of this form. Because they are created at
;  run time, you can't directly refer to them as objects in
;  code. Consequently, attach is used to attach the ui var
;  to the string "Fred.Bill", which is evaluated at run time.
;  As long as mouseUp is called before mouseRightUp, those
;  objects will exist.
if ui.attach("Fred.Bill") then
  ui.delete()
  ui.attach("Fred")
  ui.delete()
  {This would do the same thing as previous four lines,
    because Fred contains Bill at run time:
    ui.attach("Fred")
    ui.delete()
  }
endif
endmethod
```

See also

- create, methodSet

deleteRecord

UIObject

Beginner

Method

Deletes the current record from the table.

Syntax

deleteRecord ( ) Logical

Description

Deletes the current record of a table without prompting for confirmation. It returns True if successful; otherwise, it returns False. This operation cannot be undone.

deleteRecord the same effect as the action constant DataDeleteRecord, so the following statements are equivalent:

```
obj.deleteRecord()
```

```
obj.action(DataDeleteRecord)
```

Example

The following example assumes that there are two table frames on a form, *CUSTOMER* and *CUSTARC*, and one button, named

archiveButton. The form is renamed *thisForm*. When *archiveButton* is pushed, the current record in *CUSTOMER* is moved to *CUSTARC*.

First, the method looks at the Editing property of the form; if Editing is False, the method starts Edit mode. The method then copies the current record in *CUSTOMER* to the *arcRecord* array and attempts to delete the current record. If the current record can't be locked and deleted, the record is not copied to the target table *CUSTARC*. If the record delete is successful, the method adds a new blank record to the target table, and writes the contents of the array to the record.

```
; archiveButton::pushButton
method pushButton(var eventInfo Event)
var
  arcRecord Array[] String
endVar

; check to see if form is in edit mode
if thisForm.editing = False then ; if not, then start
  CUSTOMER.action(DataBeginEdit)
endif

; move the current record from CUSTOMER to archive in CUSTARC
CUSTOMER.copyToArray(arcRecord)
arcRecord.view()          ; take a look at the array
; if the record can't be locked, it won't be deleted
if CUSTOMER.deleteRecord() = True then
  ; if it is deleted, then copy it to the archive table
  CUSTARC.insertRecord()
  CUSTARC.copyFromArray(arcRecord)
endif

endmethod
```

See also

- empty, insertAfterRecord, insertBeforeRecord, insertRecord

---

edit

UIObject

Beginner

Method

Puts a table into Edit mode.

Syntax

edit ( ) Logical

Description

Puts all tables on a form into Edit mode so changes can be made. If a form is already in Edit mode, an unnecessary **edit** does not cause an error, and does not toggle out of Edit mode.

In Edit mode, record changes are posted when the focus moves off the record, when the table receives a DataPostRecord or DataUnlockRecord action, or when **endEdit** is executed. Use **cancelEdit** if you need to cancel changes to the record before departing from the record.

endEdit has the same effect as the action constant DataEndEdit, so the following statements are equivalent:

```
obj.endEdit()

obj.action(DataEndEdit)
```

Example

In this example, assume that a form contains a table frame bound to the *Orders* table, and one button, named *changeDate*. The **pushButton** method for *changeDate* checks the *Sale_Date* and *Ship_Date* fields of the current record, and updates *Sale_Date* if *Ship_Date* is less than *Sale_Date*. Once the transaction is complete, **endEdit** posts the record and ends Edit mode.

```
; changeDate::pushButton
method pushButton(var eventInfo Event)

; first, see if you want to change Ship Date
if ORDERS.Sale Date > ORDERS.Ship Date then
  ; start Edit mode for the form
  ORDERS.edit()
  ; if Sale Date is later than Ship Date, change Ship Date
  ORDERS.Ship Date = ORDERS.Sale Date + 5
  ORDERS.endEdit()      ; end editing—changes to the record
                        ; can't be cancelled
endif

endmethod
```

See also

- cancelEdit, endEdit

---

empty

UIObject

Method

Deletes all records from a table.

Syntax

empty ( ) Logical

Description

Deletes all records from a table without prompting for confirmation. The table does not have to be in Edit mode. This operation cannot be undone.

In multiuser applications, this method tries, for the duration of the retry period, to place a full lock on the table. If a lock can't be placed, the method fails.

Example

The following example assumes a form with three buttons: *createTable*, *emptyTable*, and *deleteTable*. *createTable* creates a copy of the *Orders* table called *TmpOrder*, then places a table frame on the form and binds *TmpOrder* to it. *emptyTable* deletes all the records from *TmpOrder*.

deleteTable removes the table frame, removes the table from the data model of the form, and deletes the temporary table.

This is the code for the *createTable* button:

```
; createTable::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  ui  UIObject
endVar

tbl.attach("Orders.db")
tbl.copy("TmpOrder.db")  ; copy Orders to TmpOrder

ui.create(TableFrameTool, 720, 720, 4320, 1440) ; create TableFrame
ui.TableName = "TmpOrder.db" ; also adds table to data model
ui.Visible = True

endmethod
```

This is the code for the *emptyTable* button:

```
; emptyTable::pushButton
method pushButton(var eventInfo Event)
var
  ui  UIObject
endVar

if ui.attach("TMPORDER") then
  if msgYesNoCancel("Empty",
    "Delete all records from this table?") = "Yes" then
    ui.empty()  ; deletes all records from the TMPORDERS table
  endif
endif

endmethod
```

This is the code for the *deleteTable* button:

```
; deleteTable::pushButton
method pushButton(var eventInfo Event)
var
  tbl Table
  ui  UIObject
endVar

; clean up
if ui.attach("TMPORDER") then
  ui.delete()                  ; deletes table frame
  DMRemoveTable("TmpOrder.db") ; remove from data model
  tbl.attach("TmpOrder.db")
  tbl.delete()                 ; delete table from disk
endif
endmethod
```

See also

- deleteRecord

UIObject

end

UIObject

Beginner

Method

Moves to the last record in a table.

Syntax

end ( ) Logical

Description

Sets the current record to the last record in a table.

end has the same effect as the action constant DataEnd, so the following statements are equivalent:

```
obj.end()
```

```
obj.action(DataEnd)
```

Example

This example moves to the last record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToEnd* is a button on the same form.

```
; moveToEnd::pushButton
method pushButton(var eventInfo Event)
CUSTOMER.end()  ; move to the last record
                ; same as:  CUSTOMER.action(DataEnd)
msgInfo("At the last record?", CUSTOMER.atLast())
endmethod
```

See also

❐ home, nextRecord, priorRecord, currRecord, skip, moveTo

endEdit

UIObject

Beginner

Method

Leaves Edit mode, and accepts changes to the current record.

Syntax

endEdit ( ) Logical

Description

Takes a table out of Edit mode and posts changes to the current record.

endedit has the same effect as the action constant DataEndEdit, so the following statements are equivalent:

```
obj.endEdit()
```

```
obj.action(DataEndEdit)
```

Example

See the example for **edit**.

See also

- cancelEdit, edit

enumFieldNames

UIObject

Method

Fills an array with the names of the fields in a table.

Syntax

enumFieldNames (var ***fieldNames*** Array[] String)

Description

Fills *fieldArray* with the names of the fields in a table. If *fieldArray* is resizeable, it grows automatically to hold the field names; if it is not resizeable, it holds as many as it can, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

Example

The following example uses **enumFieldNames** to write the field names from the *Orders* table to an array named *fieldNames*. Assume that a form has a table frame bound to *Orders* and a button called *getFieldNames*.

```
; getFieldNames::pushButton
method pushButton(var eventInfo Event)
var
  fieldNames Array[] String
endVar
ORDERS.enumFieldNames(fieldNames)
fieldNames.view()
endmethod
```

See also

- enumObjectNames

enumLocks

UIObject

Method

Creates a Paradox table listing the locks currently applied to a UIObject; returns number of locks.

Syntax

enumLocks (const ***tableName*** String) LongInt

Description

Creates the Paradox table specified in *tableName*. *tableName* lists the locks currently applied to the table object. If *tableName* exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. For dBASE tables, this method lists only the lock you've placed (not all locks currently on the table).

You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of *tableName* is listed below:

Field Name	Type	Size
UserName	A	15
Lock Type	A	32
Net Session	N	
Session	N	
Record Number	N	

Example

In this example, the built-in **pushButton** method for the *showLocks* button creates a table listing the locks currently applied to the *Customer* table.

```
; showLocks::pushButton
method pushButton(var eventInfo Event)
var
  obj        UIObject
  howMany    LongInt
  enumTable TableView
endVar
obj.attach(CUSTOMER)              ; table frame on form
lock("Customer", "Write")         ; put a write lock on Customer
howMany = obj.enumLocks("lockenum.db")  ; enumerate locks
message("There are ", howMany, " locks on Customer table.")
enumTable.open("lockenum.db")     ; show the resulting table
enumTable.wait()
enumTable.close()
endmethod
```

See also

- lockStatus
- lock, lockStatus in the TCursor type

enumObjectNames — UIObject

Method/Procedure

Fills an array with the names of the objects in a form.

Syntax

enumObjectNames (var ***objectNames*** Array[] String)

Description

Fills an array with object names. If *objectNames* is resizeable, it grows automatically to hold the object names; if it is not resizeable, it holds as many as it can, and discards the rest. If *objectNames* already exists, this method overwrites it without asking for confirmation.

This method returns the names of bound objects and unbound objects, beginning with the object that called this method, and including paths to contained objects. So, to enumerate all objects in a form, put **enumObjectNames** in a method attached to the form.

Example

In the following example, assume a custom method named **customGetObjectN** is defined for the form. The form also contains a button named *getObjectNames* and all of the fields from the *Customer* table. The **pushButton** method for *getObjectNames* calls the form's custom method to write all object names on the form to an array, then to a table.

The following is attached to the custom method **customGetObjectN**:

```
; form::customGetObjectN  (custom method)
method customGetObjectN()
var
  objArray  Array [] String
endVar
enumObjectNames(objArray)          ; write names to array
objArray.view()                    ; show the array
enumUIObjectNames("objtable.db")   ; write names to table
endmethod
```

This code is for the **pushButton** method for *getObjectNames*:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
customGetObjectN()     ; call the custom method from the form
endmethod
```

See also

- enumUIObjectProperties, enumUIClasses

enumSource

UIObject

Method

Fills a table with the source code of the methods on a form.

Syntax

enumSource (const ***tableName*** String, [const ***recurse*** Logical]) Logical

Description

Fills a table with the source code of the methods on a form. If *tableName* already exists, this method overwrites it without asking for confirmation. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the table is:

Field Name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

If *recurse* is False, this method returns the method definitions for overridden methods on the current object only. To include the source code for overidden methods on objects contained by the current object, *recurse* should be True.

When *recurse* is True, **enumSource** returns the method definitions for overridden methods, beginning with the object that called this method, and including paths to contained objects. So, to enumerate the source for all objects in a form, put **enumSource** in a method attached to the form.

Example

In the following example, assume a custom method named **customGetSourceToTable** is defined for the form. The form also contains a button named *getSourceToTable*. The **pushButton** method for *getSourceToTable* calls the form's custom method, which writes all source code for all objects on the form to the *CstSource* table.This code is the **pushButton** method for *customGetSourceToTable*:

```
; customGetSourceToTable  (custom method)
method customGetSourceToTable()
self.enumSource("CstSourc.db", True)
endmethod
```

This is the **pushButton** method for *getSourceToTable*:

```
; getSourceToTable::pushButton
method pushButton(var eventInfo Event)
customGetSourceToTable()
endmethod
```

See also

- enumSourceToFile, enumUIObjectProperties, enumUIClasses

---

enumSourceToFile

UIObject

Method

Writes the source code for a form or an object to a text file.

Syntax

enumSourceToFile (const ***fileName*** String [, const ***recurse*** Logical]) Logical

Description

Writes the source code of the methods on a form to a text file. If *fileName* already exists, this method overwrites it without asking for

confirmation. You can include an alias or path in *fileName*; if no alias or path is specified, Paradox creates *fileName* in the working directory (:WORK:).

If *recurse* is False, this method returns the method definitions for overridden methods on the current object only. To include the source code for overidden methods on objects contained by the current object, *recurse* should be True.

When *recurse* is True, **enumSourceToFile** returns the method definitions for overridden methods, beginning with the object that called this method, and including paths to objects containing. So, to enumerate the source for all objects in a form, put **enumSourceToFile** in a method attached to the form.

Example

In the following example, assume that a custom method named **customGetSourceToFile** is defined for the form. The form also contains a button named *getSourceToFile*. The **pushButton** method for *getSourceToFile* calls the form's custom method, which writes all source code for all objects on the form to the CSTSOURCE.TXT file.

Following is the **pushButton** method for the *customGetSourceToFile* button:

```
; customGetSourceToFile  (custom method)
method customGetSourceToFile()
self.enumSourceToFile("CstSourc.txt", True)
endmethod
```

Following is the **pushButton** method for the *getSourceToFile* button:

```
; getSourceToFile::pushButton
method pushButton(var eventInfo Event)
customGetSourceToFile()
endmethod
```

See also

- enumSource, enumUIObjectProperties, enumUIClasses

enumUIClasses

UIObject

Procedure

Writes a list of UIObject classes to a table.

Syntax

enumUIClasses (const ***tableName*** String) Logical

Description

Creates a table *tableName* containing a list of all UIObject classes (such as bitmap, box, and field) and their property names. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The table structure is

Field Name	Type	Size
ClassName	A	32
PropertyName	A	64

Example

This example writes the types and properties to a table named *Tmpclass*.

```
; writeClasses::pushButton
method pushButton(var eventInfo Event)
enumUIClasses("TmpClass.db")
endmethod
```

See also

- enumUIObjectNames, enumUIObjectProperties

enumUIObjectNames

UIObject

Method/Procedure

Gets the names of each object in a form and writes them to a table.

Syntax

enumUIObjectNames (const ***tableName*** String) Logical

Description

Fills a table with object names. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the table is:

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

This method returns the names of bound objects and unbound objects, beginning with the object that called this method, and including paths to any contained objects. So, to enumerate all objects in a form, put **enumUIObjectNames** in a method attached to the form.

Example

In the following example, assume a custom method named **customGetObjectN** is defined for the form. The form also contains a button named *getObjectNames* and all the fields from the *Customer* table. The **pushButton** method for *getObjectNames* calls the form's custom method to write all object names on the form first to an array, then to a table. The following code is attached to the custom method **customGetObjectN**:

```
; form design::customGetObjectN  (custom method)
method customGetObjectN()
var
  objArray  Array [] String
endVar
enumObjectNames(objArray)        ; write names to array
objArray.view()                  ; show the array
enumUIObjectNames("objtable.db")  ; write names to table
endmethod
```

This is the code for *getObjectNames'* **pushButton** method:

```
; getObjectNames::pushButton
method pushButton(var eventInfo Event)
customGetObjectN()    ; call the custom method from the form
endmethod
```

See also

- enumObjectNames, enumUIObjectProperties, enumUIClasses

enumUIObjectProperties

UIObject

Method/Procedure

Gets the properties of each object in a form, and writes the data to a Paradox table.

Syntax

enumUIObjectProperties (const ***tableName*** String) Logical

Description

Gets the properties of each object in a form, and writes the data to the Paradox table specified in *tableName*. You can include an alias or path in *tableName*; if no alias or path is specified, Paradox creates *tableName* in the working directory (:WORK:).

The structure of the table is

Field Name	Type	Size
ObjectName	A	128
PropertyName	A	64
PropertyValue	A	255

Example

For the following example, assume that *getProperties* is a button on a form designed to show fields from the *Customer* table. The **pushButton** method for *getProperties* uses **enumUIObjectProperties** to write all of the property values for each object on the form to the table *CstProps*.

```
; getProperties::pushButton
method pushButton(var eventInfo Event)
enumUIObjectProperties("CstProps.db")
endmethod
```

See also

- enumUIClasses, enumUIObjectNames

execMethod

UIObject

Method/Procedure

Calls a custom method that takes no arguments.

Syntax

execMethod (const ***methodName*** String)

Description

Calls the custom method indicated by the string *methodName*. The method named in *methodName* can take no arguments. **execMethod** allows you to call a method based on the contents of a variable, which means the compiler does not know the method to call until run time.

Example

In the following example, assume that a form contains several fields, *fieldOne*, *fieldTwo*, and *fieldThree*. The form's Var window declares a dynamic array called *objPreProc*. The form's one custom method is called *fieldOnePreProc*. The form's **open** method (in the **isPreFilter=False** clause) creates elements in the *objPreProc* array: an element is created for each object on the form for which there is a preprocessing custom method.

In this example, *fieldOne* is assumed to require some preprocessing. An array element is created with an index of the object name "pageOne.fieldOne"; the value of the custom method name is "fieldOnePreProc". The **isPreFilter=True** clause of the form's open method—called for each object on the form—sorts out if an array element in *objPreProc* corresponds to the current object; if so, the custom method for that object is called. The following code defines the custom method **fieldOnePreProc**:

```
; form design::fieldOnePreProc  (custom method)
; This method is called during the form's preFilter clause,
; when the current object is fieldOne.
method fieldOnePreProc()
fieldOne.color = "Red"     ; change the color of the field
fieldOne.Value = "Initialized by the form's open method"
endmethod
```

This code is attached to the form's Var window:

```
; Var window for the form
Var
  ObjPreProc DynArray[] String  ; indexed by object name, will
                                ; hold names of methods to execute
                                ; when isPreFilter is true
endVar
```

This is the code for the form's **open** method:

```
method open(var eventInfo Event)
var
  targObj   UIObject   ; holds the target object
  targName  String     ; target object's name
  element   AnyType    ; index to dynamic array objPreProcs
endVar
if eventInfo.isPreFilter()
  then
    ; code here executes for each object in form
    eventInfo.getTarget(targObj)      ; find out who the current target is
    targName = targObj.name           ; get the name of the target
    foreach element in objPreProc     ; iterate through array
      if element = targName then      ; is there an element for the target?
        execMethod(objPreProc[targName])  ; if so, execute the corresponding
                                          ; custom method
      endif
    endforeach
  else
    ; code here executes just for form itself

    ; assign elements to the objPreProc array to indicate
    ; objects for which there is a preprocess custom method
    objPreProc["fieldOne"] = "fieldOnePreProc"
endif
endmethod
```

See also

❐ methodDelete, methodGet, methodSet

getBoundingBox

UIObject

Method

Returns the coordinates of the frame that bounds an object.

Syntax

getBoundingBox (var ***topLeft*** Point, var ***bottomRight*** Point)

Description

Returns the coordinates of the top left corner (*topLeft*) and the bottom right corner (*bottomRight*) of the invisible box (frame) that bounds an object, relative to the object's container. Strictly speaking, the bounding box is only visible in a design window. When you select an object in a design window, you can see its bounding box.

Example

This example draws a box around an ellipse based on the ellipse's bounding box. Assume that a form contains an ellipse called *redCircle.*

```
; redCircle::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  TopLeft,
  BotRight Point      ; to hold the points returned by getBoundingBox
  ui       UIObject   ; to create a new object
endVar

self.getBoundingBox(TopLeft, BotRight)
```

```
ui.create(BoxTool, TopLeft.x(),
                   TopLeft.y(),
                   BotRight.x() - TopLeft.x(),
                   BotRight.y() - TopLeft.y())
ui.Color = Green
ui.Translucent = Yes
ui.Visible = Yes

endmethod
```

See also

- convertPointWithRespectTo

getPosition

UIObject

Method

Locates the position of an object.

Syntax

getPosition (const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt)

Description

Locates the position of an object on the screen. Variables *x* and *y* specify the coordinates (in twips) of the upper left corner of the object. Variables *w* and *h* specify the width and height (in twips) of the object. If the object is not specified, self is implied.

Example

The following example moves a circle across the screen in response to timer events. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a TimerEvent every 100 milliseconds. Each TimerEvent causes *toggleButton's* timer method to execute. The **timer** method locates the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

This is the code for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then  ; if stopped, then start
  buttonLabel = "Stop Timer"         ; change label
  self.setTimer(100)           ; tell timer to issue a timer
                               ; event every 100 milliseconds
else
  buttonLabel = "Start Timer"        ; change label
  self.killTimer()                   ; stop the timer
endif

endmethod
```

This is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
;   this method is called once for every timer event
```

```
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar

ui.attach(floatCircle)          ; attach to the circle
ui.getPosition(x, y, w, h)      ; assign coordinates to vars
if x < 4320 then                ; if not at right edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the right
else
  ui.setPosition(1440, y, w, h)     ; return to the left
endif

endmethod
```

See also

- setPosition

getProperty

UIObject

Method

Returns the value of a specified property.

Syntax

getProperty (const ***propertyName*** String) AnyType

Description

Returns the value of the property specified in *propertyName*. Not all properties take strings as values. For example, if a property value is a number, this method returns a number. To return a string in every case, use **getPropertyAsString**.

getProperty is an alternative to getting a property directly; it's useful when *propertyName* is a variable. Otherwise, access the property directly, as in

```
thisColor = myBox.Color
```

Example

The following example creates a dynamic array, indexed by property names, to contain property values. The array is filled by using the array's index as the argument to the **getProperty** command.

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  propNames DynArray[] AnyType    ; to hold property names & values
  arrayIndex          String      ; index to dynamic array
endVar

propNames["Color"] = ""
propNames["Visible"] = ""
propNames["Name"] = ""

foreach arrayIndex in propNames   ; assign the properties to the array
  propNames[arrayIndex] = self.getProperty(arrayIndex)
endforeach
```

```
propNames["Color"] = "DarkBlue"

foreach arrayIndex in propNames   ; set properties from the array
  self.setProperty(arrayIndex, propNames[arrayIndex])
endforeach

endmethod
```

See also

- getPropertyAsString, setProperty

getPropertyAsString

UIObject

Method

Returns the value of a specified property as a string.

Syntax

getPropertyAsString (const ***propertyName*** String) String

Description

Returns a string containing the value of the property specified in *propertyName*.

Example

This example assigns the value of the Color property to an AnyType variable using the **getProperty** method. The value returned is a LongInt, because colors are long integer constants. Next, the Color property is obtained using **getPropertyAsString**. The value returned is a String type, such as "Blue".

```
; boxOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
var
  myColor  AnyType
endVar

myColor = self.getProperty("Color")
myColor.view()                              ; shows as LongInt
myColor = self.getPropertyAsString("Color")
myColor.view()                              ; shows as String
endmethod
```

See also

- getProperty, setProperty

getRGB

UIObject

Method

Finds the red, green, and blue components of a color.

Syntax

getRGB (const ***rgb*** LongInt, var ***red*** SmallInt, var ***green*** SmallInt, var ***blue*** SmallInt)

Description

Decomposes a composite color, *rgb*, to its component *red*, *green*, and *blue* values.

Example

This example determines the red, green, and blue components of the constant Brown.

```
; decompBrown::pushButton
method pushButton(var eventInfo Event)
var
  thisRed, thisBlue, thisGreen SmallInt
endVar
getRGB(Brown, thisRed, thisGreen, thisBlue)
msgInfo("Brown is really",
        String("Red ", thisRed, "  Green ", thisGreen,
               "  Blue ", thisBlue))
endmethod
```

See also

- rgb

hasMouse

UIObject

Method

Reports if the mouse is positioned over an object.

Syntax

hasMouse ( ) Logical

Description

Returns True if the pointer is positioned inside the boundaries of an object; otherwise, it returns False.

Example

The following example assumes that a form has a bitmap object called *cat*. The **open** method for *cat* sets the timer interval to 250 milliseconds. The **timer** method uses **hasMouse** to determine if *cat* has the mouse; if not, it moves *cat* to the mouse's position. This is the code for *cat*'s **open** method:

```
; cat::open
method open(var eventInfo Event)
; set the timer interval to 250 milliseconds
self.setTimer(250)
endmethod
```

This is the code for *cat*'s **timer** method:

```
; cat::timer
method timer(var eventInfo TimerEvent)
var
  mousePt  Point                    ; to get mouse position
endVar
if NOT cat.hasMouse() then          ; am I on the mouse?
  mousePt = getMouseScreenPosition() ; find the mouse
  cat.setPosition(mousePt.x() - 350,
                  mousePt.y() - 2880,
                  4320, 1750)       ; chase the mouse
```

```
  ; moves cat above and slightly to the left of mouse
  ; assumes cat is a bitmap with width 4320, height 1750
  ; since getMouseScreenPosition returns position of mouse
  ; on desktop, these numbers assume form is maximized
  ; offset (2880-1750) allows for height of menu and speedbar
endif
endmethod
```

See also

- The MouseEvent type

home

UIObject

Beginner

Method

Moves to the first record in a table.

Syntax

home ( ) Logical

Description

Sets the current record to the first record of a table. **home** respects the limits of restricted views displayed in a linked table frame or multi-record object; **home** moves to the first record in a restricted view.

home has the same effect as the action constant DataBegin, so the following statements are equivalent:

```
obj.home()

obj.action(DataBegin)
```

Example

This example moves to the first record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToHome* is a button on the same form.

```
; moveToHome::pushButton
method pushButton(var eventInfo Event)
CUSTOMER.home()  ; move to the first record
                 ; same as:  CUSTOMER.action(DataBegin)
msgInfo("At the first record?", CUSTOMER.atFirst())
endmethod
```

See also

- end, nextRecord, priorRecord

insertAfterRecord

UIObject

Method

Inserts a record into a table after the current record.

Syntax

insertAfterRecord ( ) Logical

Description

Inserts a new, blank record into a table after the current record. The table must be in Edit mode.

Example

In this example, suppose that *CustSort* is a copy of the *Customer* table, sorted by the Name field. The form contains a table frame named *CUSTSORT* bound to the *CustSort* table, an undefined field called *newField*, and a button called *insRecButton*. To add a new name to the table, type the name in *newField* and press *insRecButton*.

The following code is attached to the **pushButton** method for *insRecButton*. This method checks for a value in *newField*, then checks if the form is in Edit mode. If so, the method attaches the TCursor *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If it finds a name greater than the new name, the method uses **insertRecord** to insert a new blank record before the name found; otherwise, it uses **insertAfterRecord** to insert a new blank record at the end of the table.

```
; insRecButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC  TCursor
  nameStr String
endvar

if newField.Value = "" then      ; quit if the field is blank
  RETURN
endif
nameStr = newField.Value         ; get the name to add
CUSTSORT."Name".moveTo()
if thisForm.Editing then         ; check for edit mode first
  custTC.attach(CUSTSORT)
  scan custTC for custTC."Name" = nameStr:
    quitloop                     ; stop when you find the name
  endscan
  msgInfo("Current record no", custTC.recno())
  CUSTSORT.resync(custTC)        ; resync the cursor in CUSTSORT to the TC
  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord() ; inserts a blank record before current
                                 ; behaves just like insertRecord()
  else
    CUSTSORT.insertAfterRecord()  ; inserts a blank record after current
  endif
  ; ... fill the record with the rest of the customer information
else
  msgInfo("Sorry", "You must be in Edit mode to insert a record.")
endif
endmethod
```

See also

❒ insertRecord, insertBeforeRecord

insertBeforeRecord

UIObject

Method

Inserts a record into a table before the current record.

Syntax

insertBeforeRecord ( ) Logical

Description

Inserts a record into a table before the current record. The table must be in Edit mode.

insertBeforeRecord has the same effect as the action constant DataInsertRecord, so the following statements are equivalent:

```
obj.insertBeforeRecord()
```

```
obj.action(DataInsertRecord)
```

Example

In this example, suppose that *CustSort* is a copy of the *Customer* table, sorted by the Name field. The form contains a table frame named *CUSTSORT* bound to *CustSort*, an undefined field called *newField*, and a button called *insRecButton*. To add a new name to the table, type the name in *newField* and press *insRecButton*.

The following method overrides the **pushButton** method for *insRecButton*. This method checks for a value in *newField*, then checks if the form is in Edit mode. If so, the method attaches a TCursor named *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If it finds a name greater than the new name, the method uses **insertBeforeRecord** to insert a new blank record before the name found; otherwise, it uses **insertAfterRecord** to insert a new blank record at the end of the table.

```
; insRecButton::pushButton
method pushButton(var eventInfo Event)
var
  custTC  TCursor
  nameStr String
endvar

if newField.Value = "" then      ; quit if the field is blank
  RETURN
endif
nameStr = newField.Value         ; get the name to add
CUSTSORT."Name".moveTo()
if thisForm.Editing then         ; check for edit mode first
  custTC.attach(CUSTSORT)
  scan custTC for custTC."Name" = nameStr:
    quitloop                     ; stop when you find the name
  endscan
  msgInfo("Current record no", custTC.recno())
  CUSTSORT.resync(custTC)        ; resync the cursor in CUSTSORT to the TC
  if NOT CUSTSORT.atLast() then
    CUSTSORT.insertBeforeRecord() ; inserts a blank record before current
                                  ; behaves just like insertBeforeRecord()
  else
```

```
    CUSTSORT.insertRecord()       ; inserts a blank record after current
  endif
  ; ... fill the record with the rest of the customer information
else
  msgInfo("Sorry", "You must be in edit mode before inserting a record.")
endif
endmethod
```

See also

❒ insertAfterRecord, insertRecord

insertRecord

UIObject

Beginner

Method

Inserts a record into a table.

Syntax

insertRecord ( ) Logical

Description

Inserts a record before the current record into a table.

insertRecord has the same effect as **insertBeforeRecord** and the action constant DataInsertRecord, so the following three statements are equivalent:

```
obj.insertRecord()
```

```
obj.insertBeforeRecord()
```

```
obj.action(DataInsertRecord)
```

Example

See the example for insertBeforeRecord.

See also

❒ insertAfterRecord, insertBeforeRecord

isContainerValid

UIObject

Procedure

Reports whether an object's container is valid.

Syntax

isContainerValid ( ) Logical

Description

Reports if the current object's container is valid. For instance, a form has no container, so the Container property for a form is not valid.

Example

In this example, the **arrive** built-in method for a form uses **isContainerValid** to check for a container:

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
  if eventInfo.isPreFilter() then

     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isContainerValid() then
       msgInfo("Form", "This object does not have a valid container.")
     endif
  endif
endmethod
```

See also

- isLastMouseClickedValid

isEdit

Beginner

UIObject

Method

Reports whether an object is in Edit mode.

Syntax

isEdit ( ) Logical

Description

Reports whether an object is in Edit mode.

Example

See the example for lockRecord.

isEmpty

UIObject

Method

Reports if a table contains any records.

Syntax

isEmpty ( ) Logical

Description

Returns True if no records in the table are associated with the table frame. **isEmpty** respects the limits of restricted views displayed in a linked table frame or multi-record object.

You can also find out if a table is empty by checking the value returned by the **nRecords** method, or by checking the value of the NRecords property of the object.

Example

The *cascadeDelete* button in this example deletes an order and all the linked detail records for that order. Assume that a form contains a single-record object bound to the *Orders* table and a linked table frame bound to the *Lineitem* table. *Orders* has a one-to-many link to *Lineitem*.

```
; cascadeDelete::pushButton
method pushButton(var eventInfo Event)
var
  ui        UIObject
endVar

if thisForm.Editing then
  if msgQuestion("Confirm", "Delete this order?") = "Yes" then
    ui.attach(LINEITEM)
    while NOT ui.isEmpty()      ; check to see if linked table is
                                ; empty—respects restricted view
      ui.deleteRecord()         ; delete the detail records
    endwhile
    ORDERS.action(DataDeleteRecord)   ; delete the master record
  endif
else
  msgInfo("Status", "You must be editing to delete a record.")
endif
endmethod
```

See also

❐ empty, nRecords

isLastMouseClickedValid

UIObject

Procedure

Reports if the last object to receive a click is valid.

Syntax

isLastMouseClickedValid ( ) Logical

Description

Reports if the current form has received a mouse click since it opened.

Example

This method checks to see if a form has been clicked yet.

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
   if eventInfo.isPreFilter() then
     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isLastMouseClickedValid() then
       msgInfo("FYI", "This form has not been clicked yet.")
     endif
  endif
endmethod
```

See also

❐ islastMouseRightClickedValid

isLastMouseRightClickedValid

UIObject

Procedure

Reports if the last object to receive a right mouse click is valid.

Syntax

isLastMouseRightClickedValid ( ) Logical

Description

Reports if the current form has received a right mouse click since it opened.

Example

This method checks to see if a form has been right-clicked yet.

```
; thisForm::arrive
method arrive(var eventInfo MoveEvent)
  if eventInfo.isPreFilter() then

     ;Code here executes before each object
  else
     ;Code here executes afterwards (or for form)
     if NOT isLastMouseRightClickedValid() then
       msgInfo("FYI", "This form has not been right-clicked yet.")
     endif
  endif
endmethod
```

See also

- isLastMouseClickedValid

isRecordDeleted

UIObject

Method

Reports whether the current record has been deleted (dBASE tables only).

Syntax

isRecordDeleted ( ) Logical

Description

Reports whether the current record has been deleted. **isRecordDeleted** works only for dBASE tables because deleted Paradox records can't be displayed. This method returns True if the current record has been deleted; otherwise, it returns False.

Deleted records in a dBASE table are not shown by default. For **isRecordDeleted** to work correctly, you must call **showDeleted** to show deleted records in the table; otherwise, deleted records are not visible to **isRecordDeleted**.

See also

- undeleteRecord
- isShowDeletedOn, showDeleted in the TCursor type

keyChar

UIObject

Beginner

Method

Sends an event to an object's **keyChar** method.

Syntax

1. **keyChar** (const ***characters*** String [, const ***state*** SmallInt]) Logical
2. **keyChar** (const ***ansiKeyValue*** SmallInt) Logical
3. **keyChar** (const ***ansiKeyValue*** SmallInt, const ***vChar*** SmallInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **keyChar** method of an object with that event.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following example overrides the **pushButton** method of a button named *sendKeyChar*. This method sends keystrokes to a field, called *fieldOne*, on the same form.

```
; sendKeyChar::pushButton
method pushButton(var eventInfo Event)
var
  x  SmallInt
endVar
fieldOne.keyChar("Send me an ")      ; send a string
fieldOne.keyChar(65, 65, Shift)      ; send ANSI char, decimal
                                     ; equivalent of VK_Char,
                                     ; and keyboardstate
fieldOne.keyChar(" and a ", Shift) ; send a string with the keyboardstate
x = 98                               ; set the code
fieldOne.keyChar(x)                  ; send ANSI char code

endmethod
```

See also

- keyPhysical
- The KeyEvent type

keyPhysical

UIObject

Method

Sends an event to an object's **keyPhysical** method.

Syntax

keyPhysical (const ***ansiKeyValue*** SmallInt, const ***vChar*** SmallInt, const ***state*** SmallInt) Logical

Description

Sends an event to an object's **keyPhysical** method.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following code is attached to the **pushButton** method of a button named *sendKeyPhys*. This method sends the character "a" to the field *fieldOne*.

```
; sendKeyPhys::pushButton
method pushButton(var eventInfo Event)
fieldOne.keyPhysical(97, 97, Shift)   ; send an "a"
endmethod
```

See also

- keyChar
- The KeyEvent type

killTimer

UIObject

Method

Stops the timer associated with an object.

Syntax

killTimer ( )

Description

Stops a timer associated with an object.

Example

The following example moves a circle across the screen in response to TimerEvents. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a TimerEvent every 100 milliseconds. Each TimerEvent causes *toggleButton's* timer method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**. The following code is attached to *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then  ; if stopped, then start
  buttonLabel = "Stop Timer"         ; change label
  self.setTimer(100)          ; tell timer to issue a timer
                              ; event every 100 milliseconds
else
  buttonLabel = "Start Timer"        ; change label
  self.killTimer()                   ; stop the timer
endif

endmethod
```

This is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
;   this method is called once for every timer event
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar

ui.attach(floatCircle)          ; attach to the circle
ui.getPosition(x, y, w, h)      ; assign coordinates to vars
if x < 4320 then                ; if not at right edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the right
else
  ui.setPosition(1440, y, w, h)     ; return to the left
endif

endmethod
```

See also

- setTimer

locate

UIObject

Beginner

Method

Searches for a specified value.

Syntax

1. **locate** (const ***fieldName*** String, const ***exactMatch*** AnyType [,const ***fieldName*** String, const ***exactMatch*** AnyType]*) Logical
2. **locate** (const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType [,const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType]*) Logical

Description

Searches a table frame or multi-record object for records whose values match the criteria specified in one or more field/value pairs. This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search always starts from the beginning of the table, but if no match is found, the pointer returns to the current record. If a match is found, the pointer moves to that record. This operation fails if the current record cannot be posted and unlocked (for example, because of a key violation).

Example

In the following example, assume that a form contains a table frame bound to the *Customer* table, and a button named *locateButton*. The **pushButton** method for *locateButton* attempts to find the customer named "Sight Diver" in the city "Kato Paphos". If found, the customer's name is changed to "Right Diver".

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
  Cust UIObject
endVar
Cust.attach(CUSTOMER)
; find customer named "Sight Diver" in Kato Paphos
if Cust.locate("Name", "Sight Diver", "City", "Kato Paphos") then
   Cust.edit()
   Cust."Name" = "Right Diver"
   Cust.endEdit()
endIf
endmethod
```

See also

- locateNext, locateNextPattern, locatePattern

locateNext

UIObject

Beginner

Method

Searches forward from the current record for a specified field value.

Syntax

1. **locateNext** (const ***fieldName*** String,
   const ***exactMatch*** AnyType [, const ***fieldName*** String,
   const ***exactMatch*** AnyType]*) Logical
2. **locateNext** (const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType [, const ***fieldNum*** SmallInt,
   const ***exactMatch*** AnyType]*) Logical

Description

Searches a table for records whose values match the criteria specified in one or more field/value pairs. This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search begins with the record after the current record. If a match is found, the pointer moves to that record. If no match is found, the pointer returns to the current record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be committed (for example, because of a key violation).

Example

For this example, suppose a form contains a table frame bound to the *Customer* table, and one button named *locateButton*. The **pushButton** method for *locateButton* searches for customers in the city of Freeport. If the first **locate** is successful, the method uses **locateNext** to find successive records.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
```

```
   Cust      UIObject
   searchFor String
   numFound  SmallInt
endVar
Cust.attach(CUSTOMER)
searchFor = "Freeport"
if Cust.locate("City", searchFor) then
   numFound = 1
   message("")
   while Cust.locateNext("City", searchFor)
      numFound = numFound + 1
   endWhile
   msgInfo("Found " + searchFor, strval(numFound) + " times.")
endIf
```

See also

- locate, locateNextPattern

locateNextPattern

UIObject

Method

Locates the next record containing a field that has a specified pattern of characters.

Syntax

1. **locateNextPattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType,] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locateNextPattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType,] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds substrings (for example, "comp" in "computer"). The search begins with the record after the current record. If a match is found, the pointer moves to that record. If no match is found, the pointer returns to the current record. This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session

UIObject

type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database").

```
tc.locateNextPattern("Name", "Borland",  "Product", "Paradox", "Keywords",
"data..")
```

Example

The following example searches for multiple occurrences of the letter "C" in the Name field of the *Customer* table, and writes the matching names to an array. Suppose that the CUSTOMER table frame is bound to *Customer*, and *locateButton* is a button on the same form.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject           ; to attach to CUSTOMER table frame
   searchFor String             ; the pattern string to search for
   numFound  SmallInt           ; the number of matches located
   custNames Array[] String     ; the matches found
endVar

cust.attach(CUSTOMER)
searchFor = "C.."               ; find customers whose name
                                ; begins with C
if cust.locatePattern("Name", searchFor) then  ; if you can find one
  numFound = 1                                 ; post it to the array
  custNames.grow(1)                            ; then keep looking

  custNames[numFound] = cust."Name"
  while cust.locateNextPattern("Name", searchFor)
    numFound = numFound + 1
    custNames.grow(1)
    custNames[numFound] = cust."Name"
  endWhile
endIf
if custNames.size() > 0 then     ; if there's anything in the array
  custNames.view()               ; show the array
endIf
endmethod
```

This example is similar to the previous example, except that it searches for records based on the value of the City field and a pattern in the Name field:

```
; locateButtonTwo::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject           ; to attach to CUSTOMER TableFrame
   searchFor String             ; the pattern string to search for
   numFound  SmallInt           ; the number of matches located
   custNames Array[] String     ; the matches found
```

```
endVar

cust.attach(CUSTOMER)
searchFor = "..C.."             ; find customers whose name
                                ; includes a C
if cust.locatePattern("City", "Marathon", "Name", searchFor) then  ; if you can
find one
  numFound = 1                                  ; post it to the array
  custNames.grow(1)                             ; then keep looking

  custNames[numFound] = cust."Name"
  while cust.locateNextPattern("City", "Marathon", "Name", searchFor)
    numFound = numFound + 1
    custNames.grow(1)
    custNames[numFound] = cust."Name"
  endWhile
endIf
if custNames.size() <> 0 then     ; if there's anything in the array
  custNames.view()               ; show the array
endIf
endmethod
```

See also

- locate, locateNext, locatePattern
- advMatch, match in the String type

locatePattern

UIObject

Method

Searches for a record containing a field that has a specified pattern of characters.

Syntax

1. **locatePattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType,] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locatePattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType,] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds substrings (for example, "comp" in "computer"). This method uses active indexes when it can to speed the search. This method respects the limits of restricted views in linked detail tables.

The search always starts at the beginning of the table, but if no match is found, the pointer returns to the current record. If a match is found, the pointer moves to that record. This operation fails if the current record cannot be committed (for example, because of a key violation).

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox," and the Keywords field for words beginning with "data" (for example, database).

To start a search after the current record, use **locateNextPattern**.

```
tc.locatePattern("Name", "Borland",  "Product", "Paradox", "Keywords", "data..")
```

Example

See the example for locateNextPattern.

See also

- locate, locateNextPattern
- advMatch, match in the String type

locatePrior

UIObject

Method

Searches backward for a specified field value.

Syntax

1. **locatePrior** (const ***fieldName*** String,
   const ***searchValue*** AnyType [,const ***fieldName*** String,
   const ***searchValue*** AnyType]*) Logical
2. **locatePrior** (const ***fieldNum*** SmallInt,
   const ***searchValue*** AnyType [,const ***fieldNum*** SmallInt,
   const ***searchValue*** AnyType]*) Logical

Description

Searches backward from the current record in a table for records whose values match the criteria specified in one or more field/value pairs. This method uses active indexes when it can to speed the search. It respects the limits of restricted views in linked detail tables.

The search begins with the record before the current record and moves up through the table. If a match is found, the pointer moves to

that record. If no match is found, the pointer returns to the current record. To start a search from the beginning of a table, use **locate**.

This operation fails if the current record cannot be committed (for example, because of a key violation).

Example

The method shown in this example locates the last occurrence of a value in a table by moving to the end of the table and using **locatePrior** to search up for a match. Assume that the form contains a table frame bound to the *Customer* table, and one button named *locateButton*.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject      ; to attach to CUSTOMER table frame
   searchFor String        ; the string to search for
endVar
Cust.attach(CUSTOMER)      ; attach to table frame
Cust.end()                 ; move to the end of the table
searchFor = "Freeport"
if Cust.locatePrior("City", searchFor) then   ; find record
   msgInfo("Status", "The last record with a City of " +
           searchFor + " is record " + Cust.recno + ".")
endIf

endmethod
```

See also

- locate, locatePriorPattern

locatePriorPattern

UIObject

Method

Locates the prior record containing a field that has a specified pattern of characters.

Syntax

1. **locatePriorPattern** ([const ***fieldName*** String, const ***exactMatch*** AnyType,] * const ***fieldName*** String, const ***pattern*** String) Logical
2. **locatePriorPattern** ([const ***fieldNum*** SmallInt, const ***exactMatch*** AnyType,] * const ***fieldNum*** SmallInt, const ***pattern*** String) Logical

Description

Finds substrings (for example, "comp" in "computer"). This method uses active indexes when it can to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record before the current record and moves up through the table. If a match is found, the pointer moves to that record. If no match is found, the pointer returns to the current

record. This method uses active indexes when it can to speed the search.

This operation fails if the current record cannot be committed (for example, because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records based on the value of a single field, specify the field in *fieldName* or *fieldNum*, and specify a pattern of characters in *pattern*.

You can include the pattern operators @ and .. in the *pattern* argument. The .. operator stands for any string of characters (including none at all); @ stands for any single character. Any combination of literal characters and wildcards can be used in constructing a search. If **advancedWildcardsInLocate** (in the Session type) is on, you can use advanced match pattern operators, not the standard pattern operators. See the description of **advMatch** in the String type for more information about advanced match pattern operators, and **advancedWildcardsInLocate** and **isAdvancedWildcardsInLocate** in the Session type.

To search records based on the values of more than one field, specify exact matches on all fields *except* the last one in the list. For example, the next statement searches the Name field for exact matches on "Borland", the Product field for "Paradox", and the Keywords field for words beginning with "data" (for example, "database").

```
obj.locatePriorPattern("Name", "Borland",  "Product", "Paradox", "Keywords",
"data..")
```

Example

The method shown in this example locates the last occurrence of a value in a table by moving to the end of the table and using **locatePriorPattern**. Assume that the form contains a table frame bound to the *Customer* table, and one button named *locateButton*.

```
; locateButton::pushButton
method pushButton(var eventInfo Event)
var
   Cust      UIObject       ; to attach to CUSTOMER table frame
   searchFor String         ; the string to search for
endVar
Cust.attach(CUSTOMER)       ; attach to table frame
Cust.end()                  ; move to the end of the table
searchFor = "Freeport"
if Cust.locatePrior("City", searchFor, "Name", "..C..") then   ; find record
   msgInfo("Status", "The last record with a City of " + searchFor +
           "and a name with C is record " + Cust.recno + ".")
endIf

endmethod
```

See also

- locatePattern, locatePrior
- advMatch, match in the String type

lockRecord

UIObject

Beginner

Method

Puts a write lock on the current record.

Syntax

lockRecord ( ) Logical

Description

Returns True if it successfully places an explicit write lock on the current record; otherwise, it returns False. If the record already exists, it is locked and becomes the current record.

Note You can also examine the locked property of an object to find out whether an object is locked, but you can't change the property to lock or unlock an object. The Locked property is a read-only property.

Example

This example first checks to see if the *Customer* table is in Edit mode. If so, the method locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*. Assume also that the record inside the CUSTOMER table frame is named *custRec*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  obj  UIObject
endVar
obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")
if CUSTOMER.isEdit() then
  if NOT obj.lockRecord() then
     msgStop("Lock failed", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
  else
     msgStop("Lock succeeded", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
     obj.custRec."Name" = "Right Diver"  ; quotes on Name indicate
                                         ; field name instead of property
     obj.unlockRecord()
  endif
else
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endif
endmethod
```

The next example shows how you can examine the Locked property for a record object to determine if the record is locked. This example behaves roughly the same as the previous example.

```
; lockButtonTwo::pushButton
method pushButton(var eventInfo Event)
var
  obj,
  recObj  UIObject
endVar
```

```
obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")

if thisForm.editing then
  obj.lockRecord()                ; no write access to Locked property
                                  ; so use method to lock record
  recObj.attach(CUSTOMER.custRec)
  if NOT recObj.Locked then       ; check the property to see
                                  ; if the record is locked
     msgStop("Lock failed", "recObj.Locked is " +
             String(recObj.Locked))
  else
     msgStop("Lock succeeded", "recObj.Locked is " +
             String(recObj.Locked))
     recObj."Name" = "Right Diver" ; name is in quotes to indicate Name
                                  ; field instead of obj's Name property
     obj.unlockRecord()
  endif
else
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endIf
endmethod
```

See also

- postRecord, recordStatus, unlockRecord

lockStatus

UIObject

Method

Returns the number of locks on a table.

Syntax

lockStatus (const ***lockType*** String) SmallInt

Description

Returns the number of times you've placed a lock of type *lockType* on a table. Specify a *lockType* of "Write", "Read", or "Any".

If you specify "Any" for *lockType*, **lockStatus** returns the total number of locks you've placed on the table. **lockStatus** only reports on locks you've placed explicitly, not on locks placed by Paradox or by other users or applications.

If you haven't placed any locks of a given type, **lockStatus** returns 0.

Example

This example assumes that a form has a table frame named *CUSTOMER* bound to the *Customer* table, and a button named *lockButton*. The **pushButton** method for *lockButton* removes all locks from *CUSTOMER*, checks for locks with **lockStatus**, places a lock, then reports on the locks with **lockStatus** again.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  CustTC TCursor     ; to place a lock on the table
  Cust UIObject
```

```
  l    Logical
endVar
CustTC.attach(CUSTOMER)     ; attach the TCursor to CUSTOMER
l = unlock(CustTC, "ALL")   ; remove any locks
l.view("Unlock successful:")
Cust.attach(CUSTOMER)       ; attach the UIObject to CUSTOMER
if Cust.lockStatus("ANY") = 0 then   ; check for locks
   l = lock(CustTC, "WL")            ; place a write lock
   l.view("Lock successful:")        ; check up on it
endif
msgInfo("Status", "Table " + Cust.Name + " has " +
         String(Cust.lockStatus("WL")) + " write lock(s).")
unlock(CustTC, "ALL")                ; remove any locks
endmethod
```

See also

- enumLocks, lockRecord

menuAction

UIObject

Method/Procedure

Sends an event to an object's **menuAction** method.

Syntax

menuAction (const ***action*** SmallInt) Logical

Description

Constructs a MenuEvent and sends it to a built-in **menuAction** method. *action* is one of the MenuCommand constants, or a user-defined constant. ObjectPAL provides constants for *action*; see MenuCommands in the Constants dialog box. For more information on user-defined constants, see Chapter 6 in the *ObjectPAL Developer's Guide*.

Note You can't use **menuAction** to send a menu command constant that is equivalent to a command on the File menu. To simulate a File menu command, use one of the regular Action constants, manipulate a property, or use one of the many System type methods that emulate File menu commands.

Example

In this example, the *sendATile* button on the current form sends the form (*thisForm*) a MenuWindowTile action.

```
; sendATile::pushButton
method pushButton(var eventInfo Event)
thisForm.menuAction(MenuWindowTile)
endmethod
```

See also

- action

methodDelete

UIObject

Method

Deletes a specified method.

Syntax

methodDelete (const ***methodName*** String) Logical

Description

Deletes the method specified by *methodName*. The form that contains the object must be in the Form Design window.

Example

This example uses **methodGet**, **methodSet**, and **methodDelete** to copy methods from one object to another. The method shown here overrides the **pushButton** method for a button named *copyMethods*. Four other objects are on the same form: the *targetForm* field lets you specify the name of the form containing the objects to copy; the *sourceObject* field holds the name of the object containing the methods to copy; the *destinationObject* field contains the name of the object to copy the methods to; and a radio button field, named *copyOrMove*, lets you specify whether methods in the source should be copied, or copied then deleted.

```
; copyMethods::pushButton
method pushButton(var eventInfo Event)
var
  otherForm            Form       ; a handle to a form
  sourceObj,                      ; object to copy from
  destObj              UIObject   ; object to copy to
  methodStr            String     ; stores the method definition
  methodArray Array[]  String     ; holds method names to copy
  i                    SmallInt   ; array index
endvar

; open the form and attach to the objects
if targetForm = "" OR sourceObject = "" OR destinationObject = "" then
  msgStop("Error", "Please fill in form, source, and destination.")
  return
endif
if NOT otherForm.load(targetForm.value) then
  msgStop("Error", "Couldn't open named form.")
  return
endif
if NOT sourceObj.attach(otherForm, sourceObject.value) then
  otherForm.close()
  msgStop("Error", "Couldn't find source object. Please specify entire path.")
  return
endif
if NOT destObj.attach(otherForm, destinationObject.value) then
  otherForm.close()
  msgStop("Error", "Couldn't find destination object. Specify entire path.")
  return
endif

; set up the array of method names to copy
methodArray.addLast("mouseUp")
methodArray.addLast("mouseDown")
methodArray.addLast("mouseDouble")
methodArray.addLast("mouseEnter")
```

```
methodArray.addLast("mouseExit")
methodArray.addLast("mouseRightUp")
methodArray.addLast("mouseRightDown")
methodArray.addLast("mouseRightDouble")
methodArray.addLast("mouseMove")
methodArray.addLast("open")
methodArray.addLast("close")
methodArray.addLast("canArrive")
methodArray.addLast("arrive")
methodArray.addLast("setFocus")
methodArray.addLast("canDepart")
methodArray.addLast("depart")
methodArray.addLast("removeFocus")
methodArray.addLast("depart")
methodArray.addLast("timer")
methodArray.addLast("keyPhysical")
methodArray.addLast("keyChar")
methodArray.addLast("action")
methodArray.addLast("menuAction")
methodArray.addLast("error")
methodArray.addLast("status")

; add the method names specific to fields and buttons
if sourceObj.class = "Field" AND destObj.class = "Field" then
  methodArray.addLast("changeValue")
  methodArray.addLast("newValue")
endif

if sourceObj.class = "Button" AND destObj.class = "Button" then
  methodArray.addLast("pushButton")
endif
if sourceObj.class <> "Button" AND destObj.class <> "Button" then
  methodArray.addLast("mouseClick")
endif

; copy methods from sourceObj to destObj on form otherForm
for i from 1 to methodArray.size()
  ; write the method named in methodArray to the string
;  msgInfo("methodArray is", methodArray[i])
  try
    methodStr = sourceObj.methodGet(methodArray[i])
    msgInfo("FYI", "Retrieved " + methodArray[i] + " method.")
    ; write the string to the method named in methodArray
    destObj.methodSet(methodArray[i], methodStr)
    if copyOrMove.Value = "Move" then
      sourceObj.methodDelete(methodArray[i])
    endif
  onfail
  ;  loop
  endTry
endfor

endmethod
```

See also

❒ create, methodGet, methodSet

---

methodGet

UIObject

Method

Returns the text of a specified method.

Syntax **methodGet** (const ***methodName*** String) String

Description Returns a string containing the text of the method specified in *methodName*.

Example See the example for methodDelete.

See also ❐ create, methodDelete, methodSet

methodSet **UIObject**

Method Sets the text of a specified method.

Syntax **methodSet** (const ***methodName*** String, const ***methodText*** String) Logical

Description Specifies in *methodText* the source code for the method named in *methodName*. The form that contains the object should be in the Form Design window.

Example See the example for methodDelete.

See also ❐ create, methodDelete, methodGet

mouseClick **UIObject**

Method Generates a mouseClick MouseEvent and sends it to an object.

Syntax **mouseClick** ( ) Logical

Description Constructs an event and calls the built-in **mouseClick** method of an object with that event.

Example The following example sends a **mouseClick** MouseEvent to *fieldTwo* on the same form:

```
; sendMouseClick::pushButton
method pushButton(var eventInfo Event)
; send a mouseClick to fieldTwo
fieldTwo.mouseClick()
endmethod
```

See also

❒ mouseUp

mouseDouble

UIObject

Method

Sends an event to an object's **mouseDouble** method.

Syntax

mouseDouble (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseDouble** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) to use with *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following example sends a **mouseDouble** MouseEvent to *fieldTwo* on the same form:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseDouble to fieldTwo
fieldTwo.mouseDouble(100, 100, LeftButton)
endmethod
```

See also

❒ mouseDown, mouseUp, mouseRightDouble

mouseDown

UIObject

Method

Sends an event to an object's **mouseDown** method.

Syntax

mouseDown (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseDown** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state

constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following example sends a mouseDown and a mouseUp MouseEvent to the object *fieldOne* on the same form:

```
method pushButton(var eventInfo Event)
var
  fPt  Point
endVar
fPt = fieldOne.Position
fieldOne.mouseDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)
fieldOne.mouseUp(fPt.x(), fPt.y(), LeftButton)
endmethod
```

See also

- mouseDouble, mouseRightDown, mouseRightUp, mouseUp

mouseEnter

UIObject

Method

Sends an event to an object's **mouseEnter** method.

Syntax

mouseEnter (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseEnter** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following example sends a **mouseEnter** MouseEvent to a field named *fieldSix* on the same form:

```
; sendMouseEnter::pushButton
method pushButton(var eventInfo Event)
; send a mouseEnter to fieldSix
fieldSix.mouseEnter(100,100,LeftButton)
endmethod
```

See also

- mouseExit, mouseMove

mouseExit

UIObject

Method

Sends an event to an object's **mouseExit** method.

Syntax

mouseExit (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseExit** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

This example sends a **mouseExit** MouseEvent to *fieldSeven* on the same form:

```
; sendMouseExit::pushButton
method pushButton(var eventInfo Event)
; send a mouseExit to fieldSeven
fieldSeven.mouseExit(100, 100, LeftButton)
endmethod
```

See also

- ❐ mouseEnter, mouseMove

mouseMove

UIObject

Method

Sends an event to an object's **mouseMove** method.

Syntax

mouseMove (const ***x*** LongInt, const ***y*** LongInt,
const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseMove** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

This example sends a **mouseDown**, a **mouseUp**, and a **mouseMove** HomeEvent to a field named *fieldFive* on the same form:

```
; sendMouseMove::pushButton
method pushButton(var eventInfo Event)
fieldFive.mouseDown(100, 100, LeftButton)
fieldFive.mouseUp(100, 100, LeftButton)
; send a mouseMove to fieldFive
fieldFive.mouseMove(100, 100, LeftButton)
endmethod
```

See also

- mouseEnter, mouseExit

mouseRightDouble

UIObject

Method

Sends an event to an object's **mouseRightDouble** method.

Syntax

mouseRightDouble (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseRightDouble** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard. Calling this method triggers the receiving object's **mouseRightDouble** method.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

This example sends a **mouseRightDouble** MouseEvent to a field named *fieldTwo* on the same form:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseDouble to fieldTwo
fieldTwo.mouseDouble(100, 100, LeftButton)
endmethod
```

See also

- mouseDouble, mouseRightUp, mouseRightDown

mouseRightDown

UIObject

Method

Sends an event to an object's **mouseRightDown** method.

Syntax

mouseRightDown (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseRightDown** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

This example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the same form:

```
; sendMouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  fPt Point
endVar
fP - fieldThree.position    ; get the position, send a mouseRightDown
fieldThree.mouseRightDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)                  ; pause, then send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endmethod
```

See also

- mouseRightUp, mouseRightDouble, mouseUp, mouseDown

---

mouseRightUp

UIObject

Method

Sends an event to an object's **mouseRightUp** method.

Syntax

mouseRightUp (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseRightUp** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twips). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

This example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the same form:

```
; sendMouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
  fPt Point
endVar
fP = fieldThree.position    ; get the position, send a mouseRightDown
fieldThree.mouseRightDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)                  ; pause, then send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endmethod
```

See also

- mouseDown, mouseRightDouble, mouseRightDown, mouseUp

---

mouseUp

UIObject

Method

Sends an event to an object's **mouseUp** method.

Syntax

mouseUp (const ***x*** LongInt, const ***y*** LongInt, const ***state*** SmallInt) Logical

Description

Constructs an event and calls the built-in **mouseUp** method of an object with that event. The event will have the coordinates specified in *x* and *y* (in twipointerps). Use *state* to specify the state of the mouse or keyboard.

ObjectPAL provides constants (for example, LeftButton) for *state*; see KeyboardStates in the Constants dialog box. The keyboard state constants can be added together to create combined key states, such as *Alt+Ctrl*.

Example

The following example sends a **mouseDown** and a **mouseUp** MouseEvent to the object *fieldOne* on the same form:

```
method pushButton(var eventInfo Event)
var
  fPt  Point
endVar
fPt = fieldOne.Position
fieldOne.mouseDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)
fieldOne.mouseUp(fPt.x(), fPt.y(), LeftButton)
endmethod
```

See also

- mouseDouble, mouseDown, mouseRightDouble, mouseRightDown, mouseRightUp

moveTo

UIObject

Beginner

Method/Procedure

Sets the focus to a specified object.

Syntax

1. (Method) **moveTo** ( ) Logical
2. (Procedure) **moveTo** (const ***ObjectName*** String) Logical

Description

Moves the focus to a specified object. If you call **moveTo** as a procedure, specify the name of the object to move as a string in *objectName*.

Example

In this example, assume a form contains a table frame bound to *Orders*, and another table frame bound to *LineItem*. *Orders* has a one-to-many link to *LineItem*. A button named *findDetails* is also on the form. Suppose you want to be able to search through the entire *LineItem* table—not just through those records linked to the current order. In this case, the **pushButton** method for *findDetails* searches for orders that include the current part number.

This code is attached to the Var window for *findDetails*:

```
; findDetails::Var
Var
  lineTC TCursor  ; instance of LINEITEM for searching
endVar

; findDetails::open
method open(var eventInfo Event)
lineTC.open("LineItem.db")
endmethod
```

This is the code for *findDetails'* **pushButton** method:

```
; findDetails::pushButton
method pushButton(var eventInfo Event)
var
  stockNum  Number
  orderTC    TCursor
  OrderNum    Number
endVar

; get Stock No from current LineItem
stockNum = LINEITEM.lineRecord."Stock No"
; lineTC was declared in Var window and opened by open method
if NOT lineTC.locateNext("Stock No", stockNum) then
  lineTC.locate("Stock No", stockNum)
endif
orderTC.attach(ORDERS)
orderTC.locate("Order No", lineTC."Order No")
ORDERS.moveToRecord(orderTC)       ; move to CUSTOMER and
                             ; resynchronize with TCursor
LINEITEM.lineRecord.Stock No.moveTo()   ; move cursor to LINEITEM detail
; move cursor to matching record
LINEITEM.locate("Stock No", stockNum)
endmethod
```

This code is for *findDetails'* **close** method:

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close()   ; close the TCursor to LineItem
endmethod
```

See also

- attach

moveToRecNo

UIObject

Method

Movesto a specific record in a dBASE table.

Syntax

moveToRecNo (const ***recordNum*** LongInt) Logical

Description

Sets the current record to the record *recordNum*. It returns an error if *recordNum* is not in the table. Use the method **nRecords** or examine the NRecords property to find out how many records a table contains. This method is recommended only for dBASE tables.

Example

This example moves to the midpoint of a table. Assume that a form contains a table frame bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecNo(halfWay)

endmethod
```

See also

- moveToRecord, resync, nRecords

moveToRecord

UIObject

Method

Moves to a specific record in a table.

Syntax

1. moveToRecord (const ***recordNum*** LongInt **)** Logical
2. moveToRecord (const ***tc*** TCursor **)** Logical

Description

Sets the current record to the record *recordNum*, or to the record pointed to by the TCursor *tc*. It returns an error if *recordNum* is greater than the number of records in the table. Use the **nRecords** method or examine the NRecords property to find out how many records a table contains. This method can be very slow for dBASE tables; use **moveToRecNo** instead.

Example

For an example of how to use **moveToRecord** with a TCursor, see **moveTo**.

This example moves to the midpoint of a table. Assume that a form contains a table frame bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endmethod
```

See also

- moveTo, moveToRecNo, nRecords, resync

nextRecord

Beginner

UIObject

Method

Moves to the next record in a table.

Syntax

nextRecord ( ) Logical

Description

Sets the current record to the next record in a table. It returns an error if the pointer is already at the last record.

nextRecord has the same effect as the action constant DataNextRecord, so the following statements are equivalent:

```
obj.nextRecord()

obj.action(DataNextRecord)
```

Example

This example moves to the next record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToNext* is a button on the same form.

```
; moveToNext::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atLast() then
  CUSTOMER.nextRecord()  ; move to the next record
  ; same as:  CUSTOMER.action(DataNextRecord)
  msgInfo("What record?", CUSTOMER.recno)
else
  msgInfo("Status", "Already at the last record.")
endif
endmethod
```

See also

- end, home, moveToRecord, priorRecord

nFields

UIObject

Method

Returns the number of fields in a table.

Syntax

nFields ( ) LongInt

Description

Returns the number of fields in a table.

Note To find the number of columns displayed in an object bound to a table, examine the value of the NCols property for that object.

Example

The following example reports on the number of fields and key fields in the *LineItem* table. Assume that a form has a table frame named *LINEITEM* bound to the *LineItem* table, and a button named *tableStats*.

```
; tableStats::pushButton
method pushButton(var eventInfo Event)
msgInfo("Status", "The LineItem table has " +
        String(LINEITEM.nFields()) + " fields and " +
        String(LINEITEM.nKeyFields()) + " key fields." +
        "\nThere are " + String(LINEITEM.NCols) +
        " columns in the table frame.")
endmethod
```

See also

- nKeyFields, nRecords

nKeyFields

UIObject

Method

Returns the number of key fields in a table.

Syntax

nKeyFields ( ) LongInt

Description

Returns the number of key fields in a table.

Example

See the example for nFields.

See also

❐ nFields, nRecords

nRecords

Beginner

UIObject

Method

Returns the number of records in a table.

Syntax

nRecords ( ) LongInt

Description

Returns the number of records in a table. This operation can take a long time for dBASE tables. You can also examine the NRecords property for an object to find the number of records in the table bound to that object.

The **nRecords** method and the NRecords property respect the limits of restricted views. If a table-based object is the detail table in a one-to-many relationship, **nRecords** reports the number of linked detail records, not the total number of records in the entire table.

Example

This example moves to the midpoint of a table. Assume that a form contains a table frame named *LINEITEM* bound to the *LineItem* table, and a button called *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
  halfWay  LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endmethod
```

See also

❐ moveToRecord, nFields, nKeyFields

UIObject

pixelsToTwips

UIObject

Method

Converts screen coordinates from pixels to twips.

Syntax

pixelsToTwips (const ***pixels*** Point) Point

Description

Converts the screen coordinates specified in *pixels* from pixels to twips. A pixel (the name comes from *pic*ture *el*ement) is a dot on the screen, and a twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).

Example

This example assumes that a form contains a two-inch square box named *twoSquare*. The *twoSquare* box contains two text boxes: *pixNum* to display the width of the box in pixels and *twipNum* to display the width in twips.

```
; twoSquare::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  twTopLeft,              ; top left point in twips
  twBottomRight,          ; bottom right point in twips
  pxTopLeft,              ; top left in pixels
  pxBottomRight,          ; bottom right in pixels
  selfPos      Point      ; current position property
endvar
self.getBoundingBox(twTopLeft, twBottomRight)    ; returns points in twips
twipNum.Text = twBottomRight.x() - twTopLeft.x() ; get the width in twips
pxTopLeft = TwipsToPixels(twTopLeft)             ; convert to pixels
pxBottomRight = TwipsToPixels(twBottomRight)
pixNum.Text = pxBottomRight.x() - pxTopLeft.x()  ; get the width in pixels
; cross check
twTopLeft = PixelsToTwips(pxTopLeft)      ; convert from pixels back to twips
twTopLeft.view("Top left in twips")       ; twTopLeft should match selfPos

selfPos = self.Position                   ; get selfPos, twips by default
selfPos.view("Position of box in twips")  ; show the result
endmethod
```

See also

- twipsToPixels

postAction

UIObject

Method

Posts an action to an action queue for delayed execution.

Syntax

postAction (const ***actionId*** SmallInt)

Description

Works like **action**, except that the action is not executed immediately. Instead, Paradox waits until the entire method has finished executing and Paradox is in a steady state. The action specified by *actionID* is

posted to an action queue at the time of the method call; Paradox performs the action after the current method has finished executing.

See also

- action
- postAction in the Form type
- The discussion of actions in Chapter 2

postRecord

UIObject

Beginner

Method

Posts a pending record to a table.

Syntax

postRecord ( ) Logical

Description

Returns True if the current record is successfully posted to the underlying table; otherwise, it returns False. **postRecord** does not unlock a locked record.

postRecord has the same effect as the action constant DataPostRecord, so the following statements are equivalent:

```
obj.postRecord()

obj.action(DataPostRecord)
```

Example

This example locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. The method changes the record and posts it with **postRecord**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
  obj  UIObject
endVar
obj.attach(CUSTOMER)
obj.locate("Name", "Sight Diver")
if thisForm.Editing then
  if NOT obj.lockRecord() then
     msgStop("Lock failed", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
  else
     msgStop("Lock succeeded", "recordStatus(\"Locked\") is " +
             String(obj.recordStatus("Locked")))
     obj.custRec."Name" = "Right Diver"  ; quotes on Name indicates
                                         ; field name instead of property
     obj.postRecord()
     message("Record is locked: ", obj.custRec.locked)
  endif
else
```

UIObject

```
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endif
endmethod
```

See also

- lockRecord, recordStatus
- attachToKeyViol in the TCursor type

priorRecord

UIObject

Beginner

Method

Moves to the previous record in a table.

Syntax

priorRecord ( ) Logical

Description

Sets the current record to the previous record in a table. It returns an error if the pointer is already at the first record.

priorRecord has the same effect as the action constant DataPriorRecord, so the following statements are equivalent:

```
obj.priorRecord()
```

```
obj.action(DataPriorRecord)
```

Example

This example moves to the prior record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form; *moveToPrior* is a button on the same form.

```
; moveToPrior::pushButton
method pushButton(var eventInfo Event)
if NOT CUSTOMER.atFirst() then
  CUSTOMER.priorRecord()  ; move to the previous record
  ; same as CUSTOMER.action(DataPriorRecord)
  msgInfo("What record?", CUSTOMER.recno)
else
  msgInfo("Status", "Already at the first record.")
endif
endmethod
```

See also

- currRecord, end, home, moveToRecord, nextRecord, skip

pushButton

UIObject

Method

Generates a **pushButton** Event and sends it to an object.

Syntax

pushButton **( )** Logical

Description

Constructs an event and calls the built-in **pushButton** method of an object with that event.

Example

The following example sends a **pushButton** event to *buttonTwo* on the same form:

```
; sendPushButton::pushButton
method pushButton(var eventInfo Event)
; send a pushButton to buttonTwo
buttonTwo.pushButton()
endmethod
```

See also

- mouseClick, mouseUp

recordStatus

UIObject

Method

Reports about the status of a record.

Syntax

recordStatus **(** const ***statusType*** String **)** Logical

Description

Returns True or False to a question about the status of a record. Use the argument *statusType* to specify the status in question, where *statusType* is "New", "Locked", or "Modified".

"New" means the record has just been inserted into the table. "Locked" means a lock (implicit or explicit) has been placed on the record. "Modified" means at least one of the field values has been changed. You can obtain similar information about the current record by examing the Inserting, Locked, Focus, and Touched properties for the record.

Example

This example locates a record, attempts to lock it with **lockRecord**, then checks the status of the lock with **recordStatus**. The method changes the record and unlocks it with **unlockRecord**. Assume that a form contains a table frame bound to the *Customer* table, and a button named *lockButton*.

```
; lockButton::pushButton
method pushButton(var eventInfo Event)
var
```

```
  Cust   UIObject                    ; to attach to table frame
  newKey Number
endVar

Cust.attach(CUSTOMER)                ; attach to CUSTOMER table frame
Cust.locate("Name", "Sight Diver")   ; find the record
if NOT thisForm.editing then         ; check if form is in Edit mode
  msgInfo("Status", "You must be in Edit mode for this operation.")
else
  if NOT Cust.lockRecord() then      ; try to lock the record
     msgStop("Status", "Lock Failed. recordStatus(\"Locked\") is " +
             String(Cust.recordStatus("Locked")))
  else
     msgInfo("Record locked?", Cust.recordStatus("Locked"))
     newKey = 1384
     Cust.custRec."Customer No" = newKey    ; change the key value
     msgInfo("Record modified?", Cust.recordStatus("Modified"))
     Cust.unlockRecord()             ; try to unlock the record-if it
                                     ; causes a keyviol, Paradox
                                     ; leaves record locked
     if Cust.recordStatus("Locked") then


       msgInfo("Status", "Record was a key violation. Changing key.")
       newKey = 1451
       Cust.custRec."Customer No" = newKey ; change to a new key
       Cust.postRecord()                   ; post it
       ; record will "fly away" to a new position based on key
     endif
     Cust.locate("Customer No", newKey)   ; find the "fly away"
  endif
endif
endmethod
```

See also

- lockRecord, unlockRecord

---

resync

UIObject

Method

Resynchronizes an object to a TCursor.

Syntax

resync (const ***tc*** TCursor) Logical

Description

Changes the current record pointer of a UIObject to the current record of the TCursor *tc*. When you resynchronize a table object to a TCursor, the table's filters and indexes will be changed to those of the TCursor. (For dBASE tables, the table will also take the Show Deleted setting of the TCursor.)

Example

See the example for insertRecord.

See also

- attach, moveToRecord

rgb

UIObject

Procedure

Defines a color.

Syntax

rgb (const ***red*** SmallInt, const ***green*** SmallInt, const ***blue*** SmallInt) LongInt

Description

Defines a color based on the values of *red*, *green*, and *blue*, which can be integers ranging from 0 to 255, or constants (for example, LightBlue).

The color constants are listed in the Constants dialog under Colors.

Example

The following example uses **rgb** to set the color of boxes as they're created. The method creates a color palette. Assume that the titles exist on the form in the appropriate locations. The form has one button, named **showPalette**.

```
; drawPalette::pushButton
method pushButton(var eventInfo Event)
var
  palAr Array[5] SmallInt   ; array to hold rgb values
  setBaseX       LongInt    ; base position
  setBaseY       LongInt    ; base position
  ui             UIObject   ; handle to create boxes
endVar
const
  horizInc = 1440           ; amount to move horizontally (twips)
  vertInc = 1080            ; amount to move vertically
endConst

palAr[1] = 0
palAr[2] = 64
palAr[3] = 128
palAr[4] = 192
palAr[5] = 255

for i from 1 to palAR.size()        ; reds(diagonal position)
  setBaseX = 720 + ((i - 1) * 150)  ; change base as i increases
  setBaseY = 720 + ((i - 1) * 150)
  for j from 1 to palAR.size()      ; greens (vertical positioning)
    for k from 1 to palAR.size()    ; blue   (horizontal positioning)
      ui.create(boxTool, setBaseX + (horizInc * (k - 1)),
                setBaseY + (vertInc * (j - 1)), 250, 250)
      ; set the color using rgb and values from array
      ui.Color = rgb(palAr[i], palAr[j], palAr[k])
      ui.Visible = Yes
    endfor                          ; k (blue, horizontal)
  endfor                            ; j (green, vertical)
endfor                              ; i (red, diagonal)

endmethod
```

See also

- getProperty, getRGB, setProperty

setFilter

UIObject

Method

Sets the range of records a table object can point to.

Syntax

setFilter ([const ***exactMatchVal*** AnyType,] *
const ***minVal*** AnyType, const ***maxVal*** AnyType) Logical

Description

Specifies conditions for including a range of records. Records that meet the conditions are included, records that don't are filtered out. This operation fails if the current record cannot be committed or if the table object is not bound to a keyed table.

This method compares the criteria you specify with values in the corresponding fields of a table's index. To filter records based on the value of a single field, specify values in *minVal* and *maxVal*. For example, the following statement checks values in the first field of the index of each record. If a value is less than 14 or greater than 88, that record is filtered out.

```
tblObj.setFilter(14, 88)
```

To specify an exact match on a single field, assign *minVal* and *maxVal* the same value. For example, the following statement filters out all values except 55:

```
tblObj.setFilter(55, 55)
```

You can filter records based on the values of more than one field. To do so, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on "Borland" and "Paradox" (assuming they're the first fields in the index), and values ranging from 100 to 500, inclusive, for the third field:

```
tblObj.setFilter("Borland", "Paradox", 100, 500)
```

Calling **setFilter** without any arguments resets the filter to include the entire table.

Example

For this example, assume that the first field in *Lineitem*'s key is Order No. When you press the *getDetailSum* button, the **pushButton** method limits the number of records included in the LINEITEM object to those with 1005 in the first key field.

```
; getDetails::pushButton
method pushButton(var eventInfo Event)
var
  tblObj  UIObject
endVar
if tblObj.attach(LINEITEM) then
```

```
  ; this limits tblObj's view to records that have
  ; 1005 as their key value (Order No. 1005).
  tblObj.setFilter(1005, 1005)
  ; now display the number of records for Order No. 1005
  msgInfo("Total records for order 1005", tblObj.nRecords())
else
  msgStop("Sorry", "Can't attach to table.")
endif
endmethod
```

See also

- switchIndex
- setFilter in the TCursor type

setPosition

UIObject

Method

Sets the position of an object.

Syntax

setPosition (const ***x*** LongInt, const ***y*** LongInt, const ***w*** LongInt, const ***h*** LongInt)

Description

Sets the position of an object on the screen. Variables *x* and *y* specify the coordinates (in twips) of the upper left corner of the object. Variables *w* and *h* specify the width and height (in twips) of the object. If the object is not specified, self is implied.

You can also set and examine an object's position and size with the Position and Size properties. For instance,

```
self.Position = Point(100, 150)
self.Size = Point(2000, 2500)
```

is the same as

```
self.setPosition(100, 150, 2000, 2500)
```

Example

The following example moves a circle across the screen in response to TimerEvents. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a timer event every 100 milliseconds. Each timer event causes *toggleButton's* **timer** method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

Following is the code for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
; label for button was renamed to buttonLabel
if buttonLabel = "Start Timer" then   ; if stopped, then start
```

```
  buttonLabel = "Stop Timer"           ; change label
  self.setTimer(10)                    ; start the timer
else                                   ; if started, then stop
  buttonLabel = "Start Timer"          ; change label
  self.killTimer()                     ; stop the timer
endif

endmethod
```

Following is the code for *toggleButton*'s **timer** method:

```
; toggleButton::timer
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar
ui.attach(floatCircle)           ; attach to the circle
ui.getPosition(x, y, w, h)       ; assign coordinates to vars
if x  4320 then                  ; if not at left edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the left
else
  ui.setPosition(1440, y, w, h)     ; return to the right
endif
endmethod
```

See also

- getPosition

setProperty

UIObject

Method

Sets a property to a specified value.

Syntax

setProperty (const ***propertyName*** String,
const ***propertyValue*** AnyType)

Description

Sets the *propertyName* property of an object to *propertyValue*. If the object does not have a property *propertyName*, or if **propertyValue** is invalid, an error results.

setProperty is an alternative to setting a property directly; it's useful when *propertyName* is a variable. Otherwise, access the property directly, as in

```
myBox.Color = Red
```

Example

The following example creates a dynamic array, indexed by property names, to contain property values. The array is filled by using the array's index as the argument to the **getProperty** command. The method changes one of the values of the properties and resets the object's properties from the dynamic array with the **setProperty** method.

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  propNames DynArray[] AnyType    ; to hold property names & values
  arrayIndex          String     ; index to dynamic array
endVar

propNames["Color"] = ""
propNames["Visible"] = ""
propNames["Name"] = ""

foreach arrayIndex in propNames
  propNames[arrayIndex] = self.getProperty(arrayIndex)
endforeach

propNames["Color"] = "DarkBlue"

foreach arrayIndex in propNames
  self.setProperty(arrayIndex, propNames[arrayIndex])
endforeach

endmethod
```

See also

- getProperty, getPropertyAsString

setTimer

UIObject

Method

Starts the timer for an object.

Syntax

setTimer (const ***milliSeconds*** LongInt [, const ***repeat*** Logical])

Description

Starts a timer for an object. The timer interval is specified using *milliSeconds*. The optional argument *repeat* specifies if the timer automatically repeats. If repeat is True or omitted, the timer repeats; otherwise, the TimerEvent is sent once. Usually, **setTimer** is attached to an object's **open** method, and the object's response is defined in its **timer** method.

Note Windows allows a maximum of 16 timers for all applications. However, Paradox has no limit. System resources may limit the number of timers you can set, and you may run out of Windows timers, but Paradox is not restricted by the 16-timer limit.

Example

The following example moves a circle across the screen in response to timer events. The **pushButton** method for *toggleButton* uses **setTimer** and **killTimer** to start or stop a timer, depending on the condition of the button. When the timer starts, it issues a timer event every 100 milliseconds. Each timer event causes *toggleButton's* **timer** method to execute. The **timer** method gets the current position of the ellipse with **getPosition**, then moves it 100 twips to the right with **setPosition**.

The following code is for *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then   ; if stopped, then start
  buttonLabel = "Stop Timer"          ; change label
  self.setTimer(10)                   ; start the timer
else
  buttonLabel = "Start Timer"         ; change label
  self.killTimer()                    ; stop the timer
endif

endmethod
```

The following code is for *toggleButton*'s **timer** method:

```
; toggleButton::timer
method timer(var eventInfo TimerEvent)
var
  ui          UIObject
  x, y, w, h  SmallInt
endVar
ui.attach(floatCircle)        ; attach to the circle
ui.getPosition(x, y, w, h)    ; assign coordinates to vars
if x  4320 then               ; if not at left edge of area
  ui.setPosition(x + 100, y, w, h)  ; move to the left
else
  ui.setPosition(1440, y, w, h)     ; return to the right
endif
endmethod
```

See also

- killTimer

skip

UIObject

Method

Moves forward or backward a specified number of records in a table.

Syntax

skip (const ***nRecords*** LongInt) Logical

Description

Sets the current record to the record *nRecords* from the current record. You'll get an error if **skip** tries to move beyond the limits of the table.

Positive values for *nRecords* move forward through the table (*nRecords* = 1 is the same as **nextRecord**), negative values move backward (*nRecords* = -1 is the same as **priorRecord**), and setting *nRecords* to 0 doesn't move (*nRecords* = 0 is the same as **currRecord**).

Example

The following example fills a table with a sampling of records from the *Orders* table. Assume that the table *SampOrd* already exists with the same structure as *Orders*. The *createSampling* button, whose **pushButton** method is shown below, exists on a form along with a table frame bound to *Orders*. The method moves the pointer through

the *Orders* table, skips a random number of records, and copies the record it lands on to the sampling table.

```
; createSampling::pushButton
method pushButton(var eventInfo Event)
var
  ordSampleTC        TCursor       ; handle to sampling table
  copyRec Array[]    String        ; holds record copied from Orders
  randInt            SmallInt      ; random number to skip
  OrdObj             UIObject      ; handle to Orders
endVar

ordObj.attach(ORDERS)              ; attach to ORDERS table frame
ordObj.home()                      ; move to the first record
if ordSampleTC.open("OrdSamp.db")  then
  ordSampleTC.empty()              ; clear out sampling table
  ordSampleTC.edit()               ; start editing
  while NOT OrdObj.atLast()
    randInt = int(rand() * 20) + 1  ; create an integer between 1 and 20
    randInt.view()                 ; show the number
    OrdObj.skip(randInt)           ; skip a random number of records
    OrdObj.copyToArray(copyRec)    ; get the record
    ordSampleTC.insertRecord()     ; make a space for it
    ordSampleTC.copyFromArray(copyRec)  ; insert the record
  endwhile
  ordSampleTC.endEdit()            ; end editing
  msgInfo("Status", "OrdSamp table now has " +
         String(ordSampleTC.nRecords()) + " records.")
  ordSampleTC.close()              ; close it out
else
  msgStop("Oops", "Sorry. Couldn't find OrdSamp table.")
endif
endmethod
```

See also

- currRecord, end, home, moveToRecord, nextRecord, priorRecord

switchIndex

UIObject

Method

Specifies another index to use to view the records in a table.

Syntax

1. **switchIndex** ([const ***indexName*** String] [, const ***stayOnRecord*** Logical]) Logical
2. **switchIndex** ([const ***indexFileName*** String [, const ***tagName*** String]] [, const ***stayOnRecord*** Logical]) Logical

Description

Specifies in *indexName* an index file to use with a table. In syntax 1, *indexName* specifies an index to use with a Paradox table. Syntax 2 is for dBASE tables, where *indexFileName* can specify a .NDX file or a .MDX file, and optional argument *tagName* specifies an index tag in a production index (.MDX) file.

In both syntaxes, if optional argument *stayOnRecord* is Yes, this method maintains the current record after the index switch; if it is No, the first record in the table becomes the current record. If omitted, *stayOnRecord* is No by default.

Example

For this example, assume that *Customer* is a keyed Paradox table that has a secondary index named "NameAndState". This example attaches to a table frame bound to *Customer*, and calls **switchIndex** to switch from the pimary index to the "NameAndState" index.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  tblObj UIObject
endvar

tblObj.attach(CUSTOMER)                ; attach to Customer
tblObj.switchindex("NameAndState")     ; switch to index NameAndState
tblObj.home()                          ; make sure we're on the first record
msgInfo("First Record", tblObj.Name)   ; display value in Name field

endmethod
```

See also

- setFilter
- reIndex, reIndexAll in the TCursor type
- setIndex in the Table type

twipsToPixels

UIObject

Method

Converts screen coordinates from twips to pixels.

Syntax

twipsToPixels (const ***twips*** Point) Point

Description

Converts the screen coordinates specified in *twips* from twips to pixels. A pixel (the name comes from *pic*ture *el*ement) is a dot on the screen, and a twip is a device-independent unit equal to 1/1440 of an inch (1/20 of a printer's point).

Example

See the example for pixelsToTwips.

See also

- pixelsToTwips

unDeleteRecord

UIObject

Method

Undeletes the current record from a dBASE table.

Syntax

unDeleteRecord ( ) Logical

Description

Undeletes the current record of a dBASE table. This operation can be successful only if **showDeleted** has been set True, the current record is a deleted record, and the table object is in Edit mode.

See also

- deleteRecord, isRecordDeleted
- isShowDeletedOn, showDeleted in the Table type

unlockRecord

Beginner

UIObject

Method

Removes a write lock from the current record.

Syntax

unlockRecord ( ) Logical

Description

Returns True if it successfully removes an explicit write lock on the current record; otherwise, it returns False.

Note You can also examine the locked property to find out whether an object is locked, but you can't change the property to lock or unlock an object. The Locked property is a read-only property.

Example

See the example for recordStatus.

See also

- recordStatus, lockRecord
- attachToKeyViol, didFlyAway, setFlyAwayControl in the TCursor type

view

Beginner

UIObject

Method

Displays the value of an object in a dialog box.

Syntax

view ([const ***title*** String])

UIObject

Description

Displays the value of an object in a dialog box. Paradox suspends method execution until you close the dialog box. You have the option to specify, in *title*, a title for the dialog box. If you omit *title*, the title is the data type of the value.

This method works only with the following UIObjects:

- Buttons as checkboxes or radio buttons
- Unbound fields only as lists or radio buttons
- Fields bound to a table; the field's data type can be any data type except Memo and Graphic

Calling **view** with any other UIObject causes a run-time error.

Example

For this example, assume that a form contains a table frame, named *CUSTOMER* bound to the *Customer* table, and a button. The following code is attached to the button's **pushButton** method. It creates an array of seven UIObjects, then tries to view each item in the array.

```
;  page::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
   obj          UIObject
   arr Array[7] UIObject
   i            SmallInt
endVar
arr[1].attach(CUSTOMER.Phone) ; the Phone field (A15) in the table frame

                              ; shows the phone number
arr[2].attach(aGraphic)       ; a bitmap (invalid)
arr[3].attach(someText)       ; a text object (invalid)
arr[4].attach(someList)       ; an unbound list field
                              ; shows the list item selected

arr[5].attach(someUnField)    ; an unbound field (invalid)
arr[6].attach(someRadio)      ; an unbound field as a radio button

                              ; shows the value of the active radio button
arr[7].attach(someButton)     ; an unbound field as a checkbox

                              ; True if checked, otherwise False
for i from 1 to arr.size()
  arr[i].view(arr[1].Class + ": Item " + String(i))
endFor
endmethod
```

See also

- attach

wasLastClicked

UIObject

Method

Tells if an object was the last object to receive a mouse click.

Syntax

wasLastClicked ( ) Logical

Description

Returns True if an object was the last object to receive a mouse click; otherwise, it returns False. This method can be used only with objects in the current form.

Example

The following code is attached to the **mouseUp** method for an object called *boxOne*. If *boxOne* receives the click, the message appears; if *boxOne* was sent a **mouseUp** event from another object, the method beeps instead.

The following code is for *boxOne*'s **mouseUp** method:

```
; boxOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
if self.wasLastClicked() then
  msgInfo("Hey!", "Quit clicking me.")  ; method invoked by clicking
else
  beep()                                ; method invoked indirectly
endif
endmethod
```

This code is for *sendAClick*'s **mouseUp** method:

```
; sendAClick::mouseUp
method mouseUp(var eventInfo MouseEvent)
boxOne.mouseUp(eventInfo)  ; when boxOne's mouseUp gets this,
                           ; it will beep
endmethod
```

See also

- wasLastRightClicked, hasMouse

wasLastRightClicked

UIObject

Method

Tells if an object was the last object to receive a right mouse click.

Syntax

wasLastRightClicked ( ) Logical

Description

Returns True if an object was the last object to receive a right mouse click; otherwise, it returns False. This method can be used only with objects in the current form.

Example

The following is attached to the **mouseRightUp** method for an object called *circleOne*. If the ellipse received the right click, the message displays; if the ellipse was sent a **mouseRightUp** event from another object, the method beeps instead.

This is the code for *circleOne*'s **mouseUp** method:

```
; circleOne::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
if self.wasLastRightClicked() then
  ; method invoked by right-click
  msgInfo("Right-click", "Go click on someone your own size.")
else
  beep()       ; method invoked indirectly
endif
endmethod
```

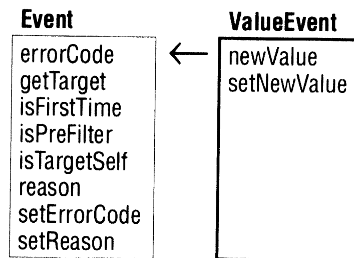
The following code is for *sendARightClick*'s **mouseUp** method:

```
; sendARightClick::mouseRightUp
method mouseRightUp(var eventInfo MouseEvent)
circleOne.mouseRightUp(eventInfo)  ; when circleOne gets this,
                                   ; it will beep
endmethod
```

See also

- wasLastClicked, hasMouse

ValueEvent



ValueEvent methods control field value changes. The **changeValue** built-in method is the only method triggered by a ValueEvent. The built-in **newValue** method is not called with a ValueEvent; **newValue** takes an Event instead. All the built-in methods are discussed in Chapter 2.

The ValueEvent type includes several methods defined for the Event type.

Do not confuse **changeValue** with **newValue**. The built-in **changeValue** method is called when the value of a field is about to change. **changeValue** gives you a chance to check the value and decide whether you want to post it. The built-in **newValue** method reports when a field has received a new value; **newValue** is usually called after the fact (fields defined as buttons and lists behave differently). Also note that the built-in **newValue** method is *not* the same as the **newValue** method described in this section.

For more information and examples, refer to Chapter 6 in the *ObjectPAL Developer's Guide.*

newValue

ValueEvent

Beginner

Method

Returns the unposted new value of a ValueEvent.

Syntax

newValue ( ) AnyType

Description

Returns the new value to be assigned to a field for a ValueEvent. The new value is not yet assigned to the field so the following two statements may return different values:

```
field.Value
```

```
eventInfo.newValue()
```

Example

In this example, the **changeValue** method for the *creditLimit* field checks the old value and the new value to see if there is more than a 25% change. If the difference between the old and new values is too large, the method blocks the change. Assume that *creditLimit* is an unbound field on a form, and that there is at least one other field to move to.

```
; creditLimit::changeValue
method changeValue(var eventInfo ValueEvent)
var
  oldVal,
  newVal   Number
endVar
oldVal = self.Value            ; the property may be different
newVal = eventInfo.newValue()    ; than the new value
if newVal > oldVal AND oldVal <> 0 then
  if (newVal - oldVal)/oldVal > 0.25 then
    msgStop("Stop", "You are not allowed to increase the " +
                "credit limit more than 25%.")
    self.action(EditUndoField)  ; --use this to restore old value
    eventInfo.setErrorCode(CanNotDepart)   ; block departure
  endif
endif
endmethod
```

See also

❒ setNewValue

setNewValue

ValueEvent

Method

Specifies a value to set for a ValueEvent.

Syntax

setNewValue (const ***newValue*** AnyType)

Description

Specifies in *newValue* a value to set for a ValueEvent. The data supplied in *newValue* should be consistent with the field's type.

Example

In this example, assume a form contains the field *authorAbbrToName*, as well as at least one other field. When the user enters an author abbreviation, then moves off the field, the **changeValue** method fills in the full author name.

```
; authorAbbrToName::changeValue
method changeValue(var eventInfo ValueEvent)
var
  abbrValue String
  fullValue String
endVar

abbrValue = upper(eventInfo.newValue())  ; get the value and convert
                                         ; to uppercase
; user enters an abbreviation--change to full name
switch
  case abbrValue = "SG" : fullValue = "Susan Grafton"
```

```
  case abbrValue = "SP" : fullValue = "Sara Paretsky"
  case abbrValue = "MHC": fullValue = "Mary Higgins Clark"
  case abbrValue = "FK" : fullValue = "Faye Kellerman"
  case abbrValue = "AF" : fullValue = "Antonia Fraser"
  case abbrValue = "AC" : fullValue = "Agatha Christie"
  otherwise : fullValue = "Author Unknown"
endswitch

eventInfo.setNewValue(fullValue)
endmethod
```

See also

- newValue

ValueEvent

PART II

Appendixes

This part of the *ObjectPAL Reference* contains appendixes with reference material frequently needed by ObjectPAL programmers.

This part consists of the following appendixes:

- Appendix A, "ObjectPAL methods by type"
- Appendix B, "ANSI character set"
- Appendix C, "Windows keycodes"
- Appendix D, "Keywords"
- Appendix E, "Compatibility functions"
- Appendix F, "Properties"
- Appendix G, "Constants"

APPENDIX A

ObjectPAL methods by type

Table A-1 ObjectPAL methods by type

Type	Method	Description
ActionEvent	actionClass	Returns the class number of an ActionEvent
	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	id	Returns the ID number of an ActionEvent
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setId	Specifies an event
	setReason	Specifies a reason for generating an event
AnyType	blank	Returns a blank value
	dataType	Returns a string representing the data type of a variable
	isAssigned	Reports whether a variable has been assigned a value
	isBlank	Reports whether an expression has a blank value
	isFixedType	Reports whether a variable's data type has been explicitly declared
	unAssign	Sets a variable's state to unAssigned
	view	Displays in a dialog box the value of a variable
Application	bringToTop	Brings the window to the top of the display stack
	getPosition	Reports the position of a window onscreen
	getTitle	Returns the text of the window title bar
	hide	Makes a window invisible
	isMaximized	Reports whether a window is displayed at its maximum size
	isMinimized	Reports whether a window is displayed as an icon
	isVisible	Reports whether any part of a window is displayed

Type	Method	Description
	maximize	Maximizes a window
	minimize	Minimizes a window
	setPosition	Positions a window on the screen
	setTitle	Sets the text in the window title bar
	show	Displays a minimized window at its original size
	windowClientHandle	Returns the handle of a client window
	windowHandle	Returns the handle of a window
Array	addLast	Inserts an element at the end of a resizeable array
	append	Appends the contents of one array to another
	contains	Searches the items of an array for a pattern of characters
	countOf	Counts the occurrences of a value in an array
	empty	Removes all items from an array
	exchange	Swaps the contents of two cells in an array
	fill	Fills an array with a value
	grow	Increases the size of a resizeable array
	indexOf	Returns the position of an element in an array
	insert	Inserts one or more empty cells into a resizeable array
	insertAfter	Inserts an item into an array after a specified item
	insertBefore	Inserts an item into an array before a specified item
	insertFirst	Inserts an element at the beginning of an array
	isResizeable	Reports whether an array can be resized
	remove	Removes one or more items from an array
	removeAllItems	Removes all occurrences of an array item
	removeItem	Deletes a specified item from an array
	replaceItem	Overwrites an item in an array with another item
	setSize	Specifies the size of an array
	size	Returns the number of items in an array
	view	Displays in a dialog box the contents of an array
Binary	readFromFile	Reads data from a file and stores it in a Binary variable
	size	Returns the number of bytes in a Binary variable
	writeToFile	Writes the data stored in a Binary variable to a disk file
Currency	Currency	Casts a value as Currency
	view	Displays in a dialog box the value of a variable
Database	close	Closes a database
	delete	Deletes a table from a database
	executeQBE	Executes a QBE query
	executeQBEFile	Opens and executes a QBE file
	executeQBEString	Executes a QBE string
	isAssigned	Reports whether a Database variable has been assigned a value

Type	Method	Description
	isTable	Reports whether a table exists in a database
	open	Opens a database
	writeQBE	Writes a query statement of a query string to a file
Date	date	Casts a value as a Date data type
	dateVal	Converts a value to a date
	day	Extracts the day of the month from a date
	daysInMonth	Returns the number of days in a month
	dow	Returns the day of the week of a date
	dowOrd	Returns the number of a day of the week
	doy	Returns the number of a day of the year
	isLeapYear	Reports whether a year has 366 days
	month	Extracts the month from a date as a number
	moy	Extracts the month from a date as a string
	today	Returns the current date
	view	Displays in a dialog box the value of a variable
	year	Extracts the year from a date
DateTime	dateTime	Casts a variable as a DateTime data type
	day	Extracts the day of the month from a date
	daysInMonth	Returns the number of days in a month
	dow	Returns the day of the week of a date
	dowOrd	Returns the number of a day of the week
	doy	Returns the number of a day of the year
	hour	Extracts as a number the hour from a DateTime
	isLeapYear	Reports whether a year has 366 days
	milliSec	Extracts as a number the milliseconds from a DateTime
	minute	Extracts as a number the minutes from a DateTime
	month	Extracts the month from a date as a number
	moy	Extracts the month from a date as a string
	second	Extracts as a number the seconds from a DateTime
	view	Displays in a dialog box the value of a variable
	year	Extracts the year from a date
DDE	close	Closes a DDE link
	execute	Sends a command via a DDE link
	open	Opens a DDE link to another application
	setItem	Specifies an item in a DDE conversation
DynArray	contains	Searches a DynArray for a pattern of characters
	empty	Removes all items from a dynamic array
	getKeys	Loads a resizeable array with the indexes of an existing DynArray
	removeItem	Deletes a specified item from a DynArray

Type	Method	Description
	size	Returns the number of elements in a DynArray
	view	Displays the contents of a DynArray in a dialog box
ErrorEvent	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setReason	Specifies a reason for generating an event
Event	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setReason	Specifies a reason for generating an event
FileSystem	accessRights	Reports about access rights (also called file attributes) for a file
	copy	Copies a file
	delete	Deletes a file
	deleteDir	Deletes a directory
	drives	Returns the letters of the drives attached to the system and known to Windows
	enumFileList	Writes information about files to a table
	existDrive	Reports whether a drive is attached to the system
	findFirst	Searches a file system for a filename
	findNext	Searches a file system for multiple instances of a filename
	freeDiskSpace	Returns the amount of free space on a drive
	fullName	Returns the full path of a file
	getDir	Returns a directory path that the FileSystem variable is pointing to
	getDrive	Returns the drive letter pointed to by the FileSystem variable
	getFileAccessRights	Reports about access rights (also called file attributes) for a file
	getValidFileExtensions	Returns the valid file extensions for a specified object
	isDir	Reports whether a specified string represents the name of a directory
	isFile	Reports whether a specified string is the name of a file in the current file system

Type	Method	Description
	isFixed	Reports whether a drive is fixed
	isRemote	Reports whether a drive is remote (a network drive)
	isRemovable	Reports whether a drive is removable
	makeDir	Creates a new directory
	name	Returns the name of a file
	privDir	Returns the name of the user's private directory
	rename	Renames a file
	setDir	Sets the directory path for a FileSystem variable
	setDrive	Makes a specified drive the default drive
	setFileAccessRights	Sets access rights (also called attributes) of a file
	size	Returns the size of a file
	splitFullFileName	Breaks a full path name into its component parts
	startUpDir	Returns a string containing the path to the user's start-up directory
	time	Returns the time and date a file was last modified
	totalDiskSpace	Returns the capacity of a drive
	windowsDir	Returns the path to the WINDOWS directory
	windowsSystemDir	Returns the path to the WINDOWS\SYSTEM directory
	workingDir	Returns the name of the current working directory
Form	action	Performs an action command
	attach	Associates a Form variable with an open form
	bringToTop	Brings the window to the top of the display stack and makes it active
	close	Closes a window
	create	Creates a blank form in a design window
	delayScreenUpdates	Turns delayed screen updates on or off
	deliver	Delivers a form
	design	Switches a running form to a design window
	disableBreakMessage	Prevents program interruption by Ctrl+Break
	dmAddTable	Adds a table to a form's data model
	dmGet	Retrieves a field value from a table in the data model
	dmHasTable	Reports whether a table is part of a form's data model
	dmPut	Writes data to a table in a form's data model
	dmRemoveTable	Removes a table from a form's data model
	enumSource	Creates a table listing the methods for each object in a form
	enumSourceToFile	Creates a file listing the methods for each object in a form
	enumTableLinks	Creates a table listing the tables linked to a form
	enumUIObjectNames	Creates a table listing the UIObjects contained in a form
	enumUIObjectProperties	Creates a table listing the properties of each UIObject contained in a form

Type	Method	Description
	formCaller	Returns a handle to the calling form
	formReturn	Returns control to a suspended method
	getPosition	Reports the position of a window onscreen
	getTitle	Returns the text of the window title bar
	hide	Makes a window invisible
	hideSpeedBar	Makes the SpeedBar invisible
	isDesign	Reports whether a form is in Design mode
	isMaximized	Reports whether a window is displayed at its maximum size
	isMinimized	Reports whether a window is displayed as an icon
	isSpeedBarShowing	Reports whether the SpeedBar is visible
	isVisible	Reports whether any part of a window is displayed
	keyChar	Sends an event to a form's **keyChar** method
	keyPhysical	Sends an event to a form's **keyPhysical** method
	load	Opens a form in the Form Design window
	maximize	Maximizes a window
	menuAction	Sends an event to a form's **menuAction** method
	methodDelete	Deletes a form-level method from a form
	methodGet	Gets a form-level method
	methodSet	Sets the definition of a method attached to a form
	minimize	Minimizes a window
	mouseDouble	Calls a form's **mouseDouble** method
	mouseDown	Calls a form's **mouseDown** method
	mouseEnter	Calls a form's **mouseEnter** method
	mouseExit	Calls a form's **mouseExit** method
	mouseMove	Calls a form's **mouseMove** method
	mouseRightDouble	Calls a form's **mouseRightDouble** method
	mouseRightDown	Calls a form's **mouseRightDown** method
	mouseRightUp	Calls a form's **mouseRightUp** method
	mouseUp	Calls a form's **mouseUp** method
	moveTo	Moves to a form
	moveToPage	Displays a specified page of a form
	open	Opens a window
	openAsDialog	Opens a window as a dialog box
	postAction	posts an action to an action queue for delayed execution
	run	Switches a form from the Form Design mode to the View Data window
	save	Saves a form to disk
	setPosition	Positions a window on the screen
	setTitle	Sets the text in the window title bar

Type	Method	Description
	show	Displays a minimized window at its previous size
	showSpeedBar	Makes the SpeedBar visible
	wait	Suspends execution of a method
	windowClientHandle	Returns the handle of a client window
	windowHanlde	Returns the handle of a window
Graphic	readFromClipboard	Reads a bitmap from the Clipboard
	readFromFile	Reads a bitmap from a file
	writeToClipboard	Writes a bitmap to the Clipboard
	writeToFile	Writes a bitmap to a file
KeyEvent	char	Returns the character associated with a keypress
	charAnsiCode	Returns the ANSI value associated with a keypress
	errorCode	Reports about the status of the error flag
	getTarget	Returns the name of the target of an event
	isAltKeyDown	Reports whether *Alt* was held down during a KeyEvent
	isControlKeyDown	Reports whether *Ctrl* was held down during a KeyEvent
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isShiftKeyDown	Reports whether *Shift* was held down during a KeyEvent
	isTargetSelf	Reports whether an object is the target of a KeyEvent
	reason	Reports the reason for an event
	setAlyKeyDown	Simulates pressing and holding *Alt* during a KeyEvent
	setChar	Specifies an ANSI character to send as the result of a KeyEvent
	setControlKeyDown	Simulates pressing and holding *Ctrl* during a KeyEvent
	setErrorCode	Sets the error code for a KeyEvent
	setReason	Specifies a reason for generating an event
	setShiftKeyDown	Simulates pressing and holding *Shift* during a KeyEvent
	setVChar	Specifies a Windows virtual character for a KeyEvent
	setVCharCode	Specifies a Windows virtual character for a KeyEvent
	vChar	Returns a Windows virtual character
	vCharCode	Returns the integer value of a Windows virtual character
Library	close	Closes a library
	enumSource	Writes the code from a library to a Paradox table
	enumSourceToFile	Writes the code from a library to a text file
	execMethod	Calls a custom method that takes no arguments
	open	Associates a Library variable with a library and makes the library code available
Logical	logical	Casts a value as type Logical
	view	Displays in a dialog box the value of a variable

Type	Method	Description
LongInt	bitAND	Performs a bitwise AND operation on two values
	bitIsSet	Reports whether a bit is 1 or 0
	bitOR	Performs a bitwise OR operation on two values
	bitXOR	Performs a bitwise XOR operation on two values
	LongInt	Casts a value as a LongInt
	view	Displays the value of a variable in a dialog box
Memo	memo	Casts a value as a Memo
	readFromFile	Reads a memo from a file
	writeToFile	Writes a memo to a file
Menu	addArray	Appends elements of an array to a menu
	addBreak	Starts a new row in a menu
	addPopUp	Adds a pop-up menu to a menu bar item
	addStaticText	Adds an unselectable text string to a menu
	addText	Adds a selectable text string to a menu
	contains	Tells whether an item is in a menu
	count	Returns the number of items in a menu
	empty	Removes all items from a menu
	getMenuChoiceAttribute	Reports the display attribute of a menu item
	getMenuChoiceAttributeById	Reports the display attribute of a menu item specified by its menu ID
	hasMenuChoiceAttribute	Reports whether a menu item contains a given display attribute
	remove	Removes an item from a menu
	removeMenu	Removes a custom menu and restores the default menu
	setMenuChoiceAttribute	Sets the display attribute of a menu item
	setMenuChoiceAttributeById	Sets the display attribute of a menu item specified by its menu ID
	show	Displays a menu
MenuEvent	data	Returns information about a MenuEvent
	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	id	Returns the ID of a MenuEvent
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isFromUI	Reports whether an event was generated by the user interacting with Paradox
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of a KeyEvent
	menuChoice	Returns a string containing an item chosen from a menu
	reason	Reports why an event occurred
	setData	Specifies information about a MenuEvent
	setErrorCode	Sets the error code for a MenuEvent

Type	Method	Description
	setId	Specifies the ID of a MenuEvent
	setReason	Specifies a reason for generating a MenuEvent
MouseEvent	errorCode	Reports about the status of the error flag
	getMousePosition	Returns the mouse position as a Point
	getObjectHit	Returns a handle to the UIObject that received the event
	getTarget	Returns the name of the target of an event
	isControlKeyDown	Reports whether *Ctrl* is held down during a MouseEvent
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isFromUI	Reports whether an event was generated by the user interacting with Paradox
	isInside	Reports whether the mouse is inside the border of the target object
	isLeftDown	Reports whether the left mouse button is held down during a MouseEvent
	isMiddleDown	Reports whether the middle mouse button is held down during a MouseEvent
	isRightDown	Reports whether the right mouse button is pressed during a MouseEvent
	isShiftKeyDown	Reports whether *Shift* is held down during a MouseEvent
	isTargetSelf	Reports whether an object is the target of a MouseEvent
	reason	Reports why an event occurred
	setControlKeyDown	Simulates pressing and holding *Ctrl* during a MouseEvent
	setErrorCode	Sets the error code for a MouseEvent
	setInside	Sets the mouse to be inside the current object
	setLeftDown	Simulates pressing the left mouse button
	setMiddleDown	Simulates pressing the middle mouse button
	setMousePosition	Sets the position of the mouse for an event
	setReason	Specifies a reason for generating a MouseEvent
	setRightDown	Simulates pressing the right mouse button
	setShiftKeyDown	Simulates pressing and holding *Shift*
	setX	Specifies the horizontal coordinate of the pointer position
	setY	Specifies the vertical coordinate of the pointer position
	x	Returns the horizontal coordinate of the pointer position
	y	Returns the vertical coordinate of the pointer position
MoveEvent	errorCode	Reports the status of the error flag
	getDestination	Reports which object is the destination of a move
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf

Type	Method	Description
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setReason	Specifies a reason for generating an event
Number	abs	Returns the absolute value of a number
	acos	Returns the 2-quadrant arc cosine of a number
	asin	Returns the 2-quadrant arc sine of a number
	atan	Returns the 2-quadrant arc tangent of a number
	atan2	Returns the 4-quadrant arc tangent of a number
	ceil	Rounds a numeric expression to the greatest whole number
	cos	Returns the cosine of an angle
	cosh	Returns the hyperbolic cosine of an angle
	exp	Returns the exponential (base e) of a number
	floor	Rounds a numeric expression to the least whole number
	fraction	Returns the fractional part of a number
	fv	Returns the future value of a series of equal payments
	ln	Returns the natural logarithm of a numeric expression
	log	Returns the base 10 logarithm of a numeric expression
	max	Returns the larger of two numbers
	min	Returns the smaller of two numbers
	mod	Returns the remainder when one number is divided by another
	number	Casts a value as a Number
	numVal	Converts a string to a number
	pmt	Returns the periodic payment required to pay off a loan
	pow	Raises a number to a power
	pow10	Calculates 10 to a specified power
	pv	Returns the present value of a series of equal payments
	rand	Generates a random value ranging from 0 to 1
	round	Rounds a number to a specifed number of decimal places
	sin	Returns the sine of an angle
	sinh	Returns the hyperbolic sine of an angle
	sqrt	Returns the square root of a number
	tan	Returns the tangent of an angle
	tanh	Returns the hyperbolic tangent of an angle
	truncate	Truncates a number to a specified number of decimal places
OLE	canReadFromClipboard	Reports whether an OLE object can be pasted from the Clipboard into an OLE variable
	edit	Launches the OLE server and lets the user edit the object or take some other action

Type	Method	Description
	enumVerbs	Creates a DynArray listing the actions supported by the OLE server
	getServerName	Reports the name of the OLE server for an OLE object
	readFromClipboard	Pastes an OLE object from the Clipboard into an OLE variable
	writeToClipboard	Copies an OLE variable to the Clipboard
Point	distance	Returns the distance between two points
	isAbove	Reports whether a point is above another point
	isBelow	Reports whether a point is below another point
	isLeft	Reports whether a point is to the left of another point
	isRight	Reports whether a point is to the right of another point
	point	Casts an expression as a point
	setX	Specifies the x-coordinate of a point
	setXY	Specifies the x- and y-coordinates of a point
	setY	Specifies the y-coordinate of a point
	x	Returns the x-coordinate of a point
	y	Returns the y-coordinate of a point
PopUpMenu	addArray	Appends elements of an array to a pop-up menu
	addBar	Adds a vertical bar to a pop-up menu
	addBreak	Starts a new column in a pop-up menu
	addPopUp	Adds a pop-up menu to the structure
	addSeparator	Adds a horizontal bar to a pop-up menu
	addStaticText	Adds an unselectable text string to a pop-up menu
	addText	Adds a selectable text string to a pop-up menu
	contains	Tells whether an item is in a pop-up menu
	count	Returns the number of items in a pop-up menu
	empty	Deletes all items from a pop-up menu
	remove	Deletes an item from a pop-up menu
	show	Displays a pop-up menu and returns the item selected
	switchMenu	Builds and displays a pop-up menu and handles the menu choice
Query	executeQBE	Executes a QBE query
	isAssigned	Reports whether a Query variable has an assigned value
	QUERY	Begins a QBE statement or string
	writeQBE	Writes a Query statement to a specified file
Record	view	Displays in a dialog box the value of a variable
Report	attach	Associates a Report variable with an open report
	bringToTop	Brings the window to the top of the display stack
	close	Closes a window
	create	Creates a blank report in the Report Design window
	currentPage	Reports the current page number of a report

Type	Method	Description
	design	Switches a report from a View Data window to the Report Design window
	enumUIObjectNames	Creates a table listing the UIObjects contained in a report
	enumUIObjectProperties	Creates a table listing the properties of each UIObject contained in a report
	getPosition	Returns the postion of a report window onscreen
	getTitle	Returns the text of the window title bar
	hide	Makes a window invisible
	isMaximized	Reports whether a window is displayed at its maximum size
	isMinimized	Reports whether a window is displayed as an icon
	isVisible	Reports whether any part of a window is displayed
	load	Opens a report in the Report Design
	maximize	Maximizes a window
	minimize	Minimizes a window
	moveToPage	Displays a specified page of a report
	open	Opens a report and prints it
	print	Prints a report
	run	Switches a report from the View Data window to the View Data window
	save	Saves a report to disk
	setPosition	Positions a window on the screen
	setTitle	Sets the text in the window title bar
	show	Displays an minimized window at its original size, makes a hidden report visible
Session	addAlias	Adds a database alias to a session
	addPassword	Presents a password allowing access to a protected table
	advancedWildcardsInLocate	Specifies whether this session can use advanced wildcards in locate operations
	blankAsZero	Specifies whether to treat blank values as zeros in calculations
	close	Closes a session
	enumAliasNames	Creates a table listing the names of database aliases available to a session
	enumDataBaseTables	Creates a Paradox table listing the tables in a database
	enumDriverCapabilities	Creates three Paradox tables that list the capabilities of the current driver
	enumDriverInfo	Lists the available drivers
	enumDriverNames	Creates a Paradox table listing the names of the drivers available in the current session
	enumDriverTopics	Creates a Paradox table listing the topics currently available for each driver type
	enumEngineInfo	Creates a Paradox table listing the current ODAPI engine properties

Type	Method	Description
	enumFolder	Creates a Paradox table of an array listing files in a folder
	enumOpenDatabases	Creates a Paradox table listing theo open databases
	enumUsers	Creates a Paradox table listing all known users with an open channel to the ODAPI engine
	getAliasPath	Returns the path for a specified alias
	getNetUserName	Returns the network user name for a session
	ignoreCaseInLocate	Specifies whether to ignore case-sensitivity in locate operations
	isAdvancedWildcardsInLocate	Reports whether this session is using advanced wildcards in locate operations
	isAssigned	Reports whether a Session variable has an assigned value
	isBlankZero	Reports whether blank values are being treated as zero in calculations
	isIgnoreCaseInLocate	Reports whether the current session is ignoring case-sensitivity in locate operations
	lock	Locks one or more tables
	open	Opens a session
	removeAlias	Removes an alias from a session
	removeAllPasswords	Removes all passwords presented to a session
	removePassword	Removes a password presented to a session
	retryPeriod	Returns the number of seconds to retry an operation on a locked record or table
	saveCGF	Saves the current session's alias information to a file
	setAliasPath	Sets the path for an alias
	setRetryPeriod	Sets the number of seconds to retry an action on a locked table
	unlock	Unlocks one or more tables
SmallInt	bitAND	Performs a bitwise AND operation on two values
	bitIsSet	Reports whether a bit is 1 or 0
	bitOR	Performs a bitwise OR operation on two values
	bitXOR	Performs a bitwise XOR operation on two values
	int	Casts a value as an integer
	smallInt	Casts a value as a small integer
	view	Displays in a dialog box the value of a variable
StatusEvent	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event

Type	Method	Description
	setReason	Specifies a reason for generating an event
	setStatusValue	Specifies the text of a status message
	statusValue	Returns the text of a status message
	view	Displays the value of a variable
String	advMatch	Searches text for a specified string
	ansiCode	Returns the ANSI code of a one-character string
	breakApart	Splits a string into substrings
	chr	Converts an ASCII code to a one-character string
	chrOEM	Converts an ANSI code to a one-character string
	chrToKeyName	Returns the virtual key-code string of a one-character string
	fill	Returns a string containing repeated instances of a string
	format	Returns a formatted string for display or printing
	ignoreCaseInStringCompares	Specifies whether to consider case when comparing strings
	isIgnoreCaseInStringCompares	Reports whether case is considered when comparing strings
	isSpace	Reports whether a string contains only spaces (or is empty)
	keyNameToChar	Returns the one-character string represented by a virtual key-code string
	keyName to VKCode	Returns the VK_Code of a virtual key-code string
	lower	Converts a string to lower case
	lTrim	Removes leading blanks from a string
	match	Compares a string with a pattern
	oemCode	Returns the OEM code of a one-character string
	rTrim	Removes trailing blanks from a string
	search	Returns the position of one string inside another
	size	Returns the length of a string
	space	Creates a string of a specified number of spaces
	string	Casts a value as a String
	strVal	Converts a value to a string
	substr	Returns a portion of a string
	toANSI	Converts a string of OEM characters to ANSI characters
	toOEM	Converts a string of ANSI characters to OEM characters
	upper	Converts a string to upper case
	view	Displays the value of a variable in a dialog box
	vkCodeToKeyName	Converts a virtual key-code constant to a virtual key-code string
System	beep	Sounds the Windows default beep
	close	Closes the current form
	constantNameToValue	Returns the numeric value of a constant
	constantValueToName	Returns the name of a constant
	cpuClockTime	Returns the number of seconds since the computer was booted

Type	Method	Description
	debug	Halts execution of a method and invokes the Debugger
	dlgAdd	Invokes the Table Add dialog box
	dlgCopy	Invokes the Table Copy dialog box
	dlgCreate	Invokes the Create Table dialog box
	dlgDelete	Invokes the Table Delete dialog box
	dlgEmpty	Invokes the Table Empty dialog box
	dlgNetDrivers	Invokes the Drivers dialog box
	dlgNetLocks	Creates and displays a table of lock information
	dlgNetRefresh	Invokes the Network Refresh Rate dialog box
	dlgNetRetry	Invokes the Network Retry Period dialog box
	dlgNetSetLocks	Invokes the Table Locks dialog box
	dlgNetSystem	Invokes the ODAPI System Information dialog box
	dlgNetUserName	Invokes the Network User Name dialog box
	dlgNetWho	Invokes the Current Users dialog box
	dlgRename	Invokes the Table Rename dialog box
	dlgRestructure	Invokes the Table Restructure dialog box
	dlgSort	Invokes the Table Sort dialog box
	dlgSubtract	Invokes the Table Subtract dialog box
	dlgTableInfo	Invokes the Structure Information dialog box
	enumDesktopWindowNames	Creates a table listing open Paradox applications
	enumFonts	Creates a table listing fonts in the user's system
	enumFormNames	Creates an array listing open forms
	enumReportNames	Creates an array listing open reports
	enumRTLClassNames	Creates a table listing the object types known to ObjectPAL
	enumRTLConstants	Creates a table listing the constants defined by ObjectPAL
	enumRTLMethods	Creates a table listing the methods in ObjectPAL
	enumWIndowNames	Creates a table or an array listing open windows
	errorClear	Clears the error stack
	errorCode	Returns a number describing the most recent runtime error or error condition
	errorLog	Adds information to the error stack
	errorMessage	Returns the text of the most recent error message
	errorPop	Removes the top layer of information from the error stack
	errorShow	Displays the error dialog box
	errorTrapOnWarnings	Specifies whether to handle wanring errors as critical errors
	execute	Executes a DOS command
	exit	Closes Paradox and exits to Windows
	fail	Causes a method to fail

Type	Method	Description
	fileBrowser	Displays the Browser and returns the names of one or more files selected by the user
	formatAdd	Adds a format
	formatDelete	Deletes a format
	formatExist	Reports whether a format exists
	formatSetCurrencyDefault	Sets the default display format for Currency values
	formatSetDateDefault	Sets the default display format for Date values
	formatSetDateTimeDefault	Sets the default display format for DateTime values
	formatSetLogicalDefault	Sets the default display format for Logical values
	formatSetLongIntDefault	Sets the default display format for LongInt values
	formatSetNumberDefault	Sets the default display format for Number values
	formatSetSmallIntDefault	Sets the default display format for SmallInt values
	formatSetStringDefault	Sets the default display format for String values
	formatSetTimeDefault	Sets the default display format for Time values
	getMouseScreenPosition	Returns the mouse position as a Point
	helpOnHelp	Displays information about how to use the Help system
	helpQuit	Tells the Help application it is no longer needed
	helpSetIndex	Sets the help index
	helpShowContext	Displays help for a particular context
	helpShowIndex	Displays the index of a specified help file
	helpShowTopic	Displays help for a specified context
	helpShowTopicInKeywordTable	Displays help for a topic identified by a keyword in an alternate keyword table
	message	Displays a message in the status line
	msgAbortRetryIgnore	Displays a dialog box containing a message and three buttons: Yes, No, and Ignore
	msgInfo	Displays a dialog box containing an i icon, a message, and an OK button
	msgQuestion	Displays a dialog box containing a message, a ? icon, and two buttons: Yes or No
	msgRetryCancel	Displays a dialog box containing a message and two buttons: Retry and Cancel
	msgStop	Displays a dialog box containing a stop icon, a message, and an OK button
	msgYesNoCancel	Displays a dialog box containing a message and three buttons: Yes, No, and Cancel
	pixelsToTwips	Converts screen coordinates from pixels to twips
	play	Plays a standalone script
	quit	Closes a form
	readEnvironmentString	Reads an item from the DOS environment
	readProfileString	Reads an item from the user's WIN.INI file

Type	Method	Description
	setMouseScreenPosition	Displays the pointer at a specified position
	setMouseShape	Specifies the shape of the pointer
	sleep	Produces a delay of a specified duration
	sound	Creates a sound of specified frequency and duration
	sysInfo	Creates a dynamic array of information about the system running Paradox
	tracerClear	Clears the Tracer window
	tracerHides	Hides the Tracer window
	tracerOff	Closes the Tracer window
	tracerOn	Activates code tracing
	tracerSave	Saves the contents of the Tracer window to a file
	tracerShow	Makes the Tracer window visible
	tracerToTop	Makes the Tracer window the topmost window
	tracerWrite	Writes a message to the Tracer window
	twipsToPixels	Converts screen coordinates from twips to pixels
	version	Returns the Paradox version number
	winGetMessageID	Returns the ID of a Windows message
	winPostMessage	Posts a message to Windows
	winSendMessage	Sends a message to Windows
	writeEnvironmentString	Writes information about the user's system environment to a file
	writeProfileString	Writes information about the user's system to a file
Table	add	Adds the data in one table to another table
	attach	Associates a Table variable with a table on disk
	cAverage	Returns the average value of a field (column) in a table
	cCount	Returns the number of non-blank values in a field (column) of a table
	cMax	Returns the maximum value of a field (column) in a table
	cMin	Returns the minimum value in a field (column) of a table
	cNpv	Returns the net present value of a field (column), based on a specified discount or interest rate
	compact	Removes deleted records from a dBASE table
	copy	Copies a table
	CREATE	Creates a table
	cSamStd	Returns the sample standard deviation of a field (column) of a table
	cSamVar	Returns the sample variance of a field (column) in a table
	cStd	Returns the standard deviation of a field (column) in a table
	cSum	Returns the sum of the values in a field (column) of a table
	cVar	Returns the variance of a field in a table
	delete	Deletes a table
	dropIndex	Removes an index from a table

Type	Method	Description
	empty	Removes all records from a table in a database
	enumFieldNames	Fills an array with the names of fields in a table
	enumFieldNamesInIndex	Fills an array with the names of fields in a table
	enumFieldStruct	Creates a Paradox table listing the field structure of a table
	enumIndexStruct	Creates a Paradox table listing the structure of a table's secondary indexes
	enumRefIntStruct	Creats a Paradox table listing a table's referential integrity information
	enumSecStruct	Creates a Paradox table listing a table's security information
	familyRights	Tests for a user's ability to create or modify objects in a table's family
	fieldName	Returns the name of field in a table, given a field number
	fieldNo	Returns the position of a field in a table
	fieldType	Returns the type of a field in a table
	INDEX	Creates a secondary index on a specified field of a table
	isAssigned	Reports whether a Table variable has been assigned a value
	isEmpty	Reports whether a table in a database contains any records
	isEncrypted	Reports whether a table is encrypted
	isShared	Reports whether a table is in the user's current private directory
	isTable	Reports whether a table exists in a database
	lock	Locks a specified table
	nFields	Returns the number of fields in a table
	nKeyFields	Returns the number of key fields in the primary or current index for a table
	nRecords	Returns the number of records in a table
	protect	Encrypts and assigns an owner password to a table
	reIndex	Rebuilds specified index files
	reIndexAll	Rebuilds all index files
	rename	Renames a table
	setExclusive	Specifies whether to give the user exclusive rights to a table when it is opened
	setFilter	Specifies a range of records to include
	setIndex	Specifies an index for a table
	setReadOnly	Specifies whether to give the user read-only rights to a table when it is opened
	showDeleted	Specifies whether to display deleted records in a dBASE table
	SORT	Sorts a table
	subtract	Subtracts the records in one table from another table
	tableRights	Tells whether the user has rights to perform certain operations on a table
	type	Returns the type of a table

Type	Method	Description
	unAttach	Ends the association between a Table variable and a table on disk
	unlock	Unlocks a specified table
	unProtect	Decrypts and removes an owner password from a table
	usesIndexes	Specifies index files to use and maintain with a dBASE table
TableView	action	Performs an action command
	bringToTop	Brings the window to the top of the display stack
	close	Closes a window
	getPosition	Returns the position of the window onscreen
	getTitle	Returns the text displayed in the window's title bar
	hide	Makes a window invisible
	isMaximized	Reports whether a window is displayed at its maximum size
	isMinimized	Reports whether a window is displayed as an icon
	isVisible	Reports whether any part of a window is displayed
	maximize	Maximizes a window
	minimize	Minimizes a window
	moveToRecord	Moves to a specific record in a table
	open	Opens a window
	setPosition	Positions a window on the screen
	setTitle	Sets the text in the window title bar
	show	Displays an iconized window at its original size
	wait	Suspends execution of a method
	windowHandle	Returns a handle to a window
TCursor	add	Adds the records of one table to another
	atFirst	Reports whether the cursor is at the first record of a table
	atLast	Reports whether the cursor is at the last record in table
	attach	Binds a TCursor to a UIObject
	attachToKeyViol	Attaches a TCursor to the existing record that has the same key value as the record you attempted to post
	bot	Tests for a move past the beginning of a table
	cancelEdit	Ends Edit mode without changing the current record
	cAverage	Returns the average value of a field (column) in a table
	cCount	Returns the number of non-blank values in a field (column) of a table
	close	Closes a table
	cMax	Returns the maximum value of a field (column) in a table
	cMin	Returns the minimum value in a field (column) of a table
	cNpv	Returns the net present value of a field (column), based on a specified discount or interest rate
	commit	Posts changes to a record
	compact	Removes deleted records from a dBASE table

Type	Method	Description
	copy	Copies a table
	copyFromArray	Copies data from an array to the fields of the current record
	copyRecord	Copies a record into the current record of a table
	copyToArray	Copies the fields of the current record to an array
	cSamStd	Returns the sample standard deviation of a field (column) of a table
	cSamVar	Returns the sample variance of a field (column) in a table
	cStd	Returns the standard deviation of a field (column) in a table
	cSum	Returns the sum of the value in a field (column) of a table
	currRecord	Reads the current record into the record buffer
	cVar	Returns the variance of a field in a table
	deleteRecord	Deletes the current record from the table
	didFlyAway	Reports whether the current record moved to a different position as the result of a key value change
	dropIndex	Stops using a specified index file
	edit	Puts a table into Edit mode
	empty	Deletes all records from a table
	end	Moves to the last record in a table
	endEdit	Exits Edit mode and accepts changes made to a record
	enumFieldNames	Fills an array with the names of fields in a table
	enumFieldNamesInIndex	Fills an array with the names of fields in a table's index
	enumFieldStruct	Creates a Paradox table listing the structure of a TCursor
	enumIndexStruct	Creates a table listing the structure of a TCursor's secondary indexes
	enumLocks	Creates a Paradox table listing the locks currently applied to a TCursor
	enumRefIntStruct	Creates a Paradox table listing referential integrity information for a TCursor
	enumSecStruct	Writes table security information to a Paradox table
	enumTableProperties	Writes the properties of a TCursor to a table
	eot	Tests for a move past the end of a table
	familyRights	Tests for a user's ability to create or modify objects in a table's family
	fieldName	Returns the name of a field
	fieldNo	Returns the position of a field in a table
	fieldRights	Reports whether a user has rights to read or modify a field in a table
	fieldSize	Returns the size of a field
	fieldType	Returns the data type of a field
	fieldUnits2	Returns the number of decimal places being maintained for a field in a dBASE table
	fieldValue	Reads the value of a specified field

Type	Method	Description
	getLanguageDriver	Reports the current language driver for a table
	getLanguageDriverDesc	Reports the current language driver description for a table
	home	Moves to the first record of a table
	initRecord	Empties the current record buffer
	insertAfterRecord	Inserts a record into a table after the current record
	insertBeforeRecord	Inserts a record into a table before the current record
	insertRecord	Inserts a record into a table
	isAssigned	Reports whether a TCursor variable has been assigned a value
	isEdit	Reports whether a TCursor is in Edit mode
	isEmpty	Determines whether a table contains any records
	isEncrypted	Reports whether a table is password-protected
	isRecordDeleted	Reports whether the current record has been deleted (dBASE tables only)
	isShared	Reports whether a table is in a shared network directory other than the user's private directory
	isShowDeletedOn	Reports whether deleted records in a dBASE table are shown
	isValid	Reports whether the contents of a field are legitimate and complete
	locate	Searches for a specified field value
	locateNext	Searches for a specified field value
	locateNextPattern	Locates the next record containing a field that has a specified pattern of characters
	locatePattern	Locates a record containing a field that has a specified pattern of characters
	locatePrior	Searches backward for a specified field value
	locatePriorPattern	Locates the previous record containing a field that has a specified pattern of characters
	lock	Places specified locks on a specified table
	lockRecord	Puts a write lock on the current record
	lockStatus	Returns the number of times you have placed a lock on a table
	moveToRecNo	Moves a TCursor to a specific record in a table
	moveToRecord	Moves a TCursor to a specific record in a table
	nextRecord	Moves to the next record in a table
	nFields	Returns the number of fields in a table
	nKeyFields	Returns the number of key fields in the current index of a table
	nRecords	Returns the number of records in a table
	open	Opens a table
	postRecord	Posts changes to a record
	priorRecord	Moves to the previous record in a table
	qLocate	Searches an indexed table for a specified field value
	recNo	Returns the record number of the current record

Type	Method	Description
	recordStatus	Reports about the status of a record
	reIndex	Rebuilds specified index files
	reIndexAll	Rebuilds all index files
	seqNo	Returns the record (sequence) number of the current record
	setFieldValue	Assigns a value to a specified field
	setFilter	Sets the range of records a TCursor can point to
	setFlyAwayControl	Controls whether the TCursor points to a record whose position has changed as the result of a **unlockRecord**
	showDeleted	Specifies whether to show deleted records in a dBASE table
	skip	Moves forward or backward a specified number of records in a table
	sortTo	Sorts a table
	subtract	Subtracts the records in one table from another table
	switchIndex	Specifies another index file to use
	tableName	Returns the name of the table associated with a TCursor
	tableRights	Reports about the operations you can perform on a table
	type	Returns the type of a table
	unDeleteRecord	Undeletes the current record from a dBASE table
	unlock	Removes specified locks from a TCursor
	unlockRecord	Unlocks the current record
	updateRecord	Updates the existing record with data from the new record when a key violation exists
TextStream	advMatch	Searches for a pattern of characters in a text file
	close	Closes a text file
	commit	Writes the contents of the text buffer to disk
	create	Creates a text file for reading and writing
	end	Sets the file pointer to the end of a text file
	eof	Tests for a move past the end of a text file
	home	Sets the file pointer to the beginning of a text file
	open	Opens a text file in a specified mode
	position	Returns the pointer's position in a text file
	readChars	Reads a specified number of characters from a text file
	readLine	Reads a line from a text file
	setPosition	Positions the pointer in a text file
	size	Returns the number of characters in a text file
	writeLine	Writes a string to a text file
	writeString	Writes a character string to a text file
Time	hour	Extracts the hour from a time as a number
	milliSec	Extracts the milliseconds from a time as a number
	minute	Extracts the minutes from a time as a number

Type	Method	Description
	second	Extracts the seconds from a time as a number
	time	Casts a value as a time
	view	Displays in a dialog box the value of a variable
TimerEvent	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf
	isTargetSelf	Reports whether an object is the target of an event
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setReason	Specifies a reason for generating an event
UIObject	action	Performs a specified action
	atFirst	Reports whether the pointer is at the first record of a table
	atLast	Reports whether the pointer is at the last record in table
	attach	Binds a UIObject to another object
	broadcastAction	Broadcasts an action to an object
	cancelEdit	Cancels record changes without ending Edit mode
	convertPointWithRespectTo	Changes the frame of reference for calculating the coordinates of a point
	copyFromArray	Copies data from an array to a record of a table
	copyToArray	Copies data from a record to an array
	create	Creates an object
	currRecord	Reads the current record into the record buffer
	delete	Deletes an object from a form
	deleteRecord	Deletes the current record from the table
	edit	Puts a table into Edit mode
	empty	Deletes all records from a table
	end	Moves to the last record in a table
	endEdit	Ends an Edit session, and accepts changes to the current record
	enumFieldNames	Fills an array with the names of the fields in a TableFrame
	enumLocks	Creates a Paradox table lising the locks currently applied to a UIObject and returns the number of locks
	enumObjectNames	Fills an array with the names of the objects in a form
	enumSource	Fills a table with the source code attached to objects in a form
	enumSourceToFile	Creates a text file listing the source code attached to objects in a form
	enumUIClasses	Writes a list of object classes to a Paradox table
	enumUIObjectNames	Writes the names of the objects in a form to a Paradox table

Type	Method	Description
	enumUIObjectProperties	Gets the properties of each object in a form, and writes the data to a Paradox table
	execMethod	Calls a custom method that takes no arguments
	getBoundingBox	Returns the coordinates of the invisible box that bounds an object
	getProperty	Returns the value of a specified property
	getPropertyAsString	Returns the value of a specified property as a string
	getRGB	Find the red, green, and blue components of a color value
	hasMouse	Reports whether the mouse is over an object
	home	Moves to the first record in a table
	insertAfterRecord	Inserts a record into a table after the current record
	insertBeforeRecord	Inserts a record into a table before the current record
	insertRecord	Inserts a record into a table
	isContainerValid	Reports whether an object's container is valid
	isEdit	Reports whether an object is in Edit mode
	isEmpty	Reports whether a table contains any records
	isLastMouseClickedValid	Reports whether the last object to receive a mouse click is valid
	islastMouseRightClickedValid	Reports whether the last object to receive a right mouse click is valid
	keyChar	Sends a key code to an object
	keyPhysical	Calls an object's **keyPhysical** method
	killTimer	Stops the timer associated with an object
	locate	Searches for a specified field value
	locateNext	Searches for a specified field value
	locateNextPattern	Locates the next record containing a field that has a specified pattern of characters
	locatePattern	Locates a record containing a field that has a specified pattern of characters
	locatePrior	Searches backward for a specified field value
	locatePriorPattern	Locates the previous record containing a field that has a specified pattern of characters
	lockRecord	Puts a write lock on the current record
	lockStatus	Returns the number of locks on a table
	menuAction	Sends an event to an object's **menuAction** method
	methodDelete	Deletes a specified method
	methodGet	Returns the text of a specified method
	methodSet	Sets the text of a specified method
	mouseClick	Generates a **mouseClick** event and sends it to an object
	mouseDouble	Generates a **mouseDouble** event and sends it to an object
	mouseDown	Generates a **mouseDown** event and sends it to an object
	mouseEnter	Generates a **mouseEnter** event and sends it to an object

Type	Method	Description
	mouseExit	Generates a **mouseExit** event and sends it to an object
	mouseMove	Generates a **mouseMove** event and sends it to an object
	mouseRightDouble	Generates a **mouseRightDouble** event and sends it to an object
	mouseRightDown	Generates a **mouseRightDown** event and sends it to an object
	mouseRightUp	Generates a **mouseRightUp** event and sends it to an object
	mouseUp	Generates a **mouseUp** event and sends it to an object
	moveTo	Sets the focus to a specified object
	moveToRecord	Moves to a specific record in a table
	moveToRecNo	Moves to a specific record in a dBASE table
	nextRecord	Moves to the next record in a table
	nFields	Returns the number of fields in a table
	nKeyFields	Returns the number of key fields in the current index of a table
	nRecords	Returns the number of records in a table
	pixelsToTwips	Converts screen coordinates from pixels to twips
	postAction	Posts an action to an action queue for delayed execution
	postRecord	Posts a pending record to a table
	priorRecord	Moves to the previous record in a table
	pushButton	Generates a **pushButton** event and sends it to an object
	recordStatus	Reports about the status of a record
	resync	Resynchronizes an object to a TCursor
	rgb	Defines a color
	setFilter	Sets the range of records a table object can point to
	setPosition	Sets the position of an object
	setProperty	Sets a property to a specified value
	setTimer	Starts the timer for an object
	skip	Moves forward or backward a specified number of records in a table
	switchIndex	Specifies another index to use to view the records in a table
	twipsToPixels	Converts screen coordinates from twips to pixels
	undeleteRecord	Undeletes the current record from a dBASE table
	unlockRecord	Removes a write lock from the current record
	view	Displays in a dialog box the value of a UIObject
	wasLastClicked	Tells whether an object was the last object to receive a mouse click
	wasLastRightClicked	Tells whether an object was the last object to receive a right mouse click
ValueEvent	errorCode	Reports the status of the error flag
	getTarget	Returns the name of the target of an event
	isFirstTime	Reports whether the form is handling an event for the first time before dispatching it
	isPreFilter	Reports whether the form is handling an event on its own behalf

Type	Method	Description
	isTargetSelf	Reports whether an object is the target of an event
	newValue	Returns the new value of a ValueEvent
	reason	Reports why an event occurred
	setErrorCode	Sets the error code for an event
	setNewValue	Specifies a value to set for a ValueEvent
	setReason	Specifies a reason for generating an event

APPENDIX B

ANSI character set

Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char	Code	Char
0	•	32	(space)	64	@	96	`	128	•	160	(hard space)	192	À	224	à
1	•	33	!	65	A	97	a	129	•	161	¡	193	Á	225	á
2	•	34	"	66	B	98	b	130	‚	162	¢	194	Â	226	â
3	•	35	#	67	C	99	c	131	ƒ	163	£	195	Ã	227	ã
4	•	36	$	68	D	100	d	132	„	164	¤	196	Ä	228	ä
5	•	37	%	69	E	101	e	133	…	165	¥	197	Å	229	å
6	•	38	&	70	F	102	f	134	†	166	¦	198	Æ	230	æ
7	•	39	'	71	G	103	g	135	‡	167	§	199	Ç	231	ç
8	•	40	(	72	H	104	h	136	ˆ	168	¨	200	È	232	è
9	•	41	)	73	I	105	i	137	‰	169	©	201	É	233	é
10	•	42	*	74	J	106	j	138	Š	170	ª	202	Ê	234	ê
11	•	43	+	75	K	107	k	139	‹	171	«	203	Ë	235	ë
12	•	44	,	76	L	108	l	140	Œ	172	¬	204	Ì	236	ì
13	•	45	-	77	M	109	m	141	•	173	-	205	Í	237	í
14	•	46	.	78	N	110	n	142	•	174	®	206	Î	238	î
15	•	47	/	79	O	111	o	143	•	175	¯	207	Ï	239	ï
16	•	48	0	80	P	112	p	144	•	176	°	208	Đ	240	ð
17	•	49	1	81	Q	113	q	145	'	177	±	209	Ñ	241	ñ
18	•	50	2	82	R	114	r	146	'	178	²	210	Ò	242	ò
19	•	51	3	83	S	115	s	147	"	179	³	211	Ó	243	ó
20	•	52	4	84	T	116	t	148	"	180	´	212	Ô	244	ô
21	•	53	5	85	U	117	u	149	•	181	µ	213	Õ	245	õ
22	•	54	6	86	V	118	v	150	–	182	¶	214	Ö	246	ö
23	•	55	7	87	W	119	w	151	—	183	·	215	×	247	÷
24	•	56	8	88	X	120	x	152	˜	184	¸	216	Ø	248	ø
25	•	57	9	89	Y	121	y	153	™	185	¹	217	Ù	249	ù
26	•	58	:	90	Z	122	z	154	š	186	º	218	Ú	250	ú
27	•	59	;	91	[	123	{	155	›	187	»	219	Û	251	û
28	•	60	<	92	\	124	|	156	œ	188	¼	220	Ü	252	ü
29	•	61	=	93	]	125	}	157	•	189	½	221	Ý	253	ý
30	•	62	>	94	^	126	~	158	•	190	¾	222	Þ	254	þ
31	•	63	?	95	_	127	•	159	Ÿ	191	¿	223	ß	255	ÿ

• Characters not supported by Windows

APPENDIX C

Windows keycodes

Table C-1 Windows keycodes

Name	Dec	Hex	Description
VK_LBUTTON	1	01H	Left mouse button
VK_RBUTTON	2	02H	Right mouse button
VK_CANCEL	3	03H	Used for control-break processing
VK_MBUTTON	4	04H	Middle mouse button (3-button mouse)
	5-7	05H-07H	Undefined
VK_BACK	8	08H	BACKSPACE key
VK_TAB	9	09H	TAB key
	10-11	0AH-0BH	Undefined
VK_CLEAR	12	0CH	CLEAR key
VK_RETURN	13	0DH	RETURN key
VK_SHIFT	16	10H	SHIFT key
VK_CONTROL	17	11H	CONTROL key
VK_MENU	18	12H	MENU key
VK_PAUSE	19	13H	PAUSE key
VK_CAPITAL	20	14H	CAPITAL key
VK_	21-25	15H-19H	Reserved for Kanji systems
	26	1AH	Undefined
VK_ESCAPE	27	1BH	ESCAPE key
	28-31	1CH-1FH	Reserved for Kanji systems
VK_SPACE	32	20H	SPACEBAR
VK_PRIOR	33	21H	PAGE UP key
VK_NEXT	34	22H	PAGE DOWN key
VK_END	35	23H	END key
VK_HOME	36	24H	HOME key
VK_LEFT	37	25H	LEFT ARROW key
VK_UP	38	26H	UP ARROW key

Name	Dec	Hex	Description
VK_RIGHT	39	27H	RIGHT ARROW key
VK_DOWN	40	28H	DOWN ARROW key
VK_SELECT	41	29H	SELECT key
	42	2AH	OEM specific
VK_EXECUTE	43	2BH	EXECUTE key
VK_SNAPSHOT	44	2CH	PRINTSCREEN key for Windows 3.0 and later
VK_INSERT	45	2DH	INSERT key
VK_DELETE	46	2EH	DELETE key
VK_HELP	47	2FH	HELP key
VK_0	48	30H	0 key
VK_1	49	31H	1 key
VK_2	50	32H	2 key
VK_3	51	33H	3 key
VK_4	52	34H	4 key
VK_5	53	35H	5 key
VK_6	54	36H	6 key
VK_7	55	37H	7 key
VK_8	56	38H	8 key
VK_9	57	39H	9 key
	58-64	3AH-40H	Undefined
VK_A	65	41H	A key
VK_B	66	42H	B key
VK_C	67	43H	C key
VK_D	68	44H	D key
VK_E	69	45H	E key
VK_F	70	46H	F key
VK_G	71	47H	G key
VK_H	72	48H	H key
VK_I	73	49H	I key
VK_J	74	4AH	J key
VK_K	75	4BH	K key
VK_L	76	4CH	L key
VK_M	77	4DH	M key
VK_N	78	4EH	N key
VK_O	79	4FH	O key
VK_P	80	50H	P key
VK_Q	81	51H	Q key
VK_R	82	52H	R key

Name	Dec	Hex	Description
VK_S	83	53H	S key
VK_T	84	54H	T key
VK_U	85	55H	U key
VK_V	86	56H	V key
VK_W	87	57H	W key
VK_X	88	58H	X key
VK_Y	89	59H	Y key
VK_Z	90	5AH	Z key
	91-95	5BH-5FH	Undefined
VK_NUMPAD0	96	60H	Key pad 0 key
VK_NUMPAD1	97	61H	Key pad 1 key
VK_NUMPAD2	98	62H	Key pad 2 key
VK_NUMPAD3	99	63H	Key pad 3 key
VK_NUMPAD4	100	64H	Key pad 4 key
VK_NUMPAD5	101	65H	Key pad 5 key
VK_NUMPAD6	102	66H	Key pad 6 key
VK_NUMPAD7	103	67H	Key pad 7 key
VK_NUMPAD8	104	68H	Key pad 8 key
VK_NUMPAD9	105	69H	Key pad 9 key
VK_MULTIPLY	106	6AH	Multiply key
VK_ADD	107	6BH	Add key
VK_SEPARATOR	108	6CH	Separator key
VK_SUBTRACT	109	6DH	Subtract key
VK_DECIMAL	110	6EH	Decimal key
VK_DIVIDE	111	6FH	Divide key
VK_F1	112	70H	F1 key
VK_F2	113	71H	F2 key
VK_F3	114	72H	F3 key
VK_F4	115	73H	F4 key
VK_F5	116	74H	F5 key
VK_F6	117	75H	F6 key
VK_F7	118	76H	F7 key
VK_F8	119	77H	F8 key
VK_F9	120	78H	F9 key
VK_F10	121	79H	F10 key
VK_F11	122	7AH	F11 key
VK_F12	123	7BH	F12 key
VK_F13	124	7CH	F13 key
VK_F14	125	7DH	F14 key

Name	Dec	Hex	Description
VK_F15	126	7EH	F15 key
VK_F16	127	7FH	F16 key
	128-135	80H-87H	OEM specific
	136-143	88H-8FH	Unassigned
VK_NUMLOCK	144	90H	NUM LOCK key
VK_OEM_SCROLL	145	91H	SCROLL LOCK key
	146-185	92H-B9H	Unassigned
VK_OEM_1	186	BAH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_PLUS	187	BBH	Plus (+) key
VK_OEM_COMMA	188	BCH	Comma (,) key
VK_OEM_MINUS	189	BDH	Minus (-) key
VK_OEM_PERIOD	190	BEH	Period (.) key
VK_OEM_2	191	BFH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_3	192	C0H	Keyboard-specific punctuation key (May not appear on every keyboard)
	193-218	C1H-DAH	Unassigned
VK_OEM_4	219	DBH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_5	220	DCH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_6	221	DDH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_7	222	DEH	Keyboard-specific punctuation key (May not appear on every keyboard)
VK_OEM_8	223	DFH	Keyboard-specific punctuation key (May not appear on every keyboard)
	224-225	E0H-E1H	OEM specific
VK_OEM_102	226	E2H	<> or \ | on enhanced non-U.S. IBM-compatible 102-key keyboard
	227-228	E3H-E4H	OEM specific
	229	E5H	Unassigned

Name	Dec	Hex	Description
	230	E6H	OEM specific
	231-232	E7H-E8H	Unassigned
	233-245	E9H-F5H	OEM specific
	246-254	F6H-FEH	Unassigned

APPENDIX D

Keywords

The keywords in this list cannot be used to name objects, variables, arrays, methods, or procedures. The case of the words is irrelevant; they cannot be used in any combination of uppercase or lowercase.

Also, as a general rule, you should not use object type names, names of basic language elements, names of methods and procedures in the run-time library, or names of built-in methods.

Table D-1 Keywords

and	array	as
case	caseInsensitive	const
create	Database	descending
disableDefault	doDefault	DynArray
else	enableDefault	endConst
endCreate	endFor	endForEach
endIf	endIndex	endMethod
endProc	endQuery	endRecord
endScan	endSort	endSwitch
endSwitchMenu	endTry	endType
endUses	endVar	endWhile
for	forEach	from
if	iif	in
index	indexStruct	is
key	like	loop
maintained	method	not
ObjectPAL	of	on
onFail	or	otherwise
passEvent	primary	proc
query	quitLoop	record
refIntStruct	retry	return

scan	secStruct	sort
step	struct	switch
switchMenu	tag	then
to	try	type
unique	uses	var
where	while	with
without		

APPENDIX E

Compatibility procedures

The procedures in this appendix are provided as a convenience to DOS PAL programmers. You can use these procedures to operate on tables by specifying the table name, rather than using a variable. Each of these procedures has a corresponding method or procedure in the ObjectPAL run-time library.

For example, to use the Table type **cSum** method, you would do the following:

```
var
   ordersTB Table
endVar

ordersTB.attach("orders.db")
message(ordersTB.cSum("Qty"))
```

To use the **cSum** compatibility function, you would do the following:

```
message(cSum("orders.db", "Qty"))
```

add

Adds the data in one table to another table.

Syntax

1. **add** (const ***sourceTableName*** String, const ***destTableName*** String) [, const ***append*** Logical [, const ***update*** Logical]]) Logical
2. **add** (const ***sourceTableName*** String, const ***destTableVar*** Table) [, const ***append*** Logical [, const ***update*** Logical]]) Logical

cAverage

Returns the average value of a field (column) in a table.

Syntax

1. **cAverage** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cAverage** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cCount

Returns the number of nonblank values in a field (column) of a table.

Syntax

1. **cCount** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cCount** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cMax

Returns the maximum value of a field (column) in a table.

Syntax

1. **cMax** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cMax** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cMin

Returns the minimum value in a field (column) of a table.

Syntax

1. **cMin** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cMin** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cNpv

Returns the net present value of a field (column), based on a specifed discount or interest rate.

Syntax

1. **cNpv** (const ***tableName*** String, const ***fieldName*** String, const ***discRate*** AnyType) Number
2. **cNpv** (const ***tableName*** String, const ***fieldNum*** SmallInt, const ***discRate*** AnyType) Number

copy

Copies a table.

Syntax

1. **copy** (const ***sourceTable*** String, const ***destTable*** String) Logical
2. **copy** (const ***sourceTable*** String, const ***destTable*** Table) Logical

cSamStd

Returns the sample standard deviation of a field (column) of a table.

Syntax

1. **cSamStd** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cSamStd** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cSamVar

Returns the sample variance of a field (column) in a table.

Syntax

1. **cSamVar** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cSamVar** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cStd

Returns the standard deviation of a field (column) in a table.

Syntax

1. **cStd** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cStd** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cSum

Returns the sum of the values in a field (column) of a table.

Syntax

1. **cSum** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cSum** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

cVar

Returns the variance of a field in a table.

Syntax

1. **cVar** (const ***tableName*** String, const ***fieldName*** String) Number
2. **cVar** (const ***tableName*** String, const ***fieldNum*** SmallInt) Number

delete

Deletes a table.

Syntax

delete (const ***tableName*** String) Logical

empty

Removes all records from a table in a database.

Syntax

empty (const ***tableName*** String) Logical

familyRights

Tests for a user's abitlity to create or modify object's in a table's family.

Syntax

familyRights(const ***tableName*** String, ***rights*** AnyType) Logical

fieldName

Returns the name of field in a table, given a field number.

Syntax

fieldName (const ***tableName*** String, const ***fieldNum*** SmallInt) String

fieldNo

Returns the position of a field in a table.

Syntax

fieldNo (const ***tableName*** String, const ***fieldName*** String) SmallInt

fieldType

Returns the type of a field in a table.

Syntax

1. **fieldType** (const ***tableName*** String, const ***fieldName*** String) String
2. **fieldType** (const ***tableName*** String, const ***fieldNum*** SmallInt) String

isEmpty

Reports whether a table contains any records.

Syntax

isEmpty (const ***tableName*** String) Logical

isEncrypted

Reports whether a table is encrypted.

Syntax

isEncrypted (const ***tableName*** String) Logical

isShared

Reports whether a table is being shared by another user.

Syntax

isShared (const ***tableName*** String) Logical

isTable

Reports whether a table exists in a database.

Syntax

isTable (const ***tableName*** String) Logical

nFields

Returns the number of fields in a table.

Syntax

nFields (const ***tableName*** String) LongInt

nKeyFields

Returns the number of key fields in the table.

Syntax

nKeyFields (const ***tableName*** String) LongInt

nRecords

Returns the number of records in a table.

Syntax

nRecords (const ***tableName*** String) LongInt

protect

Encrypts and assigns an owner password to a table.

Syntax

protect (const ***tableName*** String, const ***password*** String) Logical

rename

Renames a table.

Syntax

rename (const ***tableName*** String, const ***destTableName*** String) Logical

subtract

Subtracts the records in one table from another table.

Syntax

1. **subtract** (const ***sourceTableName*** String, const ***destTableName*** String) Logical
2. **subtract** (const ***sourceTableName*** String, const ***destTableName*** Table) Logical

tableRights

Tells whether the user has rights to perform certain operations on a table.

Syntax

tableRights (const ***tableName*** String, const ***rights*** String) Logical

APPENDIX F

Properties

This appendix lists the properties and property values for each UIObject. A solid dot (●) marks a read-write property, a hollow dot (○) marks a read-only property. This information is also listed online. To display the list, open an ObjectPAL Editor window, and choose Language | Properties; then select an object name and a property.

Table F-1 UIObject properties

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
Alignment							●		●													●	●	●
Arrived	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
BlankRecord									○	○						○			○	○	○		○	
Border										●														
Breakable	●						●																	
ButtonType				●																				
Caption										●														
CenterLabel				●																				
Class	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○	○	○
Color			●		●	●	●	●	●		●		●	●		●		●	●	●	●	●	●	●
CompleteDisplay							●		●														●	
ContainerName	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
ControlMenu										●														
CurrentRecordMarker.Color																					●			
CurrentRecordMarker.LineStyle																					●			
CurrentRecordMarker.Show																					●			

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
CursorColumn									●													●		
CursorLine									●													●		
CursorPos									●													●		
DataSource															●									
DateFormat																							●	
Default									○														○	
Deleted									○	○						○			○	○	○		○	
Design.ContainObjects	●	●	●	●	●			●			●		●				●	●				●		
Design.PinHorizontal	●	●	●	●	●	●	●	●	●		●	●	●	●		●	●	●		●		●		
Design.PinVertical	●	●	●	●	●	●	●	●	●		●	●	●	●		●	●	●		●		●		
Design.SizeToFit		●			●		●		●								●	●		●				
DesignSizing																						●		
DesktopForm										●														
DialogForm										●														
DialogFrame										●														
DisplayType							○		●															
Editing									○	○											○			
End														●										
FieldName									●												○		○	
FieldNo									○												○		○	
FieldRights									○														○	
FieldSize									○														○	
FieldType									○														○	
FieldUnits2									○														○	
FieldValid									○															
FieldView										○											○			
First	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
FlyAway									○	○						m			m	○				
Focus	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
Font.Color							●		●													●	●	●
Font.Size							●		●													●	●	●
Font.Style							●		●													●	●	●

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
Font.Typeface							●		●													●	●	●
Format.DateFormat							●		●														●	
Format.LogicalFormat							●		●														●	
Format.NumberFormat							●		●														●	
Format.TimeFormat							●		●														●	
Format.TimeStampFormat							●		●														●	
Frame.Color		●	●				●		●		●					●	●					●		
Frame.Style		●	●				●		●		●					●	●					●		
Frame.Thickness		●	●				●		●		●					●	●					●		
FullName	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
FullSize	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○		○		
Grid.Color						●														●				
Grid.GridStyle						●														●				
Grid.RecordDivider																				●				
Gridlines.Color																					●			
Gridlines.ColumnLines																					●			
Gridlines.HeadingLines																					●			
Gridlines.LineStyle																					●			
Gridlines.RowLines																					●			
Gridlines.Spacing																					●			
HeadingHeight																					●			
Headings	●																							
HorizontalScrollBar		●			●	●			●	●								●		●		●		
IndexField									○														○	
Inserting									○	○						○			○	○	○			
KeyField									○														○	
LabelText				●					●															
Line.Color								●																
Line.LineStyle								●																
Line.Thickness								●																
LineEnds														●										
LineSpacing																						●		

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
LineStyle														●										
LineType														●										
List.Count															●									
List.Selection															●									
Locked									)	)						)			)	)	)		)	
LookupTable									)														)	
LookupType									)														)	
Magnification		●					●		●								●						●	
Manager	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)		)		
MarkerPos									●													●		
MaximizeButton										●														
Maximum									)														)	
MemoView										)											)			
MinimizeButton										●														
Minimum									)														)	
Modal										●														
MouseActivate										●														
NCols																●				●	)			
NRecords									)	)						)			)	)	)		)	
NRows																●				●	)			
Name	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	)	●	●	●
Next	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)		)		
NoEcho							●		●															
OverStrike									●													●		
Owner	)	)	)	)	)	)	)	)	)		)	)	)	)	)	)	)	)	)	)		)		
Pattern.Color			●					●	●		●					●		●	●	●		●		
Pattern.Style			●					●	●		●					●		●	●	●		●		
PersistView										)										)				
Picture									)														)	
Position	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
PrecedePageHeader	●																							
Prev	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)	)		)		

	Band	Bitmap	Box	Button	Cell	Crosstab	EditRegion	Ellipse	Field	Form	Graph	Group	Header	Line	List	Multirecord	OLE	Page	Record	TableFrame	TableView	Text	TVData	TVHeading
PrintOn1stPage	●																							
RasterOperation		●					●		●															
Readonly							●		●							○			●	○	○			
RecNo									○	○						○			○	○	○		○	
Refresh									○	○						○			○	○	○		○	
Required									○														○	
RowHeight																					○			
RowNo									○							○			○	○	○		○	
Scroll		●			●	●	●		●	●	●	●			●	●	●	●	●	●	●	●		
SelectedText									●													●		
SeqNo									○	○						○			○	○	○		○	
Shrinkable	●																							
Size	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
SizeToFit										●														
SortOrder	●																							
StandardMenu										●														
Start														●										
Style				●																				
TabStop				●			●		●		●													
TableName						●			●	○	●					●			○	●	○		○	
Text									●													●		
ThickFrame										●														
Thickness														●										
Title										●														
TopLine									●													●		
Touched						○			●	●	○					●			●	●	○		○	
Translucent			●			●	●	●	●		●		●			●		●	●			●		
Value				●					●						●							●	●	
VerticalScrollBar		●			●	●			●	●						●	●			●		●		
Visible		●	●	●		●	●	●	●		●	●	●	●	●	●	●		●	●		●		
WordWrap							●		●													●		

Table F-2

Property	Data type	Values	Description
Alignment	SmallInt	TextAlignCenter, TextAlignJustify, TextAlignLeft, TextAlignRight	Specifies the position of text relative to a field object or a text object.
Arrived	Logical	True, False	Specifies whether the object is active.
BlankRecord	Logical	True, False	Reports whether a record is blank.
Border	Logical	True, False	Reports whether a form's window has a border.
Breakable	Logical	True, False	Specifies whether an object can be split across page breaks.
ButtonType	SmallInt	CheckBoxType, PushButtonType, RadioButtonType	Specifies the display type of a button.
Caption	Logical	True, False	Reports whether a form has a caption bar.
CenterLabel	Logical	True, False	Specifies whether a label is centered within a button.
Class	String	Band, Bitmap, Box, Button, Cell, Crosstab, EditRegion, Ellipse, Field, Form, Graph, Group, Header, Line, List, Multirecord, OLE, Page, Record, TableFrame, Text	Returns the class of a UIObject.
Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the display color of an object.
CompleteDisplay	Logical	True, False	Specifies whether to display the complete contents of a field.
ContainerName	String	N/A	Reports the name of an object's container.
CurrentRecordMarker.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the display color of the current record of a table view.
CurrentRecordMarker.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of the line that marks the current record in a table view.
CurrentRecordMarker.Show	Logical	True, False	Specifies whether to highlight the current record in a table view.

Property	Data type	Values	Description
CursorColumn	LongInt	N/A	The horizontal position of the insertion point in a field object, where position 0 is to the left of the first character.
CursorLine	LongInt	N/A	The vertical position of the insertion point in a field object, where the first line is line 1.
CursorPos	LongInt	N/A	The position of the insertion point in a field object, relative to the first character in the field. Counting begins with 0, the position to the left of the first character.
DataSource	String	N/A	The name of the table that provides items in a list.
DateFormat	N/A	Format specification	Specifies a field object's date format.
Default	String	N/A	The default value of a field.
DefineGroup	Logical	True, False	Specifies whether a report band defines a group.
Deleted	Logical	True, False	Reports whether a record in a dBASE table has been flagged as deleted.
Design.ContainObjects	Logical	True, False	Specifies whether an object can contain other objects.
Design.SizeToFit	Logical	True, False	Specifies whether the object will change size to accomodate its contents.
DesignSizing	SmallInt	TextFixedSize, TextGrowOnly, TextSizeToFit	Specifies design time sizing for a text box.
DesktopForm	Logical	True, False	Specifies whether a form's menus are used by other forms on the Desktop.
DialogForm	Logical	True, False	Specifies whether a form will open as a dialog box.
DialogFrame	Logical	True, False	When True and DialogForm and Border are also True, form has a conventional dialog box frame.
DisplayType	SmallInt	BitmapField, CheckBoxField, ComboField, EditField, LabeledField, ListField, OleField, RadioButtonField	Returns the display type of a field object.
Editing	Logical	True, False	Reports whether a form is in Edit mode, or a field object is active and being edited.
End	Point	N/A	Specifies the coordinates of the end of a line. See also: Start.
FieldName	String	N/A	Specifies the name of the field to which a field object or list is bound.

Property	Data type	Values	Description
FieldNo	SmallInt	N/A	Reports the position of a field in a table, where the first field is field 1.
FieldRights	String	ReadOnly, ReadWrite, All	Reports the user's field rights.
FieldSize	SmallInt	N/A	The field's size (string and dBASE numbers).
FieldType	String	N/A	The field's data type.
FieldUnits2	SmallInt	N/A	Indicates the number of decimal places in a dBASE number field.
FieldValid	Logical	True, False	Reports whether a field passes its own value checks.
FieldView	Logical	True, False	Reports whether a field is in Field View.
First	String	N/A	Returns the name of the first child object in a container.
FitHeight	Logical	True, False	Specifies whether an edit region expands vertically to accommodate text.
FitWidth	Logical	True, False	Specifies whether an edit region or crosstab cell expands horizontally to accommodate text.
FlyAway	Logical	True, False	Reports whether a record has moved to its sorted position in a table.
Focus	Logical	True, False	Reports whether an object's built-in **setFocus** method has been called. Set to False when the object's built-in **removeFocus** method is called.
Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of characters displayed in a field object or a text object.
Font.Size	SmallInt	>0	Specifies (in printer's points) the size of characters displayed in a field object or a text object.
Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline	Specifies the style of characters displayed in a field object or a text object.
Font.Typeface	String	Depends on system	Specifies the typeface of characters displayed in a field object or a text object.
Format.DateFormat	N/A	Format specification	Specifies the format for date values.
Format.LogicalFormat	N/A	Format specification	Specifies the format for logical values.
Format.NumberFormat	N/A	Format specification	Specifies the format for number values.

Property	Data type	Values	Description
Format.TimeFormat	N/A	Format specification	Specifies the format for time values.
Format.TimeStampFormat	N/A	Format specification	Specifies the format for time stamps.
Frame.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of an object's frame.
Frame.Style	SmallInt	DashDotDotFrame, DashDotFrame, DashedFrame, DottedFrame, DoubleFrame, Inside3DFrame, NoFrame, Outside3DFrame, ShadowFrame, SolidFrame, WideInsideDoubleFrame, WideOutsideDoubleFrame	Specifies the style of an object's frame.
Frame.Thickness	SmallInt	N/A	Specifies the thickness of an object's frame in pixels.
FullName	String	N/A	Returns the full name (including containership path) of an object in a form.
FullSize	Point	N/A	In scrolling object, returns actual size if you could see the whole thing.
Grid.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of the grid in a table frame.
Grid.GridStyle	SmallInt	tfSingleLine, tfDoubleLine, tfTripleLine, tf3D, tfNoGrid	Specifies the style of grid lines in a table frame.
Grid.RecordDivider	Logical	True, False	Specifies whether dividing lines are displayed between records in a table frame.
GridLines.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of grid lines in a Table window.
GridLines.ColumnLines	Logical	True, False	Specifies whether to display column lines in a Table window.
GridLines.HeadingLines	Logical	True, False	Specifies whether to display heading lines in a Table window.

Property	Data type	Values	Description
GridLines.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of grid lines in a Table window.
GridLines.RowLines	Logical	True, False	Specifies whether to display grid lines in a Table window.
GridLines.Spacing	SmallInt	TextSingleSpacing, TextDoubleSpacing, TextTripleSpacing	Specifies the spacing of gridlines in a Table window.
Heading.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of the heading of a Table window.
Heading.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of characters in the heading of a Table window.
Heading.Font.Size	SmallInt	Depends on system	Specifies the size of characters in the heading of a Table window.
Heading.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline	Specifies the style of characters in a heading of a Table window.
Heading.Typeface	String	Depends on system	Specifies the typeface of characters in a heading of a Table window.
Heading.Justification	SmallInt	TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter	Specifies the justification of characters in the heading of a Table window.
Headings	String	"GroupOnly", "PageAndGroup"	Specifies which report headings to print.
HorizontalScrollBar	Logical	True, False	Specifies whether a table frame has a horizontal scroll bar.
IndexField	Logical	True, False	Reports whether a field object is bound to an indexed field in a table.
Inserting	Logical	True, False	Returns True when a record is being inserted anywhere in a form.
Justification	SmallInt	TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter	Specifies the justification of data in a Table window.

Property	Data type	Values	Description
KeyField	Logical	True, False	Reports whether a field object is bound to a key field in a table.
LabelText	String	N/A	Specifies the text displayed in a button's label.
Line.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of a line for an ellipse.
Line.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of a line for an ellipse.
Line.Thickness	SmallInt	N/A	Specifies the thickness of a line for an ellipse.
LineEnds	SmallInt	NoArrow, OnBothEnds, OnOneEnd	Specifies whether (or where) to place arrows at the end of a line.
LineSpacing	SmallInt	TextDoubleSpacing, TextDoubleSpacing2, TextSingleSpacing, TextSingleSpacing2, TextTripleSpacing	Specifies the number of blank lines to print between each line of text in a field object or a text object.
LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine	Specifies the style of a line.
LineType	SmallInt	CurvedLine, StraightLine	Specifies the type of a line.
List.Count	SmallInt	N/A	Specifies the number of items in a list.
List.Selection	SmallInt	N/A	Specifies the item selected from a list.
Locked	Logical	True, False	Reports whether the table bound to a design object is locked.
LookupTable	String	N/A	The name of the lookup table for a field object.
LookupType	String	JustCurrentField, AllCorresponding	Specifies the type of table lookup.
Magnification	SmallInt	Magnify100, Magnify200, Magnify25, Magnify400, Magnify50, MagnifyBestFit	Specifies the display magnification of a bitmap object. You can also enter a literal value.
Manager	String	N/A	Returns UIObject name of a form.
MarkerPos	LongInt	N/A	The "other end" of a selection. See also CursorPos.
MaximizeButton	Logical	True, False	Reports or specifes whether a form's window has a maximize box.
Maximum	String	N/A	Specifies the maximum value allowed in a field.

Property	Data type	Values	Description
MemoView	Logical	True, False	Specifies whether a field object is in memo view mode.
MinimizeButton	Logical	True, False	Reports or specifies whether a form's window has a minimize box.
Minimum	String	N/A	Specifies the maximum value allowed in a field.
Moda!	Logical	True, False	Specifies whether a dialog form is modal.
MouseActivate	Logical	True, False	Specifies whether a dialog form gets focus as the result of a MouseEvent.
NCols	SmallInt	N/A	Returns the number of columns in a table frame or multi-record object.
NRecords	LongInt	N/A	Reports the number of records in the table bound to a design object.
NRows	SmallInt	N/A	Returns the number of rows in a table frame or multi-record object.
Name	String	N/A	Specifies the name of a design object.
Next	String	N/A	Returns the name of the next object in the same container.
NoEcho	Logical	True, False	Specifies whether characters typed into a field object are displayed.
OverStrike	Logical	True, False	Specifies whether a field or text object is in overstrike (as opposed to insert) mode.
Owner	String	N/A	Name of the logical container of an object, irrespective of intermediate cosmetic objects.
Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent	Specifies the color of a pattern.

Property	Data type	Values	Description
Pattern.Style	SmallInt	Bricks, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern	Specifies the style of a pattern.
PersistView	Logical	True, False	Specifies whether to remain in Field View or Memo View.
Picture	String	N/A	A template that formats the value in a field object.
PinHorizontal	Logical	True, False	Specifies whether to prevent an object from moving horizontally.
PinVertical	Logical	True, False	Specifies whether to prevent an object from moving vertically.
Position	Point	N/A	Specifies the coordinates of the upper right corner of a design object.
PrecedePageHeader	Logical	True, False	Specifies whether a report band should appear before the page header.
Prev	String	N/A	Returns the name of the previous object in the same container.
PrintOn1stPage	Logical	True, False	Specifies whether to print a report band on the first page of the report.
RasterOperation	LongInt	MergePaint, NotSourceCopy, NotSourceErase, SourceAnd, SourceCopy, SourceErase, SourceInvert, SourcePaint	Specifies how to blend the colors in two overlapping design objects.
Readonly	Logical	True, False	Specifies whether a field object is read-only.

Property	Data type	Values	Description
RecNo	LongInt	N/A	Reports the position of a record. (This can be a time-consuming operation for dBASE tables.)
Refresh	Logical	True, False	Reports when data displayed onscreen is being changed, either across a network, by an ObejctPAL statement, or user action.
Required	Logical	True, False	Reports whether a field object must be assigned a value for the record to be valid.
RowNo	SmallInt	N/A	Reports the row number of a record displayed in a table frame, mutli-record object, or table view, starting with 1.
Scroll	Point	N/A	How far you've scrolled.
SelectedText	String	N/A	Returns the selected text in a field object.
SeqNo	LongInt	N/A	The actual sequence number of a record as displayed, taking filters and indexes into account.
Shrinkable	Logical	True, False	Specifies whether a report band can be shrunk.
Size	Point	N/A	Specifies the coordinates of the lower left corner of a design object.
SizeToFit	Logical	True, False	When True, the form opens at the size of the underlying page. When False, the form opens at the size it was saved.
SortOrder	Logical	True, False	Specifies the sort order of a report. True = Descending order, False = Ascending order.
StandardMenu	Logical	True, False	Specifies whether a form uses the standard menus.
Start	Point	N/A	Specifies the coordinates of the start of a line. See also: End.
Style	SmallInt	BorlandButton, WindowsButton	Reports or specifes the display style of a button.
TabStop	Logical	True, False	Specifies whether a field object is a tab stop.
TableName	String	N/A	Specifies the name of a table to which a design object is bound.
Text	String	N/A	Specifies the characters displayed in a text object.

Property	Data type	Values	Description
ThickFrame	Logical	True, False	When True, and DialogForm and Border also True, specifies a thick window frame instead of the usual pixel-wide frame.
Thickness	SmallInt	LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint	Specifies the thickness of a line.
Title	String	N/A	Specifies the text of a form's caption.
TopLine	LongInt	N/A	The number of the line currently displayed at the top of a text object.
Touched	Logical	True, False	True when user has made changes not yet committed.
Translucent	Logical	True, False	Specifies whether the color of an object is translucent.
Value	String	N/A	Specifies the value of a design object.
VerticalScrollBar	Logical	True, False	Specifies whether an object has a vertical scroll bar. Not valid for all UIObjects. Refer to chart.
Visible	Logical	True, False	Specifies whether an object is displayed.
WordWrap	Logical	True, False	Specifies whether to wrap lines that exceed the width of a field object.

Table F-3 Properties unique to graph objects

Property	Data type	Values
BackWall.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BackWall.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Property	Data type	Values
BackWall.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
Background.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Background.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Background.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
BaseFloor.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Property	Data type	Values
BaseFloor.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
BaseFloor.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
BindType	SmallInt	Graph1DSummary, Graph2DSummary, GraphTabular
CurrentSeries	SmallInt	N/A
CurrentSlice	SmallInt	N/A
GraphType	SmallInt	Graph2DArea, Graph2DBar, Graph2DColumns, Graph2DLine, Graph2DPie, Graph2DRotatedBar, Graph2DStackedBar, Graph3DArea, Graph3DBar, Graph3DColumns, Graph3DPie, Graph3DRibbon, Graph3DRotatedBar, Graph3DStackedBar, Graph3DStep, Graph3DSurface, Graph3DXY
Label.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Label.Font.Size	SmallInt	Depends on system
Label.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
Label.Font.Typeface	String	Depends on system
Label.LabelFormat	SmallInt	GraphHideY, GraphPercent, GraphShowY

Property	Data type	Values
Label.LabelLocation	SmallInt	LabelAbove, LabelBelow, LabelBottom, labelCenter, LabelLeft, LabelMiddle, LabelRight, LabelTop
Label.NumberFormat	N/A	Format specification
LeftWall.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LeftWall.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LeftWall.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
LegendBox.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Font.Size	SmallInt	Depends on system
LegendBox.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
LegendBox.Font.Typeface	String	Depends on system
LegendBox.LegendPos	SmallInt	LegendLeft, LegendRight

Property	Data type	Values
LegendBox.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
LegendBox.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
MaxGroups	SmallInt	Depends on graph
MaxXValues	SmallInt	Depends on graph
MinXValues	SmallInt	Depends on graph
Options.Elevation	SmallInt	0 to 90 degrees
Options.Rotation	SmallInt	0 to 90 degrees
Options.ShowAxes	Logical	True, False
Options.ShowGrid	Logical	True, False
Options.ShowLabels	Logical	True, False
Options.ShowLegend	Logical	True, False
Options.ShowTitle	Logical	True, False
Series.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Graph_Title.Font.Size	SmallInt	Depends on system
Series.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
Series.Graph_Title.Font.Typeface	String	Depends on system

Property	Data type	Values
Series.Graph_Title.Text	String	N/A
Series.Graph_Title.UseDefault	Logical	True, False
Series.Line.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Line.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
Series.Line.Thickness	SmallInt	LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint
Series.Marker	SmallInt	MarkerBoxedCross, MarkerBoxed_Plus, MarkerCross, MarkerFilledBox, MarkerFilledCircle, MarkerFilledDownTriangle, MarkerFilledTriangle, MarkerFilledTriangles, MarkerHollowBox, MarkerHollowCircle, MarkerHollowDownTriangle, MarkerHollowTriangle, MarkerHollowTriangles, MarkerHorizontalLine, MarkerPlus, MarkerVerticalLine
Series.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Series.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
Series.TypeOverride	SmallInt	Graph2DArea, Graph2DBar, Graph2DLine, None

Property	Data type	Values
Slice.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Slice.Explode	Logical	
Slice.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Slice.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
TitleBox.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Graph_Title.Font.Size	SmallInt	Depends on system
TitleBox.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
TitleBox.Graph_Title.Font.Typeface	String	Depends on system
TitleBox.Graph_Title.Text	String	N/A
TitleBox.Graph_Title.UseDefault	Logical	True, False
TitleBox.Pattern.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Property	Data type	Values
TitleBox.Pattern.Style	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotsPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotsPattern, MaximumDotsPattern, MinimumDotsPattern, MediumDotsPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredLinesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, WeavePattern, ZigZagPattern
TitleBox.Subtitle.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
TitleBox.Subtitle.Font.Size	SmallInt	Depends on system
TitleBox.Subtitle.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
TitleBox.Subtitle.Font.Typeface	String	Depends on system
TitleBox.Subtitle.Text	String	N/A
TitleBox.Subtitle.UseDefault	Logical	True, False
XAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
XAxis.Graph_Title.Font.Size	SmallInt	Depends on system
XAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
XAxis.Graph_Title.Font.Typeface	String	Depends on system
XAxis.Graph_Title.Text	String	N/A
XAxis.Graph_Title.UseDefault	Logical	True, False
XAxis.Scale.AutoScale	Logical	True, False
XAxis.Scale.HighValue	Number	Depends on graph
XAxis.Scale.Increment	Number	Depends on graph
XAxis.Scale.Logarithmic	Logical	True, False
XAxis.Scale.LowValue	Number	Depends on graph

Property	Data type	Values
XAxis.Ticks.Alternate	Logical	True, False
XAxis.Ticks.DateFormat	N/A	Format specification
XAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
XAxis.Ticks.Font.Size	SmallInt	Depends on system
XAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
XAxis.Ticks.Font.Typeface	String	Depends on system
XAxis.Ticks.NumberFormat	N/A	Format specification
YAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
YAxis.Graph_Title.Font.Size	SmallInt	Depends on system
YAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
YAxis.Graph_Title.Font.Typeface	String	Depends on system
YAxis.Graph_Title.UseDefault	Logical	True, False
YAxis.Scale.AutoScale	Logical	True, False
YAxis.Scale.HighValue	Number	Depends on graph
YAxis.Scale.Increment	Number	Depends on graph
YAxis.Scale.Logarithmic	Logical	True, False
YAxis.Scale.LowValue	Number	Depends on graph
YAxis.Ticks.Alternate	Logical	True, False
YAxis.Ticks.DateFormat	N/A	Format specification
YAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
YAxis.Ticks.Font.Size	SmallInt	Depends on system
YAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
YAxis.Ticks.Font.Typeface	String	Depends on system
YAxis.Ticks.Number.Format	N/A	Format specification
ZAxis.Graph_Title.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Property	Data type	Values
ZAxis.Graph_Title.Font.Size	SmallInt	Depends on system
ZAxis.Graph_Title.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
ZAxis.Graph_Title.Font.Typeface	String	Depends on system
ZAxis.Graph_Title.UseDefault	Logical	True, False
ZAxis.Scale.AutoScale	Logical	True, False
ZAxis.Scale.HighValue	Number	Depends on graph
ZAxis.Scale.Increment	Number	Depends on graph
ZAxis.Scale.Logarithmic	Logical	True, False
ZAxis.Scale.LowValue	Number	Depends on graph
ZAxis.Ticks.Alternate	Logical	True, False
ZAxis.Ticks.Font.Color	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
ZAxis.Ticks.Font.Size	SmallInt	Depends on system
ZAxis.Ticks.Font.Style	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
ZAxis.Ticks.Font.Typeface	String	Depends on system

APPENDIX G

Object PAL Constants

Table G-1 ActionClasses

Constant	Data type	Description
DataAction	SmallInt	Data actions are for navigating in a table and for such tasks as record locking and record posting.
EditAction	SmallInt	Most Edit actions alter data within a field
FieldAction	SmallInt	Field actions are a special category of Move action that enable movement between field objects.
MoveAction	SmallInt	Move actions have to do with moving within a field object
SelectAction	SmallInt	Select actions are equivalent to Move actions

Table G-2 ActionDataCommands

Constant	Data type	Description
DataArriveRecord	SmallInt	Indicates a change to the current record, regardless of the reason. Some possible reasons: navigation, editing, network refresh, and scrolling.
DataBegin	SmallInt	Moves to first record in the table associated with the given UIObject. Will force recursive action (DataUnlockRecord) if current record has been modified. If Error encountered, will call **error** method. Invoked by VCR "first record" button, Record|First menu action.
DataBeginEdit	SmallInt	Used to enter Edit mode on the form. Normally always allowed.
DataBeginFirstField	SmallInt	Moves to the first field in the first record of the table associated with the given UIObject.
DataCancelRecord	SmallInt	Used to discard changes to record. Succeeds by default, but user could block it. Invoked by Edit|Undo, *Alt+Backspace*, or Record|Cancel Changes menu item. Also used internally when moving off a locked but unmodified record.
DataDeleteRecord	SmallInt	Deletes the current record. Errors encountered will call **error** method. This action is irreversible except for dBASE tables. Invoked by Record|Delete or *Ctrl+Del.*
DataDesign	SmallInt	Switches from running the form into Design mode. Invoked by *F8*

Constant	Data type	Description
DataDitto	SmallInt	Copies into the current record all the fields from the prior record. Invoked by *Ctrl+D*.
DataEnd	SmallInt	Moves to final record in the table associated with the given UIObject. Will force recursive action(DataUnlockRecord) if current record has been modified. Error encountered will call **error** method. Invoked by SpeedBar "last record" button.
DataEndEdit	SmallInt	Used to exit Edit mode on the form. Invoked by (second) *F9* Edit Data on the SpeedBar, or Form|End Edit menu action.
DataEndLastField	SmallInt	Moves to the last field of the last record of the table associated with a UIObject.
DataFastBackward	SmallInt	Moves backward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO.
DataFastForward	SmallInt	Moves forward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO.
DataHideDeleted	SmallInt	Alters the mode of the form so that deleted records will be hidden (available only for dBASE tables). Invoked by Form|Hide Deleted.
DataInsertRecord	SmallInt	Will insert a new (blank) record before the current record. Record state will appear as "locked", and blank record will not exist in the underlying table until the record is eventually modified and unlocked. Invoked by Record|Insert, or *Ins*. Note that records created in this way can be discarded either via DataDeleteRecord or DataCancelRecord before they have been unlocked. Moving off such a record without making any changes will internally use DataCancelRecord to discard it.
DataLockRecord	SmallInt	Used to lock the current record. Errors encountered will call **error** method. Invoked by *F5*.
DataLookup	SmallInt	Used to invoke lookup table for current field, to accept user's choice of new value, and, if appropriate, to update all corresponding fields governed by lookup. Available only for fields that have been defined as lookup fields. Invoked by *F6*.
DataLookupMove	SmallInt	A special form of lookup which allows the user to choose a new master record for this detail.
DataNextRecord	SmallInt	Moves (if possible) to next sequential record in the table associated with the UIObject. Will force recursive action(DataUnlockRecord) if current record has been modified. Errors encountered will call **error** method. Invoked by Record|Next menu choice, SpeedBar "next record" button, *F12*, and so forth.
DataNextSet	SmallInt	Moves forward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO.
DataPostRecord	SmallInt	Just like DataUnlockRecord, but the record lock will not be released. As a consequence, if changes to key fields mean the record will move to a new position in the table, the table's position "flies with" that record (meaning it will still be the current record). Invoked by *Ctrl+F5*.
DataPrint	SmallInt	Prints a Form or Table window.

Constant	Data type	Description
DataPriorRecord	SmallInt	Moves (if possible) to previous record in the table associated with the UIObject. Will force recursive action(DataUnlockRecord) if current record has been modified. Errors encountered will call **error** method. Invoked by Record|Previous menu choice, SpeedBar "prior record" button, *F11*, and so forth.
DataPriorSet	SmallInt	Moves backward one "set" of records (where a "set" is defined as the number of rows in a table frame or MRO, or 1 in the case of a single record form). Will force recursive action(DataUnlockRecord) if current record has been modified. Errors encountered will call **error** method. Invoked by *PgUp*, Record|Prior Set, *Shift+F11*, and so forth.
DataRecalc	SmallInt	Forces a calculated field to recalcualte its value.
DataRefresh	SmallInt	Specifies a refresh of a value in a record displayed on the screen.
DataRefreshOutside	SmallInt	Specifies a refresh of a value in a record not displayed on the screen.
DataSaveCrosstab	SmallInt	Writes given crosstab to CROSSTAB.DB.
DataSearch	SmallInt	Brings up a dialog box to allow the user to search for a specific value within a specified field. Invoked by Record|Locate|Value or *Ctrl+Z*.
DataSearchNext	SmallInt	Will search for the next record containing the value last specified in response to the last DataSearch action. Invoked by Record|Locate Next or *Ctrl+A*.
DataSearchReplace	SmallInt	Brings up a dialog to allow the user to search for a specific value within a specified field and to replace it with a different value. Invoked by Record|Locate and Replace, or *Ctl+Shift+Z*.
DataShowDeleted	SmallInt	Alters the mode of the form so that deleted records will be shown (available only for dBASE tables). They will look no different from normal records, but status line will reflect their state. Invoked by Form|Show Deleted.
DataTableView	SmallInt	Used to bring up a Table View of the master table of a form. If this form was originally invoked as preferred form of existing Table View, this just returns focus to that Table View. Invoked by *F7* Table View on the SpeedBar or Form|Table View.
DataToggleDeleted	SmallInt	Used to reverse the state of "show deleted records" for dBASE tables.
DataToggleDeleteRecord	SmallInt	Used to reverse the deleted state of records in dBASE tables.
DataToggleEdit	SmallInt	Used to reverse the Edit state of the form. Recursively calls action(DataBeginEdit) or action(DataEndEdit) as appropriate. Invoked by *F9* Edit Data on the SpeedBar.
DataToggleLockRecord	SmallInt	Used to reverse the lock state of the current record. Actually just recursively uses action(DataLockRecord) or action(DataUnlockRecord) as appropriate. Errors encountered will call **error** method.
DataUnDeleteRecord	SmallInt	For dBASE tables, will match previously deleted record as "undeleted."
DataUnlock Record	SmallInt	Used to commit the record modifications to the table and then (if successful) to unlock the record. Error encountered will call **error** method.

Table G-3 ActionEditCommands

Constant	Data type	Description
EditCommitField	SmallInt	Write current field's modifications to record buffer (without leaving field).
EditCopySelection	SmallInt	Copies selected area of text to Clipboard.
EditCopyToFile	SmallInt	Invokes a dialog box to copy selection to a file.
EditCutSelection	SmallInt	Copies selected area of text to Clipboard and deletes it.
EditDeleteBeginLine	SmallInt	Deletes from current position to beginning of line.
EditDeleteEndLine	SmallInt	Deletes from current position to end of line.
EditDeleteLeft	SmallInt	Deletes one character position to the left. Invoked by *Backspace* in Field View.
EditDeleteLeftWord	SmallInt	Deletes up to and including beginning of word to the left of current character position. Invoked by *Ctrl+Backspace*.
EditDeleteLine	SmallInt	Deletes line on which current position is found.
EditDeleteRight	SmallInt	Deletes one character position to the right. Invoked by *Del* in Field View.
EditDeleteRightWord	SmallInt	Deletes up to and including end of word to the right of current character position.
EditDeleteSelection	SmallInt	Deletes currently selected area of text.
EditDeleteWord	SmallInt	Deletes word around the current position.
EditDropDownList	SmallInt	Used by drop-down edit fields. Will drop down the associated pick list.
EditEnterFieldView	SmallInt	Enters Field View for current field (allowing arrow keys to move around within the field). Begins by moving current position to end of field and dehighlighting it.
EditEnterMemoView	SmallInt	Enters "memo" view on memos or OLE fields.
EditEnterPersistFieldView	SmallInt	Enters "persistent Field View," meaning arrow keys always move within character positions within a field, even when moving to new fields.
EditExitFieldView	SmallInt	Exits Field View (meaning arrow keys will move between fields again) and highlights entire field.
EditExitMemoView	SmallInt	Exits "memo" view on memos or OLE fields, meaning *Enter* and *Tab* will once again move between fields.
EditExitPersistField View	SmallInt	Exits "presistent Field View" meaning arrow keys move between fields.
EditHelp	SmallInt	Invokes the help subsystem. Invoked by *F1*.
EditInsertBlank	SmallInt	Inserts a blank character at current position.
EditInsertLine	SmallInt	Inserts a blank line at current position.
EditLaunchServer	SmallInt	Used only by OLE fields, will invoke the server application appropriate for current field.
EditPaste	SmallInt	Paste from the Clipboard to current position (replacing current selection if appropriate).
EditPasteFromFile	SmallInt	Invokes a dialog box, allowing user to select file to insert at current position.
EditProperties	SmallInt	Invokes the property inspection menu for given object. Currently only graphic fields and formatted memo fields support this.
EditReplace	SmallInt	Toggles overstrike mode in a field object.

Constant	Data type	Description
EditTextSearch	SmallInt	Invokes a dialog box to allow user to search and replace text within current field.
EditToggleFieldView	SmallInt	Reverses current state of Field View. Recursively calls action(EditEnterFieldView) or action(EditExitFieldView). Invoked by *F2* Field View on the SpeedBar.
EditUndoField	SmallInt	Discards current fields modifications and reverts to value in current record buffer. Invoked by *Esc*.

Table G-4 ActionFieldCommands

Constant	Data type	Description
FieldBackward	SmallInt	Used to move one field backward in tab order. This will search for the prior UIObject marked as a "Tab Stop" in left-right/top-down order. Invoked by *Shift+Tab*.
FieldDown	SmallInt	Used to move to field below current field, whether in Field View or not. Invoked by *Alt+↓*.
FieldEnter	SmallInt	Used to commit modifications to a field (if any) and to move one field forward in tab order. Invoked by *Enter*.
FieldFirst	SmallInt	Used to move to first field within a record. Invoked by *Alt+Home* or by *Home* (when not in Field View).
FieldForward	SmallInt	Used to move one field forward in tab order. This will search for the next UIObject marked as a "Tab Stop" in left-right/top-down order. Invoked by *Tab*.
FieldGroupBackward	SmallInt	Used to move one "super" tab group backward (for example, between different table frames on the same form). Invoked by *F3*.
FieldGroupForward	SmallInt	Used to move one "super" tab group forward (for example, between different table frames on the same form). Invoked by *F4*.
FieldLast	SmallInt	Used to move to last field within a record. Invoked by *Alt+End* or by *End* (when not in Field View).
FieldLeft	SmallInt	Used to move to field to left of current field.
FieldNextPage	SmallInt	Used to move to next sequential page in multi-page form. Invoked by Form|Page|Next or *Shift+F4*.
FieldPriorPage	SmallInt	Used to move to prior page in multi-page form. Invoked by Form|Page|Previous or *Shift+F3*.
FieldRight	SmallInt	Used to move to field to right of current field, whether in Field View or not. Invoked by *Alt+→*.
FieldRotate	SmallInt	Used to rotate columns within a table frame. Invoked by *Ctrl+R*.
FieldUp	SmallInt	Used to move to field above current field, whether in Field View or not. Invoked by *Alt+↑*.

Table G-5 ActionMoveCommands

Constant	Data type	Description
MoveBegin	SmallInt	In Memo View, moves to beginning of document. Otherwise, moves to first field in first record of table. Invoked by *Ctrl+Home*.
MoveBeginLine	SmallInt	In Memo View, moves to beginning of line; otherwise, moves to first field in record. Invoked by *Home*.
MoveBottom	SmallInt	In Memo View, moves to bottom line of the text region. Otherwise, moves to last record in table.
MoveBottomLeft	SmallInt	In Memo View, moves to beginning of last line on screen. Invoked by *Ctrl+PgUp*.
MoveBottomRight	SmallInt	In Memo View, moves to end of last line on screen. Invoked by *Ctrl+PgDn*.
MoveDown	SmallInt	Moves down as appropriate. In Memo View, moves down one line on multiline fields. Otherwise, moves to next "Tab Stop" object below current object. Table frame objects move to next record. Invoked by ↓.
MoveEnd	SmallInt	In Memo View, moves to end of document; otherwise, moves to last field in last record of table. Invoked by *Ctrl+End*.
MoveEndLine	SmallInt	In Memo View, moves to end of line; otherwise, moves to last field in record. Invoked by *End*.
MoveLeft	SmallInt	Moves left as appropriate. In Memo View, moves one character position left; otherwise, moves to next Tab Stop object to left of current object. Invoked by ←.
MoveLeftWord	SmallInt	In Memo View, moves insertion point to beginning of word to the left of current insertion point. Invoked by *Ctrl*+←.
MoveRight	SmallInt	Moves right as appropriate. In Memo View, moves one character position right; otherwise, moves to next Tab Stop object to right of current object. Invoked by →.
MoveRightWord	SmallInt	In Memo View, moves insertion point to beginning of word to the right of current insertion point. Invoked by *Ctrl*+→.
MoveScrollDown	SmallInt	Scrolls image down (effectively moving viewing area up) by appropriate amount. Active fields scroll by even lines of text. Tables move to new record. In Memo View, scroll toward the bottom of the text. The insertion point remains on the same line of the display region unless the last line of the text is visible, in which case the insertion point moves down one line until the last line is reached. Invoked by *Ctrl*+↓
MoveScrollLeft	SmallInt	Scrolls image to left (effectively moving viewing area to the right) by appropriate amount. Active fields scroll roughly one character position. Tables move to new column.
MoveScrollPageDown	SmallInt	Scrolls image down (effectively moving viewing area up) by logical size of object (for example, complete page of document).

Constant	Data type	Description
MoveScrollPageLeft	SmallInt	Scrolls image left (effectively moving viewing area right) by logical size of object (for example, complete page of document).
MoveScrollPageRight	SmallInt	Scrolls image right (effectively moving viewing area left) by logical size of object (for example, complete page of document).
MoveScrollPageUp	SmallInt	Scrolls image up (effectively moving viewing area down) by logical size of object (for example, complete page of document).
MoveScrollRight	SmallInt	Scrolls image to right (effectively moving viewing area to the left) by appropriate amount. Active fields scroll roughly one character position. Tables move to new column.
MoveScrollScreenDown	SmallInt	Scrolls image down (effectively moving viewing area up) by size of viewing area (for example, size of field). In Memo View, moves down in the document by the height of the display area. Invoked by *PgDn*.
MoveScrollScreenLeft	SmallInt	Scrolls image left (effectively moving viewing area right) by size of viewing area (for example, size of field).
MoveScrollScreenRight	SmallInt	Scrolls image right (effectively moving viewing area left) by size of viewing area (for example, size of field).
MoveScrollScreenUp	SmallInt	Scrolls image up (effectively moving viewing area down) by size of viewing area (for example, size of field). In Memo View, moves up in the document by the height of the display area. Invoked by *PgUp*.
MoveScrollUp	SmallInt	Scroll image up (effectively moving viewing area down) by appropriate amount. Active fields scroll by even lines of text. In Memo View, scroll toward the top of the document by one line of text. The insertion point stays at the same line position unless the top line of the document is visible, in which case the insertion point moves up one line if it can. Invoked by *Ctrl+↑*
MoveTop	SmallInt	In Memo View, move the insertion point to the first line of text visible in the display region; otherwise, moves to first record in table.
MoveTopLeft	SmallInt	In Memo View, moves to the top left of the display region; otherwise, moves to top left field. Invoked by *Ctrl+PgUp*.
MoveTopRight	SmallInt	In Memo View, moves to the top right of the display region; otherwise, moves to top right field. Invoked by *Ctrl+PgDn*.
MoveUp	SmallInt	Moves up as appropriate. In Memo View, moves up one line on multiline fields; otherwise, it moves to next Tab Stop object above current object. Table frame objects move to prior record. Invoked by *↑*.

Table G-6 ActionSelectCommands

Constant	Data type	Description
SelectBegin	SmallInt	In Memo View, selects from current position to beginning of document; otherwise, selects from current position to first field in first record of table. Invoked by *Shift+Ctrl+Home*.
SelectBeginLine	SmallInt	In Memo View, selects from current position to beginning of line; otherwise, selects from current position to first field in record. Invoked by *Shift+Home*.
SelectBottom	SmallInt	In Field View and Memo View, select from current position to bottom of the display region; otherwise, selects from current position to last record in table.
SelectBottomLeft	SmallInt	In Memo View, selects from current position to beginning of last line in display region. Invoked by *Shift+Ctrl+PgUp*.
SelectBottomRight	SmallInt	In Memo View, selects from current position to end of last line in display region. Invoked by *Shift+Ctrl+PgDn*.
SelectDown	SmallInt	Selects down as appropriate. In Field View or Memo View, selects down one line on multiline fields. Cannot extend selection across fields in forms. Table frame objects select to next record. Invoked by *Shift+↓*.
SelectEnd	SmallInt	In Field View or Memo View, selects from current position to end of document; otherwise, selects from current position to last field in last record of table. Invoked by *Shift+Ctrl+End*.
SelectEndLine	SmallInt	In Field View or Memo View, selects from current position to end of line; otherwise, selects from current position to last field in record. Invoked by *Shift+End*.
SelectLeft	SmallInt	Selects left as appropriate. In Field View or Memo View, selects one character position left; otherwise, selects next Tab Stop object to left of current object. Invoked by *Shift+←*.
SelectLeftWord	SmallInt	In Field View or Memo View, if the insertion point is between words, selects word to the left of insertion point. If the insertion point is within a word, selects to the beginning of that word. Invoked by *Shift+Ctrl+←*.
SelectRight	SmallInt	Selects right as appropriate. In Field View or Memo View, selects one character position right. Invoked by *Shift+→*.
SelectRightWord	SmallInt	In Field View or Memo View, selects to the beginning of the next following word. If the insertion point precedes one or more spaces or tabs, selection only includes those spaces or tabs. Invoked by *Shift+Ctrl+→*.
SelectScrollDown	SmallInt	Selects image down (effectively moving viewing area up) by appropriate amount. Active fields select even lines of text. Tables select new record. Invoked by *Shift+Crtl+↓*.
SelectScrollLeft	SmallInt	Selects image on left (effectively moving viewing area to the right) by appropriate amount. Active fields select roughly one character position. Tables select to new column.
SelectScrollPageDown	SmallInt	Selects image down (effectively moving viewing area up) by logical size of object (for example, complete page of document).

Constant	Data type	Description
SelectScrollPageLeft	SmallInt	Selects image left (effectively moving viewing area right) by logical size of object (for example, complete page of document).
SelectScrollPageRight	SmallInt	Selects image right (effectively moving viewing area left) by logical size of object (for example, complete page of document).
SelectScrollPageUp	SmallInt	Selects image up (effectively moving viewing area down) by logical size of object (for example, complete page of document).
SelectScrollRight	SmallInt	Selects image on right (effectively moving viewing area to the left) by appropriate amount. Active fields select roughly one character position. Tables select new column.
SelectScrollScreenDown	SmallInt	Selects image down (effectively moving viewing area up) by size of viewing area (for example, size of field). Invoked by *Shift+Crtl+PgDn.*
SelectScrollScreenLeft	SmallInt	Selects image left (effectively moving viewing area right) by size of viewing area (for example, size of field).
SelectScrollScreenRight	SmallInt	Selects image right (effectively moving viewing area left) by size of viewing area (for example, size of field).
SelectScrollScreenUp	SmallInt	Selects image up (effectively moving viewing area down) by size of viewing area (for example, size of field). Invoked by *Shift+Ctrl+PgUp.*
SelectScrollUp	SmallInt	Select image up (effectively moving viewing area down) by appropriate amount. Active fields select by even lines of text.
SelectSelectAll	SmallInt	Selects the entire field.
SelectTop	SmallInt	In Field View or Memo View, selects from current position to top of display region; otherwise, selects from current position to first record in table.
SelectTopLeft	SmallInt	In Field View or Memo View, selects from current position to beginning of screen; otherwise, selects from current position to top left field. Invoked by *Shift+Crtl+PgUp.*
SelectTopRight	SmallInt	In Field View or Memo View, selects from current position to end of top line of screen; otherwise, selects from current position to top right field. Invoked by *Shift+Crtl+PgDn.*
SelectUp	SmallInt	Selects up as appropriate. In Field View or Memo View, selects up one line on multiline fields; otherwise, it selects next Tab Stop object above current object. Table frame objects select to prior record. Invoked by *Shift*+↑.

Table G-7 ButtonStyles

Constant	Data type	Description
BorlandButton	SmallInt	Gives a button the 3D look of buttons in Borland products.
WindowsButton	SmallInt	Gives a button the look of buttons in other Windows products.

Table G-8 ButtonTypes

Constant	Data type	Description
CheckboxType	SmallInt	Displays a button as a check box.
PushButtonType	SmallInt	Displays a button as a push button.
RadioButtonType	SmallInt	Displays a button as a radio button.

Table G-9 Colors

Constant	Data type
Black	LongInt
Blue	LongInt
Brown	LongInt
DarkBlue	LongInt
DarkCyan	LongInt
DarkGray	LongInt
DarkGreen	LongInt
DarkMagenta	LongInt
DarkRed	LongInt
Gray	LongInt
Green	LongInt
LightBlue	LongInt
Magenta	LongInt
Red	LongInt
Translucent	LongInt
Transparent	LongInt
White	LongInt
Yellow	LongInt

Table G-10 CompleteDisplay

Constant	Data type	Description
DisplayAll	SmallInt	Specifies CompleteDisplay for all field objects in the form.
DisplayCurrent	SmallInt	Specifies CompleteDisplay for the current field object.

Table G-11 ErrorReasons

Constant	Data type	Description
ErrorCritical	SmallInt	Displays a message in a model dialog box.
ErrorWarning	SmallInt	Displays a message in the status area.

Table G-12 EventErrorCodes

Constant	Data type	Description
CanArrive	SmallInt	Grants permission to arrive at an object.
CanDepart	SmallInt	Grants permission to leave an object.
CanNotArrive	SmallInt	Grants permission to arrive at an object (blocks the move).
CanNotDepart	SmallInt	Grants permission to leave an object (blocks the move).

Table G-13 ExecuteOptions

Constant	Data type	Description
ExeHidden	SmallInt	Hides the application window and passes activation to another window.
ExeMinimized	SmallInt	Minimizes the application window and activates the top-level window in the window-manager's list.
ExeShowMaximized	SmallInt	Activates the application window and displays it as a maximized window.
ExeShowMinimized	SmallInt	Activates the application window and displays it minimized (as an icon).
ExeShowMinimizedNoActivate	SmallInt	Displays the application as an icon. The window that is currently active remains active.
ExeShowNoActivate	SmallInt	Displays the application window at its most recent size and position. The currently active window remains active.
ExeShowNormal	SmallInt	Activates and displays a window

Table G-14 FieldDisplayTypes

Constant	Data type	Description
BitmapField	SmallInt	Enables a field object to display a bitmap.
CheckboxField	SmallInt	Displays a field as a check box.
ComboField	SmallInt	Displays a field as a drop-down edit list (also called a combo box).
EditField	SmallInt	Displays an unlabeled field.
LabeledField	SmallInt	Displays a labeled field.
ListField	SmallInt	Displays a list box.
OleField	SmallInt	Enables a field to contain OLE data.
RadioButtonField	SmallInt	Displays a field as one or more radio buttons.

Table G-15 FileBrowserFileTypes

Constant	Data type	Description
fbASCII	LongInt	Plain text files.
fbAllTables	LongInt	All table types supported by Paradox.

Constant	Data type	Description
fbBitmap	LongInt	Bitmap graphics.
fbDBase	LongInt	dBASE tables.
fbExcel	LongInt	Excel worksheets.
fbFiles	LongInt	All files.
fbForm	LongInt	Paradox forms.
fbGraphic	LongInt	Graphic files.
fbIni	LongInt	File whose extension is .INI
fbLibrary	LongInt	ObjectPAL libraries.
fbLotus1	LongInt	Lotus 1-2-3 version 1 worksheets.
fbLotus2	LongInt	Lotus 1-2-3 version 2 worksheets.
fbParadox	LongInt	Paradox tables.
fbQuattro	LongInt	Quattro worksheets.
fbQuattroPro	LongInt	Quattro Pro worksheets.
fbQuattroProWindows	LongInt	Quattro Pro for Windows notebooks.
fbQuery	LongInt	Query files.
fbReport	LongInt	Paradox reports.
fbScript	LongInt	ObjectPAL scripts.
fbTable	LongInt	Paradox tables.
fbTableView	LongInt	Paradox table view files.
fbText	LongInt	All text files.

Table G-16 FontAttributes

Constant	Data type	Description
FontAttribBold	SmallInt	Example: **bold**
FontAttribItalic	SmallInt	Example: *italic*
FontAttribNormal	SmallInt	Example: normal
FontAttribStrikeOut	SmallInt	Example: ~~strike out~~
FontAttribUnderline	SmallInt	Example: underline

Table G-17 FrameStyles

Constant	Data type	Description
DashDotDotFrame	SmallInt	A repeating sequence of one dash followed by two dots.
DashDotFrame	SmallInt	A repeating sequence of one dash followed by one dot.
DashedFrame	SmallInt	A repeating sequence of dashes.
DottedFrame	SmallInt	A repeating sequence of dots.
DoubleFrame	SmallInt	Two concentric boxes.
Inside3DFrame	SmallInt	The frame appears pushed into the form.
NoFrame	SmallInt	No frame.
Outside3DFrame	SmallInt	The frame appears popped out of the form.
ShadowFrame	SmallInt	A drop shadow.
SolidFrame	SmallInt	A single solid box (no dashes or dots).
WideInsideDoubleFrame	SmallInt	Two concentric boxes; the inside box is wide.
WideOutsideDoubleFrame	SmallInt	Two concentric boxes; the outside box is wide.

Table G-18 General

Constant	Data type	Description
No	Logical	False
Off	Logical	False
On	Logical	True
PI	Number	3.14159265358979323846
Yes	Logical	True

Table G-19 GraphBindTypes

Constant	Data type	Description
Graph1DSummary	SmallInt	Specifies a one-dimensional summary graph. Enables summary operators.
Graph2DSummary	SmallInt	Specifies a two-dimensional summary graph. Enables summary operators and group-by specification.
GraphTabular	SmallInt	Specifies a tabular graph (default).

Table G-20 GraphLabelFormats

Constant	Data type	Description
GraphHideY	SmallInt	Hide Y-value (2-D and 3-D Pie and Column graphs only).
GraphPercent	SmallInt	Display Y-value as a percent (2-D and 3-D Pie and Column graphs only).
GraphShowY	SmallInt	Display Y-value in the units used in the table (2-D and 3-D Pie and Column graphs only).

Table G-21 GraphLabelLocation

Constant	Data type	Description
LabelAbove	SmallInt	Display the label above the marker.
LabelBelow	SmallInt	Display the label below the marker.
LabelBottom	SmallInt	Display the label inside the column or bar at the bottom.
LabelCenter	SmallInt	Display the label inside the marker at the center.
LabelLeft	SmallInt	Display the label to the left of the marker.
LabelMiddle	SmallInt	Display the label inside the column or bar in the middle.
LabelRight	SmallInt	Display the label to the right of the marker.
LabelTop	SmallInt	Display the label inside the column or bar at the top.

Table G-22 GraphLegendPosition

Constant	Data type	Description
LegendCenter	SmallInt	Display the legend centered below the graph.
LegendLeft	SmallInt	Display the legend to the left of the graph.

Table G-23 GraphMarkers

Constant	Data type	Description
MarkerBoxedCross	SmallInt	Marker is a box with a cross in it.
MarkerBoxed_Plus	SmallInt	Marker is a box with a plus sign in it.
MarkerCross	SmallInt	Marker is a cross.
MarkerFilledBox	SmallInt	Marker is a filled box.
MarkerFilledCircle	SmallInt	Marker is a filled circle.
MarkerFilledDownTriangle	SmallInt	Marker is a filled triangle pointing down.
MarkerFilledTriangle	SmallInt	Marker is a filled triangle pointing up.
MarkerFilledTriangles	SmallInt	Marker is two filled triangles pointing at each other.
MarkerHollowBox	SmallInt	Marker is a hollow (unfilled) box.
MarkerHollowCircle	SmallInt	Marker is a hollow circle.
MarkerHollowDownTriangle	SmallInt	Marker is a hollow triangle pointing down.
MarkerHollowTriangle	SmallInt	Marker is a hollow triangle pointing up.
MarkerHollowTriangles	SmallInt	Marker is two hollow triangles pointing at each other.
MarkerHorizontalLine	SmallInt	Marker is a horizontal line.
MarkerPlus	SmallInt	Marker is a plus sign.
MarkerVerticalLine	SmallInt	Marker is a vertical line.

Table G-24 GraphTypeOverride

Constant	Data type	Description
GraphArea	SmallInt	Displays specified series as an area graph.
GraphBar	SmallInt	Displays specified series as a bar graph.
GraphDefault	SmallInt	Displays specified series in the default graph type.
GraphLine	SmallInt	Displays specified series as a line graph.

Table G-25 GraphTypes

Constant	Data type	Description
Graph2DArea	SmallInt	2-dimensional area graph.
Graph2DBar	SmallInt	2-dimensional bar graph.
Graph2DColumns	SmallInt	2-dimensional column graph.
Graph2DLine	SmallInt	2-dimensional line graph.
Graph2DPie	SmallInt	2-dimensional pie graph.
Graph2DRotatedBar	SmallInt	2-dimensional rotated bar graph.
Graph2DStackedBar	SmallInt	2-dimensional stacked bar graph.
Graph3DArea	SmallInt	3-dimensional area graph.
Graph3DBar	SmallInt	3-dimensional bar graph.
Graph3DColumns	SmallInt	3-dimensional column graph.
Graph3DPie	SmallInt	3-dimensional pie graph.
Graph3DRibbon	SmallInt	3-dimensional ribbon graph.
Graph3DRotatedBar	SmallInt	3-dimensional rotated bar graph.
Graph3DStackedBar	SmallInt	3-dimensional stacked bar graph.
Graph3DStep	SmallInt	3-dimensional step graph.
Graph3DSurface	SmallInt	3-dimensional surface graph.
GraphXY	SmallInt	XY graph.

Table G-26 GraphicMagnification

Constant	Data type	Description
Magnify100	SmallInt	Displays the graph at its actual size.
Magnify200	SmallInt	Displays the graph at twice its actual size.
Magnify25	SmallInt	Displays the graph at a quarter of its actual size.
Magnify400	SmallInt	Displays the graph at four times its actual size.
Magnify50	SmallInt	Displays the graph at half its actual size.
MagnifyBestFit	SmallInt	Resizes the graph as necessary to fit the graph in the frame.

Table G-27 IdRanges

Constant	Data type	Description
UserAction	SmallInt	The minimum value for a user-defined action constant.
UserActionMax	SmallInt	The maximum value for a user-defined action constant.
UserError	SmallInt	The minimum value for a user-defined error constant.
UserErrorMax	SmallInt	The maximum value for a user-defined error constant.
UserMenu	SmallInt	The minimum value for a user-defined menu ID constant.
UserMenuMax	SmallInt	The maximum value for a user-defined menu ID constant.

Table G-28 KeyBoardStates

Constant	Data type	Description
Alt	SmallInt	*Alt* is pressed.
Control	SmallInt	*Ctrl* is pressed.
LeftButton	SmallInt	The left mouse button is clicked.
RightButton	SmallInt	The right mouse button is clicked.
Shift	SmallInt	*Shift* is pressed.

Table G-29 Keyboard

Constant	Data type	Description
VK_ADD	SmallInt	Add key
VK_BACK	SmallInt	*Backspace* key
VK_CANCEL	SmallInt	Used for control-break processing
VK_CAPITAL	SmallInt	Capital key
VK_CLEAR	SmallInt	Clear key
VK_CONTROL	SmallInt	*Ctrl* key
VK_DECIMAL	SmallInt	Decimal key
VK_DELETE	SmallInt	*Delete* key
VK_DIVIDE	SmallInt	Divide key
VK_DOWN	SmallInt	↓ key
VK_END	SmallInt	*End* key
VK_ESCAPE	SmallInt	*Escape* key
VK_EXECUTE	SmallInt	Execute key
VK_F1	SmallInt	*F1* key
VK_F10	SmallInt	*F10* key
VK_F11	SmallInt	*F11* key
VK_F12	SmallInt	*F12* key
VK_F13	SmallInt	*F13* key
VK_F14	SmallInt	*F14* key

Constant	Data type	Description
VK_F15	SmallInt	*F15* key
VK_F16	SmallInt	*F16* key
VK_F2	SmallInt	*F2* key
VK_F3	SmallInt	*F3* key
VK_F4	SmallInt	*F4* key
VK_F5	SmallInt	*F5* key
VK_F6	SmallInt	*F6* key
VK_F7	SmallInt	*F7* key
VK_F8	SmallInt	*F8* key
VK_F9	SmallInt	*F9* key
VK_HELP	SmallInt	*Help* key
VK_HOME	SmallInt	*Home* key
VK_INSERT	SmallInt	*Insert* key
VK_LBUTTON	SmallInt	Left mouse button
VK_LEFT	SmallInt	← key
VK_MBUTTON	SmallInt	Middle mouse button (3-button mouse)
VK_MENU	SmallInt	Menu key
VK_MULTIPLY	SmallInt	Multiply key
VK_NEXT	SmallInt	*Page Down* key
VK_NUMLOCK	SmallInt	*Num Lock* key
VK_NUMPAD0	SmallInt	Key pad *0* key
VK_NUMPAD1	SmallInt	Key pad *1* key
VK_NUMPAD2	SmallInt	Key pad *2* key
VK_NUMPAD3	SmallInt	Key pad *3* key
VK_NUMPAD4	SmallInt	Key pad *4* key
VK_NUMPAD5	SmallInt	Key pad *5* key
VK_NUMPAD6	SmallInt	Key pad *6* key
VK_NUMPAD7	SmallInt	Key pad *7* key
VK_NUMPAD8	SmallInt	Key pad *8* key
VK_NUMPAD9	SmallInt	Key pad *9* key
VK_PAUSE	SmallInt	*Pause* key
VK_PRINT	SmallInt	OEM specific
VK_PRIOR	SmallInt	*Page Up* key
VK_RBUTTON	SmallInt	Right mouse button
VK_RETURN	SmallInt	*Return* key
VK_RIGHT	SmallInt	→ key
VK_SELECT	SmallInt	Select key
VK_SEPARATOR	SmallInt	Separator key
VK_SHIFT	SmallInt	*Shift* key

Constant	Data type	Description
VK_SNAPSHOT	SmallInt	Printscreen key for Windows 3.0 and later
VK_SPACE	SmallInt	Space
VK_SUBTRACT	SmallInt	Subtract key
VK_TAB	SmallInt	*Tab* key
VK_UP	SmallInt	↑ key

Table G-30 LibraryScope

Constant	Data type	Description
GlobalToDesktop	SmallInt	Makes variables in an ObjectPAL library available to one or more forms.
PrivateToForm	SmallInt	Makes variables in an ObjectPAL library available to one form only.

Table G-31 LineEnds

Constant	Data type	Description
ArrowBothEnds	SmallInt	Adds arrows to both ends of a line (only if LineType = StraightLine)
ArrowOneEnd	SmallInt	Adds an arrow to the terminal end of a line (only if LineType = StraightLine)
NoArrowEnd	SmallInt	Displays a line without arrows at either end.

Table G-32 LineStyles

Constant	Data type	Description
DashDotDotLine	SmallInt	A repeating sequence of one dash followed by two dots.
DashDotLine	SmallInt	A repeating sequence of one dash followed by one dot.
DashedLine	SmallInt	A repeating sequence of dashes.
DottedLine	SmallInt	A repeating sequence of dots.
NoLine	SmallInt	No line.
SolidLine	SmallInt	An unbroken line.

Table G-33 LineThickness

Constant	Data type	Description
LWidth10Points	SmallInt	Specifies a thickness of 10 printer's points.
LWidth1Point	SmallInt	Specifies a thickness of 1 printer's point.
LWidth2Points	SmallInt	Specifies a thickness of 2 printer's points.
LWidth3Points	SmallInt	Specifies a thickness of 3 printer's points.
LWidth6Points	SmallInt	Specifies a thickness of 6 printer's points.
LWidthHairline	SmallInt	Specifies a very thin line.
LWidthHalfFoint	SmallInt	Specifies a thickness of one half of a printer's point.

Table G-34 LineTypes

Constant	Data type	Description
CurvedLine	SmallInt	Specifies a curved (elliptical) line.
StraightLine	SmallInt	Specifies a straight line.

Table G-35 MenuChoiceAttributes

Constant	Data type	Description
MenuChecked	SmallInt	Insert a checkmark before the menu item.
MenuDisabled	SmallInt	Menu item cannot be selected. Menu stays open.
MenuEnabled	SmallInt	Menu item can be selected. Menu closes.
MenuGrayed	SmallInt	Menu item displayed in gray characters (dimmed).
MenuHilited	SmallInt	Menu item is highlighted.
MenuNotChecked	SmallInt	Display menu item without a checkmark.
MenuNotGrayed	SmallInt	Display the menu item normally (not dimmed).
MenuNotHilited	SmallInt	Display menu item without a highlight.

Table G-36 MenuCommands

Constant	Data type	Description
MenuCanClose	SmallInt	Asks for permission to continue after choosing Close from the Control menu.
MenuControlClose	SmallInt	Same as choosing Close from the Control menu.
MenuControlKeyMenu	SmallInt	Control menu was invoked by a keypress.
MenuControlMaximize	SmallInt	Same as choosing Maximize from the Control menu.
MenuControlMinimize	SmallInt	Same as choosing Minimize from the Control menu.
MenuControlMouseMenu	SmallInt	Control menu was invoked by a mouse click.
MenuControlMove	SmallInt	Same as choosing Move from the Control menu.
MenuControlNextWindow	SmallInt	Same as choosing Next Window from the Control menu.
MenuControlPrevWindow	SmallInt	Same as choosing Prev Window from the Control menu.
MenuControlRestore	SmallInt	Same as choosing Restore from the Control menu.
MenuControlSize	SmallInt	Same as choosing Size from the Control menu.
MenuEditCopy	SmallInt	Edit|Copy
MenuEditCopyTo	SmallInt	Edit|Copy To
MenuEditCut	SmallInt	Edit|Cut
MenuEditDelete	SmallInt	Edit|Delete
MenuEditPaste	SmallInt	Edit|Paste
MenuEditPasteFrom	SmallInt	Edit|Paste From
MenuEditSearchText	SmallInt	Edit|Search Text
MenuEditUndo	SmallInt	Edit|Undo

<table>
<tr><th>Constant</th><th>Data type</th><th>Description</th></tr>
<tr><td>MenuFileAliases</td><td>SmallInt</td><td>File|Aliases</td></tr>
<tr><td>MenuFileExit</td><td>SmallInt</td><td>File|Exit</td></tr>
<tr><td>MenuFileExport</td><td>SmallInt</td><td>File|Utiltities|Export</td></tr>
<tr><td>MenuFileImport</td><td>SmallInt</td><td>File|Utilitites|Import</td></tr>
<tr><td>MenuFileMultiBlankZero</td><td>SmallInt</td><td>File|System Settings|Blank As Zero</td></tr>
<tr><td>MenuFileMultiUserAutoRefresh</td><td>SmallInt</td><td>File|System Settings|Auto Refresh</td></tr>
<tr><td>MenuFileMultiUserDrivers</td><td>SmallInt</td><td>File|System Settings|Drivers</td></tr>
<tr><td>MenuFileMultiUserLock</td><td>SmallInt</td><td>File|Multiuser|Set Locks</td></tr>
<tr><td>MenuFileMultiUserLockInfo</td><td>SmallInt</td><td>File|Multiuser|Display Locks</td></tr>
<tr><td>MenuFileMultiUserRetry</td><td>SmallInt</td><td>File|Multiuser|Set Retry</td></tr>
<tr><td>MenuFileMultiUserUserName</td><td>SmallInt</td><td>File|Multiuser|User Name</td></tr>
<tr><td>MenuFileMultiUserWho</td><td>SmallInt</td><td>File|Multiuser|Who</td></tr>
<tr><td>MenuFilePrint</td><td>SmallInt</td><td>File|Print</td></tr>
<tr><td>MenuFilePrinterSetup</td><td>SmallInt</td><td>File|Printer Setup</td></tr>
<tr><td>MenuFilePrivateDir</td><td>SmallInt</td><td>File|Private Directory</td></tr>
<tr><td>MenuFileTableAdd</td><td>SmallInt</td><td>File|Utilties|Add</td></tr>
<tr><td>MenuFileTableCopy</td><td>SmallInt</td><td>File|Utilties|Copy</td></tr>
<tr><td>MenuFileTableDelete</td><td>SmallInt</td><td>File|Utilties|Delete</td></tr>
<tr><td>MenuFileTableEmpty</td><td>SmallInt</td><td>File|Utilties|Empty</td></tr>
<tr><td>MenuFileTableInfoStructure</td><td>SmallInt</td><td>File|Utilties|Info Structure</td></tr>
<tr><td>MenuFileTablePasswords</td><td>SmallInt</td><td>File|Utilties|Passwords</td></tr>
<tr><td>MenuFileTableRename</td><td>SmallInt</td><td>File|Utilties|Rename</td></tr>
<tr><td>MenuFileTableRestructure</td><td>SmallInt</td><td>File|Utilties|Restructure</td></tr>
<tr><td>MenuFileTableSort</td><td>SmallInt</td><td>File|Utilties|Sort</td></tr>
<tr><td>MenuFileTableSubtract</td><td>SmallInt</td><td>File|Utilties|Subtract</td></tr>
<tr><td>MenuFileWorkingDir</td><td>SmallInt</td><td>File|Working Directory</td></tr>
<tr><td>MenuFolderOpen</td><td>SmallInt</td><td>File|Open|Folder</td></tr>
<tr><td>MenuFormDesign</td><td>SmallInt</td><td>Form|Design</td></tr>
<tr><td>MenuFormEditData</td><td>SmallInt</td><td>Form|Edit Data</td></tr>
<tr><td>MenuFormFieldView</td><td>SmallInt</td><td>Form|Field View</td></tr>
<tr><td>MenuFormNew</td><td>SmallInt</td><td>File|New|Form</td></tr>
<tr><td>MenuFormOpen</td><td>SmallInt</td><td>File|Open|Form</td></tr>
<tr><td>MenuFormOrderRange</td><td>SmallInt</td><td>Form|Order/Range</td></tr>
<tr><td>MenuFormPageFirst</td><td>SmallInt</td><td>FormPage|First</td></tr>
<tr><td>MenuFormPageGoto</td><td>SmallInt</td><td>Form|Page|Go To</td></tr>
<tr><td>MenuFormPageLast</td><td>SmallInt</td><td>Form|Page|Last</td></tr>
<tr><td>MenuFormPageNext</td><td>SmallInt</td><td>Form|Page|Next</td></tr>
<tr><td>MenuFormPagePrevious</td><td>SmallInt</td><td>Form|Page|Previous</td></tr>
<tr><td>MenuFormShowDeleted</td><td>SmallInt</td><td>Form|Show Deleted</td></tr>
</table>

<table>
<tr><th>Constant</th><th>Data type</th><th>Description</th></tr>
<tr><td>MenuFormTableView</td><td>SmallInt</td><td>Form|Table View</td></tr>
<tr><td>MenuHelpAbout</td><td>SmallInt</td><td>Help|About</td></tr>
<tr><td>MenuHelpContents</td><td>SmallInt</td><td>Help|Contents</td></tr>
<tr><td>MenuHelpKeyboard</td><td>SmallInt</td><td>Help|Keyboard</td></tr>
<tr><td>MenuHelpSpeedBar</td><td>SmallInt</td><td>Help|SpeedBar</td></tr>
<tr><td>MenuHelpSupport</td><td>SmallInt</td><td>Help|Support Info</td></tr>
<tr><td>MenuHelpUsingHelp</td><td>SmallInt</td><td>Help|Using Help</td></tr>
<tr><td>MenuInit</td><td>SmallInt</td><td>Generated by clicking a menu item</td></tr>
<tr><td>MenuLibraryNew</td><td>SmallInt</td><td>File|New|Library</td></tr>
<tr><td>MenuLibraryOpen</td><td>SmallInt</td><td>File|Open|Library</td></tr>
<tr><td>MenuPasteLink</td><td>SmallInt</td><td>Edit|Paste Link</td></tr>
<tr><td>MenuPropertiesCurrent</td><td>SmallInt</td><td>Properties|Current Object</td></tr>
<tr><td>MenuPropertiesDesigner</td><td>SmallInt</td><td>Properties|Designer</td></tr>
<tr><td>MenuPropertiesDesktop</td><td>SmallInt</td><td>Properties|Desktop</td></tr>
<tr><td>MenuPropertiesExpandedRuler</td><td>SmallInt</td><td>Properties|Expanded Ruler</td></tr>
<tr><td>MenuPropertiesFormRestoreDefaults</td><td>SmallInt</td><td>Properties|Form Options|Restore Defaults</td></tr>
<tr><td>MenuPropertiesFormSaveDefaults</td><td>SmallInt</td><td>Properties|Form Options|Save Defaults</td></tr>
<tr><td>MenuPropertiesHorizontalRuler</td><td>SmallInt</td><td>Properties|Form Options|Horizontal Ruler</td></tr>
<tr><td>MenuPropertiesVerticalRuler</td><td>SmallInt</td><td>Properties|Form Options|Vertical Ruler</td></tr>
<tr><td>MenuPropertiesZoom100</td><td>SmallInt</td><td>Properties|Zoom|100%</td></tr>
<tr><td>MenuPropertiesZoom200</td><td>SmallInt</td><td>Properties|Zoom|200%</td></tr>
<tr><td>MenuPropertiesZoom25</td><td>SmallInt</td><td>Properties|Zoom|25%</td></tr>
<tr><td>MenuPropertiesZoom400</td><td>SmallInt</td><td>Properties|Zoom|400%</td></tr>
<tr><td>MenuPropertiesZoom50</td><td>SmallInt</td><td>Properties|Zoom|50%</td></tr>
<tr><td>MenuPropertiesZoomBestFit</td><td>SmallInt</td><td>Properties|Zoom|Best Fit</td></tr>
<tr><td>MenuPropertiesZoomFitHeight</td><td>SmallInt</td><td>Properties|Zoom|Fit Height</td></tr>
<tr><td>MenuPropertiesZoomFitWidth</td><td>SmallInt</td><td>Properties|Zoom|Fit Width</td></tr>
<tr><td>MenuQueryNew</td><td>SmallInt</td><td>File|New|Query</td></tr>
<tr><td>MenuQueryOpen</td><td>SmallInt</td><td>File|Open|Query</td></tr>
<tr><td>MenuRecordCancel</td><td>SmallInt</td><td>Record|Cancel Changes</td></tr>
<tr><td>MenuRecordDelete</td><td>SmallInt</td><td>Record|Delete</td></tr>
<tr><td>MenuRecordFastBackward</td><td>SmallInt</td><td>Record|Previous Set</td></tr>
<tr><td>MenuRecordFastForward</td><td>SmallInt</td><td>Record|Next Set</td></tr>
<tr><td>MenuRecordFirst</td><td>SmallInt</td><td>Record|First</td></tr>
<tr><td>MenuRecordInsert</td><td>SmallInt</td><td>Record|Insert</td></tr>
<tr><td>MenuRecordLast</td><td>SmallInt</td><td>Record|Last</td></tr>
<tr><td>MenuRecordLocateNext</td><td>SmallInt</td><td>Record|Locate Next</td></tr>
<tr><td>MenuRecordLocateRecordNumber</td><td>SmallInt</td><td>Record|Locate|Record Number</td></tr>
<tr><td>MenuRecordLocateSearchAndReplace</td><td>SmallInt</td><td>Record|Locate|and Replace</td></tr>
</table>

Constant	Data type	Description
MenuRecordLocateValue	SmallInt	Record|Locate|Value
MenuRecordLock	SmallInt	Record|Lock
MenuRecordLookup	SmallInt	Record|Lookup Help
MenuRecordMove	SmallInt	Record|Move Help
MenuRecordNext	SmallInt	Record|Next
MenuRecordPost	SmallInt	Record|Post/Keep Locked
MenuRecordPrevious	SmallInt	Record|Previous
MenuReportNew	SmallInt	File|New|Report
MenuReportOpen	SmallInt	File|Open|Report
MenuSave	SmallInt	File|Save
MenuScriptNew	SmallInt	File|New|Script
MenuScriptOpen	SmallInt	File|Open|Script
MenuSelectAll	SmallInt	Edit|Select All
MenuTableNew	SmallInt	File|New|Table
MenuTableOpen	SmallInt	File|Open|Table
MenuWindowArrangeIcons	SmallInt	Window|Arrange Icons
MenuWindowCascade	SmallInt	Window|Cascade
MenuWindowCloseAll	SmallInt	Window|Close All
MenuWindowTile	SmallInt	Window|Tile

Table G-37 MenuReasons

Constant	Data type	Description
MenuControl	SmallInt	Triggered by choosing an item from the control menu.
MenuDesktop	SmallInt	Triggered by choosing an item from a built-in Paradox menu.
MenuNormal	SmallInt	Triggered by choosing an item from a custom ObjectPAL menu or by clicking a SpeedBar button.

Table G-38 MouseShapes

Constant	Data type	Description
MouseArrow	LongInt	Standard pointer arrow.
MouseCross	LongInt	Pointer is a cross.
MouseIBeam	LongInt	Pointer is an I-beam (text insertion cursor).
MouseUpArrow	LongInt	Pointer is an arrow pointing up.
MouseWait	LongInt	Pointer is an hourglass.

Table G-39 MoveReasons

Constant	Data type	Description
PalMove	SmallInt	Caused by an ObjectPAL statement.
RefreshMove	SmallInt	Caused when data is updated, for example, by scrolling through a table.
ShutDownMove	SmallInt	Caused when the form closes.
StartupMove	SmallInt	Caused when the form opens.
UserMove	SmallInt	Caused by the user.

Table G-40 PatternStyles

Constant	Data type
BricksPattern	SmallInt
CrosshatchPattern	SmallInt
DiagonalCrosshatchPattern	SmallInt
DottedLinePattern	SmallInt
EmptyPattern	SmallInt
FuzzyStripesDownPattern	SmallInt
HeavyDotPattern	SmallInt
HorizontalLinesPattern	SmallInt
LatticePattern	SmallInt
LeftDiagonalLinesPattern	SmallInt
LightDotPattern	SmallInt
MaximumDotPattern	SmallInt
MediumDotPattern	SmallInt
RightDiagonalLinePattern	SmallInt
ScalesPattern	SmallInt
StaggeredDashPattern	SmallInt
ThickHorizontalLinePattern	SmallInt
ThickStripesDownPattern	SmallInt
ThickStripesUpPattern	SmallInt
ThickVerticalLinesPattern	SmallInt
VerticalLinesPattern	SmallInt
VeryHeavyDotPattern	SmallInt
WeavePattern	SmallInt
ZigZagPattern	SmallInt

Table G-41 RasterOperations

Constant	Data type	Description
MergePaint	LongInt	Inverts the source graphic and combines it with the destination using the Boolean OR operator.
NotSourceCopy	LongInt	Inverts the source graphic and copies it to the destination.
NotSourceErase	LongInt	Combines the source graphic and the destination and inverts the result using the Boolean OR operator.
SourceAnd	LongInt	Combines the source graphic and the destination using the Boolean AND operator.
SourceCopy	LongInt	Copies an unchaged source graphic to the destination.
SourceErase	LongInt	Inverts the destination and combines it with the source graphic using the Boolean AND operator.
SourceInvert	LongInt	Combines the source graphic and the destination using the Boolean XOR operator.
SourcePaint	LongInt	Combines the source graphic and the destination using the Boolean OR operator.

Table G-42 ReportPrintRestart

Constant	Data type	Description
PrintFromCopy	SmallInt	Prints the report from copies of the tables in the report's data model.
PrintLock	SmallInt	Locks tables in the report's data model before printing.
PrintNoLock	SmallInt	Prints without locking tables in the report's table model.
PrintRestart	SmallInt	Restarts print job when data changes in tables in the report's data model.
PrintReturn	SmallInt	Cancel the print job when data changes in tables in the report's data model.

Table G-43 StatusReasons

Constant	Data type	Description
ModeWindow1	SmallInt	The status bar area second from the left.
ModeWindow2	SmallInt	The status bar area third from the left.
ModeWindow3	SmallInt	The rightmost status bar area.
StatusWindow	SmallInt	The leftmost (and largest) status bar area.

Table G-44 TableFrameStyles

Constant	Data type	Description
tf3D	SmallInt	Table frame has a 3D frame.
tfDoubleLine	SmallInt	Table frame has a double-box frame.
tfNoGrid	SmallInt	Table frame has no grid.

Constant	Data type	Description
tfSingleLine	SmallInt	Table frame has a box frame.
tfTripleLine	SmallInt	Table frame has a triple-box frame.

Table G-45 TextAlignment

Constant	Data type	Description
TextAlignBotton	SamllInt	Bottom of text is aligned (table window only)
TextAlignCenter	SmallInt	Text is centered horizontally.
TextAlignJustify	SmallInt	Text is justified right and left (does not apply to table window).
TextAlignLeft	SmallInt	Text is left-justified.
TextAlignRight	SmallInt	Text is right-justified.
TextAlignTop	SmallInt	Top of text is aligned (table window only)
TextAlignVCenter	SmallInt	Text is centered vertically (table window only).

Table G-46 TextDesignSizing

Constant	Data type	Description
TextFixedSized	SmallInt	Text box does not change size.
TextGrowOnly	SmallInt	Text box grows to accommodate text.
TextSizeToFit	SmallInt	Text box grows or shrinks as necessary to accommodate text.

Table G-47 TextSpacing

Constant	Data type	Description
TextDoubleSpacing	SmallInt	2 lines.
TextDoubleSpacing2	SmallInt	2.5 lines.
TextSingleSpacing	SmallInt	1 line.
TextSingleSpacing2	SmallInt	1.5 lines.
TextTripleSpacing	SmallInt	3 lines.

Table G-48 UIObjectTypes

Constant	Data type	Description
BoxTool	SmallInt	Creates a box.
ButtonTool	SmallInt	Creates a button.
ChartTool	SmallInt	Creates a graph.
EllipseTool	SmallInt	Creates an ellipse.
FieldTool	SmallInt	Creates a field.
GraphicTool	SmallInt	Creates a graphic object.
LineTool	SmallInt	Creates a line.

Constant	Data type	Description
OleTool	SmallInt	Creates an OLE object.
RecordTool	SmallInt	Creates a record.
TableFrameTool	SmallInt	Creates a table frame.
TextTool	SmallInt	Creates a text box.
XtabTool	SmallInt	Creates a crosstab object.

Table G-49 ValueReasons

Constant	Data type	Description
EditValue	SmallInt	The built-in **newValue** method of a radio button field, list, or drop-down edit list has been triggered (for example, by poking a radio button or choosing a list item), but the field value has not been committed (for example, by moving off the field).
FieldValue	SmallInt	A field's built-in **newValue** method has been triggered, and the value has been committed.
StartupValue	SmallInt	A field's built-in **newValue** method has been triggered because the form has opened.

Table G-50 WindowStyles

Constant	Data type	Description
WinDefaultCoordinate	LongInt	Displays a window at its default size and position.
WinStyleBorder	LongInt	Specifies a sizing border.
WinStyleControlMenu	LongInt	Specifies a system-control menu.
WinStyleDefault	LongInt	Specifies default displays attributes.
WinStyleDialog	LongInt	Specifies dialog box attributes.
WinStyleDialogFrame	LongInt	Specifies a dialog box frame.
WinStyleHScroll	LongInt	Specifies a horizontal scroll bar.
WinStyleHidden	LongInt	Makes a window invisible.
WinStyleMaximize	LongInt	Displays a window at full size.
WinStyleMaximizeButton	LongInt	Specifies a miximize button.
WinStyleMinimize	LongInt	Displays a window as an icon (minimized).
WinStyleMinimizeButton	LongInt	Specifies a minimize button.
WinStyleModal	LongInt	Makes a window modal.
WinStyleThickFrame	LongInt	Specifies a thick frame.
WinStyleTitleBar	LongInt	Specifies a title bar.
WinStyleVScroll	LongInt	Specifies a vertical scroll bar.

INDEX

" (quotes) for empty strings 67, 403
$ (dollar sign) end of line operator 405, 652
& (ampersand), for accelerator keys 256, 276, 343
( ) (parentheses) grouping symbol 405, 652
* (asterisk) wildcard operator 143, 148, 405, 652
+ (plus sign) search operator 405, 652
, (comma) in format specifications 412
.. (periods) as match operator 422
: (colon)
  in drive names 150, 157, 166
  in scan statements 40
< > (comparison operator), comparing records with 355
= (assignment operator) 27, 28
  comparing records with 355
  copying records with 355
  create statements and 503
  memo variables and 249
  in queries 97, 348
= (equal sign) 479, 490
? (question mark) wildcard character 143, 148
@ (at sign) match operator 405, 422, 652
[ ] (brackets)
  advMatch method and 405, 652
  readProfileString method and 479
  writeProfileString method and 490
\ (backslash)
  path names and 458
  queries and 97, 100, 348
  string searches and 405, 423, 652
\t (tab character)
  accelerator keys and 343
  removing from strings 422
^ (caret) beginning of line operator 405, 652
| (vertical bar) OR operator 405, 652
~ (tilde variables) in queries 97, 100, 349, 352

A

Abort message 473
abs method 301
absolute value 301
accelerator keys 256, 276, 343
access rights
  See also networks; passwords
  fields 595
  files 141, 154, 162
  objects 520, 594
  read-only tables 541
  reporting 595
  tables 368, 519, 529, 537, 545, 590, 610, 646
accessRights method 141
  getFileAccessRights procedure versus 154
acos method 302
action constants 60, 170, 551, 670
action method/procedure
  built-in 12, 60
  Form type 170
  TableView type 551
  UIObject type 670
actionClass method 60
ActionEvent type
  defined 60
  methods 60–63
ActionEvents
  class number 60
  ID number 61
  specifying 62
actions
  See also events
  delaying execution 213, 732
  executing 170, 551
  responding to 170
Active variable 21
add method/procedure
  Table type 491
  TCursor type 555
addAlias method/procedure 367
addArray method
  Menu type 253
  PopUpMenu type 336
addBar method 337
addBreak method
  Menu type 253
  PopUpMenu type 338
addLast method 72
addPassword method/procedure 368

addPopUp method
  Menu type 254
  PopUpMenu type 339
addSeparator method 341
addStaticText method
  Menu type 255
  PopUpMenu type 342
addText method
  Menu type 256
  PopUpMenu type 342
advancedWildcardsInLocate procedure 369
advMatch method
  ignoreCaseInStringCompares method and 652
  String type 404
  TextStream type 651
aliases
  database 367, 380
  listing 371
  queries and 97
  removing 386
  saving 388
  setDrive method and 162
  setting path 389
alignment 411
alphanumeric strings
  See strings
Alt key
  isAltKeyDown method and 225
  setAltKeyDown method and 227
amortization 313
ampersand (&), for accelerator keys 256, 276, 343
AND operations
  bitwise 243, 392
  logical 241
annuity calculations 307, 315
ANSI characters
  See also characters; text
  converting code to string 409
  converting to OEM 428
  methods for 651–664
  returning code for 225, 406
  sending 228
  table of codes for 781
ANSI code
  converting to virtual keycode 430
  converting virtual key code to 420
ansiCode procedure 406
AnyType type
  C procedures and 51
  defined 64
  methods 64–70
append method 74
Application type
  Form type and 71
applications
  debugging 435
  Desktop windows and 71
  exiting 458
  listing open 447, 452
  multiuser *See* networks
  user interface for 669
arc cosine 302
arc sine 302
arc tangent 303
Array type 64, 69
  defined 72
  methods 72–88, 197
arrays 72
  See also dynamic arrays
  adding to pop-up menus 336
  appending items to menu 253
  combining 74
  copying to/from records 569, 571, 676, 677
  counting values in 75
  decreasing size of 78
  displaying contents of 87
  dynamic *See* dynamic arrays
  field names in 685
  filling 77
  increasing size of 78
  incrementing items 72
  inserting items into 72, 80–82
  object names in 686
  overwriting items in 85
  position of items in 79
  removing items from 76, 83–85
  reporting as resizeable 82
  reporting size of 87
  searching in 74
  size of 86, 87
  swapping cell contents 77
  writing file information to 146
arrive method 8
arrow pointer 9
ASCII codes, converting to characters 409
asin method 302
assembly language routines, calling from ObjectPAL 46
assignment operator (=) 27, 28
  comparing records with 355
  copying records with 355
  create statements and 503
  in queries 97, 348

asterisk (*) wildcard character 143, 148, 405, 652
at sign (@) match operator 405, 422, 652
atan method 303
atan2 method 303
atFirst method
  TCursor type 556
  UIObject type 671
atLast method
  TCursor type 557
  UIObject type 672
attach method
  Form type 171
  isTable method and 493
  opening tables and 629
  Report type 357
  Table type 493
  TCursor type 557
  UIObject type 672
  UIObjects and 28
  unAttach method and 493
attachToKeyViol method 559
attributes
  See properties
average of fields in a table 494, 562

B

backslash (\)
  path names and 458
  queries and 97, 100, 348
  string searches and 405, 423, 652
bar symbol (|) OR operator 405, 652
bars, adding to menu 341
batch files (DOS) 478, 490
beep procedure 431
beginning of a file 657
Binary type 64, 69
  defined 89
  methods 89–91
  returning size of variables 90
bitAND method
  LongInt type 243
  SmallInt type 392
bitIsSet method
  LongInt type 244
  SmallInt type 393
bitmaps
  See also Graphic type
  reading from Clipboard 220
  writing to Clipboard 222
  writing to files 223
bitOR method
  LongInt type 246
  SmallInt type 394
bitwise operations
  LongInt 243, 246
  SmallInt 392, 394, 395
bitXOR method
  LongInt type 246
  SmallInt type 395
blank method/procedure 65
blank values 370, 383
  filling record buffer with 603
  removing from strings 422
  reporting 67
  returning 65
blankAsZero method/procedure 65, 370, 494, 565
Boolean
  See Logical type
bot method 560
bounding box object 693
brackets ([ ])
  advMatch method and 405, 652
  readProfileString method and 479
  writeProfileString method and 490
branching, iif statements and 34
breakApart method 407
breakpoints 435
  See also Debugger
bringToTop method/procedure 172
broadCastAction method 674
bubble sort 42
bubbling
  See also containership hierarchy
  external events and 10
  internal events and 7
  Self variable and 21
built-in methods 5–25, 669
  See also methods; names of specific methods
  attaching code to 6
  default behavior 16
  defined 6
  deleting 197
  disabling 17, 29
  enabling 18, 31
  error 12
  event handling and 135, 137
  executing immediately 16, 30
  execution sequence 9, 14
  external events 10–13
  internal events 7–9
  isTargetSelf method and 137

library 235
mouse 9–11
status 12
suspending execution 19
built-in variables 20
buttons
See also UIObjects
appearance of 13
methods for 13
setFocus method and 8

C

C programming language
AnyType type and 51
.DEF file 53
passing binary data 53
passing by pointer 52
passing floating-point numbers 52
passing graphic data 53
routines, calling from ObjectPAL 46, 50
window handles and 218
C++ routines, calling from ObjectPAL 46, 53
canArrive method 7
Cancel buttons in dialog boxes 475
cancelEdit method 561, 674
canDepart method 8
canReadFromClipboard method 323
caret (^) beginning of line operator 405, 652
carriage returns, removing from strings 424
case sensitivity
comparisons and 418
converting to lowercase 421
converting to uppercase 429
passwords 387, 534
searches and 381, 384, 404, 652
specifications 411
cash flow, managing 315
casting values as strings 426
cAverage method/procedure
blankAsZero method and 370
Table type 494
TCursor type 562
cCount method/procedure
blankAsZero method and 370
Table type 495
TCursor type 562
cDouble keyword 49
ceil method 304
changeValue method 13, 749
built-in 8
newValue method and 15
char method 224
characters
See also ANSI characters; strings; TextStream type
ANSI 651
ANSI values of 225
methods for 651–664
nonprinting 651
number of in text files 663
reading from files 660, 661
searching for 615–620, 651, 709–713
virtual 231, 232, 233
charAnsiCode method 225
check boxes 13
chr procedure 409
chrOEM procedure 409
chrToKeyName procedure 410
classes
See data types
Clipboard
OLE objects and 323
reading from 220
writing graphics to 222
cLong keyword 49
cLongDouble keyword 49
close method/procedure
built-in 7
Database type 95
DDE type 120
Form type 173
Library type 235
Report type 358
Session type 371
System type 432
TableView type 551
TCursor type 563
TextStream type 653
cMax method/procedure
Table type 496
TCursor type 564
cMin method/procedure
Table type 497
TCursor type 565
cNpv method/procedure
Table type 498
TCursor type 566
code
See methods
code modules
See methods
colon (:)
in drive names 150, 157, 166
in scan statements 40

colors
  constants 737
  defining 737
comma (,) in format specifications 412
commit method 654
compact method
  Table type 499
  TCursor type 567
comparison operator (< >), comparing records with 355
compound value 307
COMSPEC variable, DOS 478, 490
constantNameToValue procedure 433
constants
  action 60, 170, 551, 670
  built-in 433
  colors 737
  declaring 29, 36
  error 454
  keyboard events 231
  keyboard state 192, 200
  library scope 239
  listing 450
  logical 241
  menu choices 274
  menu display attributes 256
  MenuCommand 197
  MenuEvent 277, 279
  MouseShapes 480
  MoveEvent 297, 398, 400
  reason method and 138, 139
  returning name of 434
  returning value of 433
  setErrorCode method and 139
  status window 398
  storing 235
  table of 823–848
  UIObject 678
  user-defined 170
  virtual characters 233
  window styles 211, 363
constantValueToName procedure 434
Container variable 21
containers, reporting on valid 701
containership hierarchy
  See also bubbling
  external events and 10
  internal events and 7
  passEvent and 19
  path 28
contains method
  Array type 74
  DynArray type 124
  Menu type 259
control structures 27
converting
  ANSI code to string 409
  ANSI code to virtual keycode 430
  to lowercase 421
  string to ANSI code 406
  string to OEM code 423
  string to virtual key code 410
  to uppercase 429
  value to a string 427
  virtual key code to ANSI code 420
  virtual key code to character 420
convertPointWithRespectTo method 675
coordinates, screen 477
copy method/procedure
  FileSystem type 142
  Table type 500
  TCursor type 568
  wildcards and 142
copyFromArray method
  TCursor type 569
  UIObject type 676
copyRecord method 570
copyToArray method
  TCursor type 571
  UIObject type 677
cos method 304
cosh method 305
cosine 304
  arc 302
  hyperbolic 305
count method 260
countOf method 75
cptr keyword 49
cpuClockTime procedure 434
create keyword 501
  isAssigned method and 527
create method
  Form type 173
  methodSet method and 174
  TextStream type 654
  UIObject type 678
critical errors 454, 455, 457
cSamStd method/procedure
  Table type 505
  TCursor type 572

cSamVar method/procedure
  Table type 506
  TCursor type 573
cStd method/procedure
  Table type 507
  TCursor type 574
cSum method/procedure
  Table type 508
  TCursor type 575
Ctrl key
  isControlKeyDown method and 226
  MouseEvents and 283, 288
  setControlKeyDown method and 229
Ctrl+Break key 177
currency method 92
Currency type 64, 69
  defined 92
  formats for 465
  methods 92–93
currentPage method 359
currRecord method
  TCursor type 576
  UIObject type 679
cursor
  See insertion point
custom methods
  See also methods; built-in methods; procedures
  calling 238
  listing 236
  storing 235
  Subject variable and 21
custom methods and procedures 692
cVar method/procedure
  Table type 509
  TCursor type 576
cWord keyword 49

D

data
  See fields
data method 273
data model
  See also forms; tables
  retrieving values from table in 178
  tables and 177–181
data types
  C routines and 51
  declaring 36, 44
  fields 523, 597
  ObjectPAL versus C or Pascal 49
  record 45
  returning 65
  storing 235
  user-defined 235
  uses statement and 49
  variables and 68
database engine
  accessing 385
  methods 367–390
  properties 377
Database type
  defined 94
  methods 95–105
  returning 40
database variables 94, 502
  assigning values to 101
databases
  See also tables
  aliases 367
  closing 95
  deleting tables from 96
  handles to 94
  listing open 379
  listing tables in 372
  opening 103
  queries and 348
  reporting tables in 530
  testing for tables in 102
dataType method 65
DataUnlockRecord constant 8
date procedure 107
Date type 64, 69
  See also DateTime type; Time type
  defined 106
  formats for 465
  methods 106–108
dates 106
  converting values to 107
  declaring values as 107
  formatting 106, 411, 465
  returning current 108
dateTime method 110
DateTime type 64, 69
  See also Date type; Time type
  declaring value as 110
  defined 109
  formats for 466
  methods 109–119
  separators for 109
dateVal procedure 107
day method 110
daysInMonth method 111

dBASE tables
  See also tables
  compacting indexes 499
  deleting indexes 517, 579
  deleting records 512, 567
  displaying deleted records 541, 611, 640
  empty 528
  field types 523, 597
  index switching 644, 743
  indexes 511, 549, 685
  listing index fields 584, 627
  listing index structure 586
  listing indexes 532
  locks, listing 588
  numeric fields in 597, 599
  rebuilding indexes 535, 634
  record counting 533, 628, 731
  record position in 633, 636
  record searches 624, 728
  removing deleted records from 499, 567
  reporting deleted records 610
  resynchronizing TCursor 736
  searching in 624, 728
  secondary indexes 525, 586
  testing for 545, 647
  undeleting records 647, 745
  writing index fields to array 514
DDE type
  closing links 120
  defined 120
  methods 120–123
  OLE and 322
  opening links 121
  returning 40
  sending commands 121
  specifying new item 122
debug procedure 435
Debugger 435
  Tracer window 484–487
delayScreenUpdates procedure 174
delete method/procedure
  Database type 96
  FileSystem type 143
  Table type 510
  UIObject type 679
deleteDir method 144
  enumFileList method and 145
deleteRecord method
  TCursor type 577
  UIObject type 680
deliver method 176
depart method 8
design method
  Form type 176
  Report type 360
  run method and 176, 360
design objects, timer events and 667
Design window
  creating forms in 173
  opening forms in 195, 211
  opening reports in 362
  saving forms and 215
Desktop window 71
  listing names 447
deviation, standard 505, 507, 572, 574
dialog boxes
  displaying 473–477
  invoking 436–446, 456
  messages in 474, 476
  opening 212
  position of 216
didFlyAway method 578
directories
  See also databases
  creating 158
  deleting 144
  private 160
  reporting name of 155
  reporting on start-up 165
  reporting on working 168
  reporting path of 152
  setting path 161
disableBreakMessage procedure 177
disableDefault statements 17, 29
  passEvent and 37
disk files
  See files; FileSystem type
disk space, reporting on 150
display formats
  See formats
distance method 329
dlgAdd procedure 436
dlgCopy procedure 436
dlgCreate procedure 437
dlgDelete procedure 438
dlgEmpty procedure 438
dlgNetDrivers procedure 439
dlgNetLocks procedure 439
dlgNetRefresh procedure 440
dlgNetRetry procedure 441
dlgNetSetLocks procedure 441
dlgNetSystem procedure 442
dlgNetUserName procedure 442
dlgNetWho procedure 443

dlgRename procedure 444
dlgRestructure procedure 444
dlgSort procedure 445
dlgSubtract procedure 445
dlgTableInfo procedure 446
DLL
  calling routines from 47
  defined 46
  specifying module name 48
  window handles and 218
dmAddTable method/procedure 177
dmGet method/procedure 178
dmHasTable method/procedure 179
dmPut method/procedure 180
dmRemoveTable method/procedure 180
doDefault statements 16, 30
  enableDefault keyword versus 18
dollar sign ($) end of line operator 405, 652
DOS
  environment 478, 490
  executing commands 458
  wildcard characters in 143, 148
dot notation
  create statements and 503
  field values and 600, 637
  index keyword and 526
dow method 111
dowOrd method 112
doy method 113
drivers
  choosing 439
  reporting on 373–377
drives
  fixed 156
  free disk space on 150
  remote 157
  removable 158
  reporting capacity of 166
  reporting on 145, 147, 153, 156, 157, 158
  setting default 161
drives method 145
drop-down lists 8, 14, 149, 254, 826
dropIndex method
  Table type 511
  TCursor type 579
dynamic arrays
  See also arrays
  forEach statement and 32
  indexes 33, 124
  removing items from 126, 128
  searching indexes 124
  size of 129
  sysInfo method and 483
dynamic data exchange
  See DDE
DynArray type 64, 69
  See also dynamic arrays
  described 124
  displaying in dialog box 129
  methods 124–130
DynArrays
  See dynamic arrays

E

edit method
  OLE type 323
  TCursor type 580
  UIObject type 681
Edit mode 681
  deleting records and 577
  exiting 561, 582, 674, 684
  inserting records in 603, 699, 700
  TCursor in 580
edit specifications 412
empty method/procedure
  Array type 76
  delete method/procedure versus 510, 512
  DynArray type 126
  Menu type 261
  Table type 512
  TCursor type 581
  UIObject type 682
empty strings 67, 419
enableDefault statements 18, 31
encrypted tables
  See tables, encrypted
end method
  TCursor type 582
  TextStream type 656
  UIObject type 684
end of file 657
end of table 593
endEdit method
  TCursor type 582
  UIObject type 684
endMethod keyword 36
endProc keyword 38
endQuery keyword 97
endSwitch keyword 42
endTry keyword 43
endVar keyword 54
endWhile keyword 54

enumAliasNames method/procedure 371
enumDataBaseTables method/procedure 372
enumDesktopWindowNames procedure 447
enumDriverCapabilities procedure 373
enumDriverInfo procedure 375
enumDriverNames method/procedure 376
enumDriverTopics procedure 377
enumEngineInfo procedure 377
enumFieldNames method
  Table type 513
  TCursor type 583
  UIObject type 685
enumFieldNamesInIndex method
  Table type 514
  TCursor type 584
enumFieldStruct method
  Table type 515
  TCursor type 585
enumFileList method 146
  deleting directories and 145
enumFolder procedure 378
enumFonts procedure 448
enumFormNames procedure 449
enumIndexStruct method
  Table type 516
  TCursor type 586
  useIndexes method and 587
enumLocks method
  TCursor type 588
  UIObject type 685
enumObjectNames method/procedure 686
enumOpenDatabases method/procedure 379
enumRefIntStruct method
  Table type 518
  TCursor type 589
enumReportNames procedure 449
enumRTLClassNames procedure 450
enumRTLConstants procedure 450
enumRTLMethods procedure 451
enumSecStruct method
  Table type 519
  TCursor type 590
enumSource method/procedure
  Form type 181
  Library type 236
enumSourceToFile method/procedure
  Form type 182
  Library type 237
  UIObject type 688
enumTableLinks method/procedure 183
enumTableProperties method 592
enumUIClasses procedure 689
enumUIObjectNames method/procedure 690
  Form type 184
  Report type 360
enumUIObjectProperties method/procedure
  Form type 185
  Report type 361
  UIObject type 691
enumUsers procedure 380
enumVerbs method 325
enumWindowNames procedure 452
environment variables, DOS 478, 490
eof method 657
eot method 593
equal sign (=) 479, 490
error messages
  reason method and 131
  setReason method and 132
  StatusEvent type and 398
error method
  built-in 12
  Event type 133
errorClear procedure 453
errorCode method/procedure 454
  reason method versus 131
ErrorEvent type
  defined 131
  methods 131–132
errorLog procedure 455
errorMessage procedure 456
errorPop procedure 456
errors 131–132
  built-methods for 8
  clearing 453
  codes 139
  constants for 454
  critical 454, 455, 457
  dialog box for 456
  flag status 133
  getting number 454
  handling 12
  methods for 453–458
  popping 456
  pushing 455
  quitLoop and 38
  recovering from 43
  returning messages 456
  warning 457
errorShow procedure 456
errorTrapOnWarnings procedure 457
Esc key
  dialog boxes and 473
  pop-up menus and 345

Event type
  defined 133
  methods 133–139
events
  See also actions
  built-in methods for 7–13
  disabling 18, 19
  example flow of 22
  form handling 137
  measuring time between 434
  movement 296
  passing 36
  reasons for 137, 139
  sending 197, 717
  timer 667
  UIObject name and 134
exchange method 77
exclusive locks
  See full locks
execMethod method/procedure
  Library type 238
  UIObject type 692
execute method/procedure
  DDE type 121
  System type 458
executeQBE method/procedure
  Database type 97
  Query type 348
executeQBEFile method/procedure 98
executeQBEString method/procedure 99
  executeQBE method/procedure versus 349
existDrive method 147
  isRemovable method and 158
exit procedure 458
exp method 306
exponent 314
exponentials (base e) 306
exponentiation 314
external libraries
  See libraries
external links
  See DDE type
external routines, DDE type and 120

F

fail procedure 459
  error recovery and 43
familyRights method
  Table type 520
  TCursor type 594
fieldName method/procedure
  Table type 521
  TCursor type 594
fieldNo method/procedure
  Table type 522
  TCursor type 595
fieldRights method 595
fields
  See also numeric fields
  access rights 595
  activating 7
  assigning values to 637
  blank values in 65, 370, 383
  calculated 370
  changing values of 13, 15, 749–751
  counting 370, 532, 626, 730
  counting values in 495, 562
  data type of 523, 597
  formatting values 411
  iif statements and calculated 35
  key 525, 532, 626, 627, 731
  maximum value of 496
  minimum value of 497
  mouse pointer and 9–11
  moving between 14
  moving to 7
  names of 513, 521, 594
  position of 522, 595
  reading value of 600
  searches for 619
  secondary indexes for 524
  size of 596
  sorting 641
  standard deviation 505
  validity checking 13, 612
  variance 506
fieldSize method 596
fieldType method/procedure
  Table type 523
  TCursor type 597
fieldUnits2 method 598
fieldValue method 600
fileBrowser procedure 460
files
  access rights 141, 154, 162
  closing 653
  copying 142
  creating 654
  deleting 143
  FileSystem type and 141
  listing 378
  listing information on 146

listing names of open 460
moving to beginning 657
moving to end 657
names, returning 154, 159
opening 658
pointer 651
position in 651, 656, 659, 662
reading from 89, 221, 250, 660, 661
renaming 160
reporting existence of 155
reporting extensions of 154
reporting names of 154, 155, 159
reporting on 146
reporting on modification of 166
reporting path of 151
reporting size of 163
searches in 651
searching for 148, 149
splitting path name 164
writing graphics to 223
writing memos to 251
FileSystem type
defined 141
methods 141–168
fill method/procedure
Array type 77
String type 410
filling arrays 77
financial methods 307, 313, 315, 498, 566
findFirst method 148
accessRights method and 141
name method and 159
size method and 163
findNext method 149
findFirst method and 149
fullName method and 151
name method and 159
size method and 163
floating-point numbers 300
C routines and 52
floor method 306
focus
defined 21
removing 8
setting 8
fonts, listing 448
for statements 31
loop statements and 35
quitLoop and 38
while loops versus 55
forEach statements 32
loop statements and 35
quitLoop and 38
Form Design window 176, 214
opening forms in 211
Form type 669
See also forms
defined 169
methods 169–219
TableView type and 550
Form variables 171
format procedure 411
format specifications 411
formatAdd procedure 463
formatDelete procedure 463
formatExist procedure 464
formats
adding 463
DateTime 466
deleting 463
Logical 466
LongInt 467
methods for 463–469
Number type 467
reporting on 464
SmallInt 468
Time type 469
formatSetCurrencyDefault procedure 465
formatSetDateDefault procedure 465
formatSetDateTimeDefault procedure 466
formatSetLogicalDefault procedure 466
formatSetLongIntDefault procedure 467
formatSetNumberDefault procedure 467
formatSetSmallIntDefault procedure 468
formatSetTimeDefault procedure 469
formCaller procedure 186
formReturn procedure 186
wait method and 217
forms
See also Form type; data model
activating 172
attaching methods to 198, 199
attaching variables to 171
binding UIObjects to 672
calling methods from 46
closing 7, 173, 432
creating 173
deleting methods from 197
deleting objects from 679
delivering 176
Design window and 214
designing 176, 195, 214

dialog boxes as 212
disabling methods for 217
displaying in Form Design window 211
displaying pages of 210
editing 176
event handling 135, 136
external events and 10
filtering methods of 137
Form Design window and 176, 214
handles 186
hiding 189, 216
isFirstTime method and 135
isTargetSelf method and 137
as library 47
listing methods in 181, 182
listing object properties of 185, 691
listing objects in 184, 690
listing open 449
listing source code of 687, 688
listing tables linked to 183
minimized 199
mouse methods for 200–209
moving to 209
object names 686
opening 7, 195, 210
pages in 210
position of 188, 215
returning control to 186
saving 215
sending key codes to 192
size of 190, 196
tables and 177–181
tables in data model of 177–181
titles for 188
UIObject type and 669, 672
fraction method 307
frame
See bounding box
freeDiskSpace method 150
from statement, for loops and 31
full locks
dropIndex method and 579
empty method and 682
index command and 525
lock method and 384, 531, 622
reIndexAll method and 635
rename method and 536
subtract method and 544
unlock method and 648
fullName method 151
splitFullFileName procedure and 164
functions, PAL-compatible 791–797
future value 307
fv method 307

G

getAliasPath method/procedure 380
getBoundingBox method 693
getDestination method 296
getDir method 152
getDrive method 153
getDir method versus 152
getFileAccessRights procedure 154
getKeys method 127
getLanguageDriver method 601
getLanguageDriverDesc method 601
getMenuChoiceAttribute procedure 262
getMenuChoiceAttributeById procedure 263
getMousePosition method 282
getMouseScreenPosition procedure 469
getNetUserName method/procedure 381
getObjectHit method 282
getPosition method/procedure
Form type 188
UIObject type 694
getProperty method 695
getPropertyAsString method 696
getRGB method 696
getServerName method 326
getTarget method 134
getTitle method/procedure 188
getValidFileExtensions procedure 154
Graphic fields, indexes and 525
Graphic type 64, 69
defined 220
methods 220–223
grow method 78

H

hasMenuChoiceAttribute procedure 265
hasMouse method 697
Help application, Windows 470
helpOnHelp procedure 470
helpQuit procedure 470
helpSetIndex procedure 471
helpShowContext procedure 471
helpShowIndex procedure 471
helpShowTopic procedure 472
helpShowTopicInKeywordTable procedure 472
hide method/procedure 189
bringToTop method and 172
hideSpeedBar procedure 190

hierarchy
  See containership hierarchy
home method
  TCursor type 602
  TextStream type 657
  UIObject type 698
hour method 114
hyperbolic functions
  cosine 305
  sine 318
  tangent 320

I

I-beam pointer 9
icons
  in dialog boxes 474, 476
  windows displayed as 191, 199, 216
id method
  ActionEvent type 61
  MenuEvent type 273
if statements 33
  nesting 34
  switch statements versus 42
ignoreCaseInLocate procedure 381
ignoreCaseInStringCompares procedure 418
  advMatch method and 652
  match method and 422
  search method and 424
iif statements 34
in method 308
index keyword 524
  reIndex method and 525
  reIndexAll method and 525
indexes
  compacting 499
  counting fields in 532, 627
  deleting 511, 517, 579
  dynamic arrays 33, 128
  graphic field 525
  Help 471, 472
  listing 532
  listing fields of 514, 584, 627
  listing structure of 586
  maintaining 499, 524, 567
  memo field 525
  nKeyFields method/procedure and 627
  OLE 525
  opening tables and 629
  Paradox tables 627
  primary 511, 525, 532, 579
  rebuilding 535, 536, 634, 635
  regenerating 499, 567
  searches and 614, 619, 632, 708
  secondary 511, 524, 525, 579, 586
  setIndex method and 540, 629
  specifying 511, 540, 549
  specifying dBASE 549
  switching 644, 743
  sysInfo arrays 483
  TCursor 586
  writing fields to array 514
indexOf method 79
initRecord method 603
insert method 80
insertAfter method 80
insertAfterRecord method
  TCursor type 603
  UIObject type 698
insertBefore method 81
insertBeforeRecord method
  TCursor type 604
  UIObject type 700
insertFirst method 82
insertion point
  location of 671, 672
  setFocus method and 8
insertRecord method
  TCursor type 605
  UIObject type 701
int procedure 396
integers, casting values as 396
interest rates 313, 315
  calculating 307
investments 307, 315
isAbove method 330
isAdvancedWildcardsInLocate procedure 382
isAltKeyDown method 225
isAssigned method
  AnyType type 66
  Database type 101
  Query type 350
  Session type 383
  Table type 527
  TCursor type 606
isBelow method 331
isBlank method 67
isBlankZero method/procedure 65, 383
isContainerValid procedure 701
isControlKeyDown method
  KeyEvent type 226
  MouseEvent type 283
isDir procedure 155
  setDir method and 161

isEdit method
  TCursor type 607
  UIObject type 702
isEmpty method/procedure
  Table type 528
  TCursor type 608
  UIObject type 702
isEncrypted method/procedure
  Table type 529
  TCursor type 609
isFile procedure 155
isFirstTime method 135
isFixed method 156
isFixedType method 68
isIgnoreCaseInLocate procedure 384
isIgnoreCaseInStringCompares procedure 419
isInside method 284
islastMouseClickedValid procedure 703
islastMouseRightClickedValid procedure 703
isLeapYear method 114
isLeft method 331
isLeftDown method 285
isMaximized method/procedure 190
isMiddleDown method 286
isMinimized method/procedure 191
isPreFilter method 136
isRecordDeleted method 610
  UIObject type 704
isRemote method 157
isRemovable method 158
isResizeable method 82
isRight method 332
isRightDown method 286
isShared method/procedure
  Table type 529
  TCursor type 610
isShiftKeyDown method 227, 287
isShowDeletedOn method 611
isSpace method 419
isSpeedBarShowing procedure 191
isTable method/procedure
  attach method and 493
  Database type 102
  Table type 530
isTargetSelf method 137
isValid method 612
isVisible method/procedure 192
items (array)
  See arrays

J

justification 411

K

key codes
  generating 705
  returning 224
  sending to forms 192
  virtual 420, 421
key fields 731
  See also indexes
  counting 626, 627, 731
key violations 555, 559
keyboard constants 231
keyChar method
  built-in 12
  Form type 192
  UIObject type 705
KeyEvent type
  See also keystrokes
  defined 224
  methods 224–234
keyNameToChar procedure 420
keyNameToVKCode procedure 420
keyPhysical method
  built-in 11
  Form type 193
  UIObject type 705
keystrokes
  See also KeyEvent type
  action method and 12
  ANSI values of 225
  character values of 224
  intercepting 11
  keyPhysical method and 11
  methods 224–234
  virtual characters 231, 232, 233
KEYVIOLS.DB file 555
keywords, table of 789–790
killTimer method 667, 706

L

LastMouseClicked variable 22
lastMouseRightClicked variable 22
leap year 114
libraries
  See also DLL; Library type
  adding code to 235
  associating variables with 239
  calling methods from 238

calling routines from external 46
closing 235
constants for 239
creating 235
listing source code 236, 237
methods for 235–240
ObjectPAL 46
scope of 239
Library command 235
Library type
See also libraries
defined 235
methods 235–240
line feed characters
in text files 663
removing from strings 424
link libraries
See DLL
load method
Form type 195
open method versus 211, 362
Report type 362
loan calculations 313, 315
locate method
case sensitivity and 381, 384
TCursor type 613
UIObject type 707
wildcards and 369, 382
locateNext method
TCursor type 614
UIObject type 708
locateNextPattern method
TCursor type 615
UIObject type 709
locatePattern method
TCursor type 617
UIObject type 711
locatePrior method
TCursor type 619
UIObject type 712
locatePriorPattern method
TCursor type 620
UIObject type 713
lock method/procedure
protect method/procedure versus 534
Session type 384
Table type 531
TCursor type 622
Locked property 715
lockRecord method
TCursor type 622
UIObject type 622, 715
locks
See also networks; specific lock types
counting 623, 716
explicit 390, 623, 624, 648
full 384, 525, 531, 544, 622, 623, 648
listing 439, 588, 685
network 441
read 384, 531, 622, 623, 648
record 622, 649, 715, 745
retry period and 388, 389, 525
returning type 623
Session type and 367
table 384, 390, 531, 547, 622, 623, 648, 716
write 384, 531, 622, 623, 624, 648, 715, 745
lockStatus method
TCursor type 623
UIObject type 716
log method 309
logarithms 308, 309
logical operators 241
logical procedure 241
Logical type 64, 69, 241
defined 241
formats for 412, 415, 466
methods 241–242
LongInt procedure 247
LongInt type 64, 69
alternate syntax and 392
converting from Number type to 247
declaring 247
defined 243
formats for 467
methods 49, 243–248
lookup tables 13, 515, 585, 824
loop statements 35
for loops and 32
forEach loops and 33
scan loops and 41
while loops and 55
loops
basic language elements and 27
interrupting 177
terminating 38
lower method 421
lowercase, converting to 421
lTrim method 422

M

maintained keyword 524
makeDir method 158
MAST application 470

match method 422
max procedure 309
maximize method/procedure 196
.MDX files 525
  for dBASE indexes 499
memo fields, indexes and 525
memo procedure 249
Memo type 64, 69
  declaring 249
  defined 249
  methods 249–251
memos
  formatting 249
  transferring 249
Menu type
  defined 252
  methods 252–272
menuAction method 197, 717
  built-in 12
  modifying 276
  pop-up menus and 336, 343
  show method and 272
menuChoice method 276
MenuEvent type
  constants for 277
  defined 273
  methods 273–279
  reasons for 279
  returning info on 273, 277
  specifying info on 278
menus
  See also Menu type; MenuEvent type; pop-up menus
  accelerator keys in 256
  adding pop-up menus to 254
  adding text to 255, 256
  aligning items in 256
  appending array items to 253
  cascading 254, 339
  choosing items from 12, 276
  constants for 274
  counting items in 260
  display attributes of 256, 262, 263, 265, 268, 270
  displaying 271
  drop-down 254
  handling choices in 272
  identifying items in 259
  new rows in 253
  ObjectPAL versus Paradox 252
  rebuilding 261
  removing 267
  removing items from 261, 266
  reporting type 277
  restoring default 267
  searching items in 259
  standardizing across forms 252
message procedure 473
messages
  See also error messages
  displaying in status line 473
  methods for 473–477
  status 12, 398–402
  System type and 431
method keyword 35
methodDelete method
  Form type 197
  UIObject type 718
methodGet method
  Form type 198
  UIObject type 719
methods
  See also procedures; specific method names
  aborting 459
  arrays and 72
  attaching to form 198, 199
  attaching to objects 357
  breakpoints 435
  built-in 5–25
  causing to fail 459
  custom 21, 235, 236, 238, 692
  debugging 435
  delaying execution of 19, 481
  deleting 197, 718
  disabling 217, 481, 553
  enabling 186
  example flow of 22
  financial 307, 313, 315, 498, 566
  keyword location and 19
  listing 451
  listing source code of 181, 687
  procedures versus 36, 38
  return statements and 40
  returning source code of 719
  scope of 36
  suspending execution 19
methodSet method
  create method and 174
  Form type 199
  UIObject type 720
Microsoft Windows
  See Windows
milliSec method 115
min procedure 310
minimize method/procedure 199

minute method 116
mod method 311
modulus, returning 311
month method 116
mortgages 315
mouse
  See also MouseEvent type
  arrow pointer 9
  built-in methods 9–11
  built-in variables buttons 22
  coordinates of 11
  displaying pointer 480
  Form type methods 200–209
  I-beam pointer 9
  pointer position 293, 294, 295, 469, 697
  pointer shape 480
  pop-up menus and 336
  position of 282, 284, 288, 290, 469, 480
  pushButton method and 13, 735
  reporting on 285–287, 703, 747
  simulating clicks 289–291
  UIObject type methods 720–726
mouseClick method 11
  UIObject type 720
mouseDouble method
  built-in 10
  Form type 200
  UIObject type 721
mouseDown method
  built-in 10
  Form type 201
  lastMouseClicked variable and 22
  UIObject type 721
mouseEnter method
  built-in 9
  Form type 202
  UIObject type 722
MouseEvent type
  See also mouse
  Ctrl key and 283, 288
  defined 281
  methods 281–295
  Shift key and 287, 292
mouseExit method
  built-in 9
  Form type 203
  UIObject type 723
mouseMove method
  built-in 11
  Form type 204
  UIObject type 723
mouseRightDouble method
  Form type 205
  UIObject type 724
mouseRightDown method
  Form type 206
  UIObject type 724
mouseRightUp method
  Form type 207
  UIObject type 725
mouseUp method
  built-in 10, 13
  Form type 208
  UIObject type 726
MoveEvent type
  defined 296
  methods 296–299
moveTo method/procedure
  active variables and 21
  Form type 209
  UIObject type 10, 727
moveToPage method/procedure
  Form type 210
  Report type 363
moveToRecNo method
  TCursor type 624
  UIObject type 728
moveToRecord method
  TCursor type 625
  UIObject type 729
moy method 117
msgAbortRetryIgnore procedure 473
msgInfo procedure 474
msgQuestion procedure 474
msgRetryCancel procedure 475
msgStop procedure 476
msgYesNoCancel procedure 476
multi-user environments
  See networks

N

name method 159
natural logarithm 308
networks
  See also access rights; passwords
  locks and 441, 525
  ODAPI information 442
  refresh rate 440
  reporting on drives in 157
  retry period 441
  user information 442, 443
  user name 381

newValue method 13, 749
  changeValue method and 15
  open method and 14
  reason method and 138
nextRecord method
  TCursor type 625
  UIObject type 729
nFields method/procedure
  Table type 532
  TCursor type 626
  UIObject type 730
nKeyFields method/procedure
  Table type 532
  TCursor type 627
  UIObject type 626, 731
No buttons, in dialog boxes 474, 476
NOT operator 241
nRecords method/procedure
  dBASE tables and 628, 731
  Table type 533
  TCursor type 628
  UIObject type 731
null strings 427
number procedure 311
Number type 64, 69, 300
  converting to LongInt 247
  converting to SmallInt 396
  declaring 311
  formats for 467
  LongInt variables and 243, 300
  methods 301–321
  SmallInt variables and 300, 392
numbers
  See also numeric fields
  absolute value 301
  alternate syntax for 300
  arc cosine 302
  arc sine 302
  arc tangent 303
  average 494, 562
  comparing 309, 310
  converting strings to 312
  cosine 304
  dividing 311
  exponential 306
  floating-point 300
  formatting 301, 411
  fractional part 307
  hyperbolic cosine 305
  hyperbolic sine 318
  hyperbolic tangent 320
  logarithms 308, 309
  maximum value 496, 564
  minimum value 497, 565
  raising to powers 314
  random 316
  rounding 304, 306, 317
  sample variance 506, 573
  sine 318
  square root 319
  standard deviation 505, 507, 572, 574
  sum 508, 575
  tangent 319
  truncating 320
  variance 509, 576
numeric fields
  See also numbers
  average value in 494, 562
  blank values in 370, 383
  counting 495
  dBASE tables 597, 599
  maximum value in 496, 564
  minimum value in 497, 565
  sum of values in 508, 575
numeric values
  See numbers
numVal procedure 312

O

object types
  See data types
ObjectPAL keyword 46
objects
  See also names of specific objects; properties; UIObjects
  access rights 520, 594
  active 21
  assigning values to 28
  binding 672
  binding TCursor to 557
  bounding box for 693
  built-in methods for 6, 13–20
  counting columns of 730
  displaying value of 745
  Focus property 8
  listing methods of 182
  listing names of 184, 360, 690
  listing properties of 691
  listing source code of 181, 687, 688
  listing types of 450, 689
  mouse clicks and 703
  mouse position and 697
  moving between 296

moving to 7
names 686
position of 694
searching for 727
setting position of 739
TableView 552
variables (built-in) 20–22
ODAPI 377, 442
OEM characters 409
converting strings to 423
converting to ANSI 428
oemCode procedure 423
OK buttons, in dialog boxes 474, 476
OLE type 64, 69
actions supported by server 325
copying to Clipboard 328
defined 322
indexes and 525
launching server 323
methods 322–328
pasting from Clipboard 323, 327
reporting server name 326
on clauses, index statement and 525
open method
built-in 7
Database type 103
DDE type 121
Form type 210
Library type 239
load method versus 211, 362
newValue method and 14
Report type 363
Session type 385
setTimer method and 741
tableName method and 645
TableView type 552
TCursor type 628
TextStream type 658
openAsDialog method 212
OR operations
bitwise 246, 394
logical 241
OR operator 405, 652

P

PAL (DOS version, compatibility with 791–797
Paradox tables
See tables
parentheses ( ) grouping symbol 405, 652
Pascal
routines, calling from ObjectPAL 46, 50
window handles and 218
passEvent statements 19, 36
disableDefault and 37
passwords 529, 609
See also access right; networks
for aliases 389
assigning 534
case sensitivity in 387
presenting for access 368
removing 387, 548
PATH variable 478, 490
payment calculations 307, 315
peObjectNotFound error code 455
periods (..) as match operator 405, 422, 652
pixels, converting to twips 477, 488, 732
pixelsToTwips method/procedure
System type 477, 488
UIObject type 732
play procedure 478
plus sign (+) search operator 405, 652
pmt method 313
point procedure 332
Point type 64, 69
See also points
declaring 332
defined 329
methods 329–335
pointer
See mouse
pointers, C routines and 52
points
See also Point type
calculating position of 329, 675
casting expressions as 332
coordinates of 333–335, 675
distance between 329
relative position of 330–332
returning mouse position as 282, 469
pop-up menus 336–347
See also menus; PopUpMenu type
adding array items to 336
adding to menu structure 254, 339
counting items in 261
creating 346
display attributes of 343
displaying 345
handling choices in 346
horizontal bars in 341
menuAction method and 336, 343
starting new columns in 338

text in 342
vertical bars in 337
population standard deviation 574
population variance 577
PopUpMenu type
See also pop-up menus
defined 336
methods 336–347
position method 659
postAction method 213, 732
postRecord method
TCursor type 630
UIObject type 733
pow method 314
pow10 method 314
powers, raising numbers to 314
present value 315
primary indexes 511, 525, 532
print method 364
printing, formatting strings for 411
priorRecord method
TCursor type 631
UIObject type 734
private directories 160
See also networks
privDir procedure 160
privileges
See access rights
proc statements 37
procedures
See also methods
custom 692
declaring 36
defining 37
methods versus 36, 38
scope of 38
storing 235
programs
See applications
PROMPT variable 478, 490
properties
See also methods; objects; UIObjects
database engine 377
getting value of 695, 696
listing 185, 361, 592, 689, 691
setting value of 740
TableView 550
UIObjects 799–822
protect method/procedure 534
protecting files
See locks
prototypes for methods and procedures 3, 57
pushButton method
built-in 13
UIObject type 735
pv method 315

Q

QBE 348–354
See also queries
executing queries 97
files 98
queries using 348, 350
strings 99, 104, 353
qLocate method 632
queries 348–354
See also QBE
aliases and 97
calculations in 370
secondary indexes and 524
TCursors and 97, 99
writing to file 104, 353
query keyword 97, 348, 350
Query type
defined 348
methods 348–354
returning 40
Query variables 97, 348, 350
assigning value to 350
question mark (?) wildcard character 143, 148
quitLoop statement 38
for loops and 32
forEach loops and 33
in scan loops 41
while loops and 55
quoted strings
See also strings
in match method 423
queries and 97, 349, 352
quotes ('') for empty strings 67, 403

R

radio buttons 13
rand procedure 316
random values 316
read locks
lock method and 622
unlock method and 648
readChars method 660
readEnvironmentString procedure 478

readFromClipboard method
  Graphic type 220
  OLE type 327
readFromFile method
  Binary type 89
  Graphic type 221
  Memo type 250
readLine method 661
readProfileString procedure 479
reason method
  constants for 138, 139
  error constants for 131
  errorCode method versus 131
  ErrorEvent type 131
  Event type 137
  MenuEvent type 277
  MoveEvent type 297, 398, 400
  StatusEvent type 398
recNo method 633, 636
record buffers
  emptying 603
  reading records into 576, 679
record data type 45
Record type 64, 69
  defined 355
  methods 355–356
records
  adding 491, 555
  attaching to TCursor 559
  cancelling edit of 674
  comparing 355, 544, 643
  copying 355, 570
  copying to/from arrays 569, 571, 676, 677
  counting 628, 731
  deleting 577
  deleting all 512, 581, 682
  deleting current 680
  deleting via dialog box 445
  displaying deleted 541, 611, 640
  editing all 681
  editing current 580
  ending edit 561, 582, 684
  filtering 538, 638, 738
  flyaway 578, 639, 649
  inserting 603–606, 698–701
  locking 8, 622, 715
  moving between tables 500
  moving to first 602, 698
  moving to last 582, 684
  moving to next 625, 729
  moving to previous 631
  navigating among 641, 742
  position of 633, 636
  posting 578, 612, 630, 733
  ranges 538, 638, 738
  reading into buffer 576, 679
  removing deleted 499, 567
  reporting existence of 528, 608, 702
  scan statement and 40
  searching for 614, 625, 728, 729
  setting current to prior 734
  status of 633, 735
  subtracting 544, 643
  undeleting 647, 745
  unlocking 649, 745
  updating 650
  write locks 622, 623, 715, 745
recordStatus method
  TCursor type 633
  UIObject type 735
referential integrity information, listing 518, 589
reIndex method
  index keyword and 525
  Table type 535
  TCursor type 634
reIndexAll method
  index keyword and 525
  Table type 536
  TCursor type 635
remainder, returning 311
remove method
  Array type 83
  Menu type 266
removeAlias method/procedure 386
removeAllItems method 84
removeAllPasswords method/procedure 387
removeFocus method 8
removeItem method
  Array type 85
  DynArray type 128
removeMenu procedure 267
removePassword method/procedure 387
  addPassword method and 368
rename method/procedure
  FileSystem type 160
  Table type 536
replaceItem method 85
Report Design window 360, 365
Report type
  defined 357
  methods 357–366
Report variables 357

reports
attaching variables to 357
closing 358
constants 363
designing 357, 360, 362, 365
displaying pages of 363
listing object properties of 361
listing objects in 360
listing open 449
methods for 357–366
onscreen position of 216
opening 362, 363
page numbers 359
pages 363
printing 364
Report Design window and 360, 365
titles for 357
reserved words
See keywords
resync method 736
dBASE tables and 736
Retry buttons in dialog boxes 473, 475
retry period, table locking and 388, 389, 525
retryPeriod method/procedure 388
return statements 38, 39
built-in methods and 17
in scan loops 41
rgb procedure 737
rightMouseUp method 11
rights
See access rights
round method 317
rTrim method 424
run method
Form type 174, 214
Report type 365
run-time library 451

S

sample standard deviation 572, 574
sample variance 573
save method 215
saveCFG method/procedure 388
scan statements 40
for keyword in 41
loop statements and 35
quitLoop and 38
scope
library 239
methods and procedures 36, 38, 48
variables 54, 71
screen
converting coordinates of 477, 488, 732, 744
coordinates 329
redrawing 174
scripts
errors 525
executing 478
search method 424
search operators 405, 652
searches
arrays 74, 86, 124
case sensitivity 381, 384
directories 155
file names 148, 149, 154, 155
menus 259
objects 727
records 614, 624, 625, 728, 729
secondary indexes and 524
strings 404
text file 651
values 613–622, 632, 707–714
wildcards and 369, 382
second method 118
secondary indexes 511, 524
Self variable 20
Session type
assigning variables 383
closing sessions 371
defined 367
methods 367–390
returning 40
SET command 478, 490
setAliasPath method/procedure 389
setAltKeyDown method 227
setChar method 228
setControlKeyDown method
KeyEvent type 229
MouseEvent type 288
setData method 278
setDir method 161
isDir method and 161
setDrive method versus 161
setDrive method 161
setErrorCode method 139
disabling internal events using 18
setExclusive method 537
setFieldValue method 637
setFileAccessRights procedure 162
setFilter method
Table type 538
TCursor type 638
UIObject type 738

setFlyAwayControl method 639
setFocus method 8
setId method
  ActionEvent type 62
  MenuEvent type 279
setIndex method 540
  nKeyFields method/procedure and 627
  opening tables and 629
setInside method 288
setItem method 122
setLeftDown method 289
setMenuChoiceAttribute procedure 268
setMenuChoiceAttributeById procedure 270
setMiddleDown method 290
setMousePosition method 290
setMouseScreenPosition procedure 480
setMouseShape procedure 480
setNewValue method 750
setPosition method/procedure
  Form type 215
  TextStream type 662
  UIObject type 739
setProperty method 740
setReadOnly method 541
setReason method
  ErrorEvent type 132
  Event type 139
  MenuEvent type 279
  MoveEvent type 298
  StatusEvent type 399
setRetryPeriod method/procedure 389
setRightDown method 291
setShiftKeyDown method
  KeyEvent type 230
  MouseEvent type 292
setSize method 86
setStatusValue method 401
setTimer method 9, 667, 741
setTitle method/procedure 216
setVChar method 231
setVCharCode method 231
setX method
  MouseEvent type 293
  Point type 333
setXY method 333
setY method
  MouseEvent type 294
  Point type 334
shared mode
  See also networks
  opening tables in 529, 610
Shift key 11
  isShiftKeyDown method and 227
  MouseEvents and 287, 292
  setShiftKeyDown method and 230
show method/procedure
  Form type 216
  Menu type 271
  PopUpMenu type 345
showDeleted method
  Table type 541
  TCursor type 640
showSpeedBar procedure 217
sign specifications 412
sin method 318
sine 318
  arc 302
  hyperbolic 318
sinh method 318
size method
  Array type 87
  Binary type 90
  DynArray type 129
  FileSystem type 163
  String type 425
  TextStream type 663
skip method
  TCursor type 641
  UIObject type 742
sleep procedure 481
SmallInt objects, cWord keyword and 49
smallInt procedure 396
SmallInt type 64, 69
  converting from Number type to 396
  declaring 396
  defined 391
  formats for 468
  methods 392–397
sort keyword
  Table type 542
sorting 641
sortTo method 641
sound procedure 482
sound, activating 431, 482
space key
  keyChar method and 12
  pushButton method and 12
space method 426
spaces
  creating strings of 426
  removing from strings 422
  blank values versus 67

SpeedBar
  displaying 191, 217
  hiding 190
splitFullFileName procedure 164
  fullName procedure and 151
sqrt method 319
square root 319
standard deviation 505, 507, 572, 574
startup directories 165
startUpDir procedure 165
status messages 12, 398–402, 473
  constants 398
status method
  built-in 12
  StatusEvent type 398
StatusEvent type
  defined 398
  methods 398–402
statusValue method 401
step statement, for loops and 32
stop sign icons 476
String procedure 426
String type 64, 69
  defined 403
  methods 404–430
  variables, Memo variables versus 403
strings
  See also quoted strings
  alternate syntax and 403
  casting values as 426
  comparing 418, 422
  converting to/from ANSI code 406, 409, 428
  converting to/from OEM code 423, 428
  converting to numbers 311, 312
  converting to/from virtual keycode 410, 420, 430
  converting values to 427
  cptr keyword and 49
  displaying 411
  empty 67, 403, 419
  formatting 411
  null 427
  printing 411
  query 100
  removing blank characters from 422
  repeating 410
  searching for 404, 615–620, 651, 709–713
  searching in 424
  of spaces 426
  splitting 407
  substrings 407, 427
  wildcards in searches 369, 382
  writing to text files 663
strVal procedure 427
Subject variable 21
substr method 427
subtract method/procedure
  Table type 544
  TCursor type 643
sums 508, 575
switch statements 41
  if statements versus 42
switchIndex method 644, 743
switchMenu keyword 346
symbols, for string searches 405, 652
syntax notation in prototypes 3, 57
sysInfo procedure 483
system environment 490
  reading info from 478
  sysInfo method 483
  writing info to file 490
System type
  defined 431
  methods 431–490

T

tab character (*) for accelerator keys 343
tab character (\t), removing from strings 422
table frames
  field names in 685
  Table type versus 491
  TableView type versus 550
Table type
  See also tables; TableView type; TCursor type
  defined 491
  methods and procedures 491–549
  returning 40
  variables 96, 97, 503
table windows
  bringToTop method and 172
  closing 551
  enumSource method and 181
  onscreen position of 216
  opening 552
  properties of 550
  TableView variables and 550
tableName method 645
tableRights method/procedure
  Table type 545
  TCursor type 646
tables
  See also dBASE tables; indexes; table windows
  access rights 368, 537, 541, 545, 590, 610, 646

adding 436
appending 491, 555
assigning variables 527
attaching variables to 493
closing 7, 551, 563
comparing 544, 643
copying 436, 500, 568
counting fields in 532, 626, 730
counting key fields in 626, 627, 731
counting records in 628, 731
creating 437, 501
data model and 177–181
dBASE vs Paradox 545, 647
decrypting 548
deleting 96, 438, 510
detaching variables from 546
disabling methods for objects in 553
documenting 181
editing 60, 681
empty 438, 512, 528, 533, 608, 702
encrypted 529, 534, 548, 609
field names in 513, 583
field types 597
getting information about 446
index switching 644
indexes 540, 627
language drivers 601
linked to form 183
listing 372
listing field structure of 515
listing locks 439
listing names of open 460
listing security information 519
locking 384, 441, 531, 622
locking during indexing 525
locks, listing 588, 685
locks, number of 623, 716
moving to first record 602, 698
moving to last record 582, 684
moving to next record 625, 729
moving to previous record 631
name of 536, 645
navigating in 60, 641, 742
opening 7, 628
opening windows 552
password protection 368, 529, 534, 548, 609
properties 550, 592
read only 541
rebuilding indexes 634
record position in 633, 636
referential integrity 518
removing 180
removing locks 648
renaming 444, 536
reporting on shared 529
resetting current record 734
restructuring 444
returning type of 545, 647
scan statement and 41
searching in 613–622, 625, 632, 707–714, 727–729
sorting 445, 641
subtracting records from 445
testing for 102, 530, 647
testing for invalid move 560, 593
unattaching variables from 493
unlocking 390, 547
TableView type
See also Table type; TCursor type
defined 550
methods 550–553
Table type versus 491
table windows and 550
tan method 319
tangent 319
arc 303
hyperbolic 320
tanh method 320
TCursor type
See also Table type; TableView type
defined 554
methods 554–650
returning 40
Table type versus 491
UIObject type versus 669
TCursors
assigning variables 606
attaching to record 559
binding to object 557
create statements and 502
listing structure of 585
location of 556, 557
locks and 384, 588
queries and 97
referential integrity 589
resynchronizing to UIObject 736
scanning variables 40
secondary indexes 586
TableView type versus 550
text
See also ANSI characters; characters; Memo type; strings
adding to menus 255, 256, 342
form titles 188, 216
formatting 249

memo 249
reading from file 250
writing to disk 654
writing to file 251
text buffers 654
text files
See also TextStream type
beginning 657
closing 653
creating 654
end of file 657
moving pointer 656
number of characters 663
opening 658
pointer position 659, 662
position in 656, 659, 662
reading from 660, 661
writing strings to 663
text strings
See strings
TextStream type
defined 651
methods 651–664
tilde variables (~) in queries 97, 100, 349, 352
time method/procedure
FileSystem type 166
Time type 665
Time type 64, 69
See also Date type; DateTime type
declaring 665
defined 665
formats for 469
methods 665–666
separators for 665
timer events 667
disabling timer 667
simulating 667
starting timer 741
stopping timer 706
timer method 9, 741
TimerEvent type
defined 667
methods 667–668
TimerEvent type, sleep procedure and 481
title bars
in dialog boxes 473
in menus 342
returning text of 188
setting text 216
titles, report 357
to statement, for loops and 32
toANSI method 428
today procedure 108
toOEM method 428
totalDiskSpace method 166
Touched property 13
Tracer window
See also Debugger
clearing 484
closing 485
displaying 486
hiding 485
saving contents 486
writing message to 487
tracerClear procedure 484
tracerHide procedure 485
tracerOff procedure 485
tracerOn procedure 486
tracerSave procedure 486
tracerShow procedure 486
tracerToTop procedure 487
tracerWrite procedure 487
truncate method 320
try statements, fail procedure and 43, 459
twips 329
converting pixels to 477
converting to pixels 488, 744
twipsToPixels method/procedure
System type 488
UIObject type 744
type method
Table type 545
TCursor type 647
type statements 44
Type window, records defined in 355
types
See data types

U

UIObject type
constants 678
defined 669
methods, assignment operator (=) and 28
methods 670–748
TCursor type versus 669
UIObjects
See also objects; names of specific objects
attach method and 28
binding objects to variables 672
binding variables to self 672
creating 678
displaying value of variables 745

- getting name of 134
- list of 669
- listing 184, 360
- listing properties of 185, 361, 691
- locks and 685
- mouse methods for 720, 721, 726, 735
- properties 799–822
- resynchronizing to TCursor 736
- Self variable and 21

unAssign method 68
unAttach method
- attach method and 493
- Table type 546

unDeleteRecord method 647, 745
unlock method/procedure
- Session type 390
- Table type 547
- TCursor type 648

unlockRecord method
- TCursor type 649
- UIObject type 745

unProtect method/procedure 548
updateRecord method 650
upper method 429
uppercase, converting to 429
useIndexes method
- enumIndexStruct method and 587

user count 371
user interface
- *See also* UIObjects
- creating 669

uses statements 45
- declaring data types 49
- keywords specific to 49

Uses window 48
usesIndexes method 549
- Table type 499

V

validity checking
- changeValue method and 13
- create keyword and 502, 503
- enumFieldStruct method and 515, 585
- familyRights method and 521, 594
- isValid method and 612

ValueEvent type
- defined 749
- methods 749–751

values
- *See also* fields
- converting to integer 396
- converting to strings 427
- currency 92
- date 106
- displaying in dialog box 69, 355
- filling array with 77
- formatting 411
- logical 241
- returning 40
- searching for 613–622, 632, 707–714

var statements 54, 68
variables
- *See also* specific variable names
- AnyType type and 64
- assigning values to 66, 101
- binary 89
- built-in 20
- changing value in dialog box 69
- database 94, 502
- declaring 36, 44, 54, 68
- displaying value of 69, 355
- DOS environment 478
- Form 171
- generating blank values for 65
- graphic 220
- logical 241
- memo 249
- Query 97
- Report 357
- returning data type of 65, 68
- scope of 54, 71
- Session 383
- storing 235
- Table 96, 97
- unassigning 68

variance 506, 509, 573, 576
vChar method 232
vCharCode method 233
version number (Paradox) 488
version procedure 488
vertical bar (|) OR operator 405, 652
vertical bars in menus 337
view method
- AnyType type 69
- Array type 87
- DynArray type 129
- Record type 355
- UIObject type 745

virtual characters, Windows 231, 232, 233

virtual key code
  converting ANSI code to 430
  converting strings to 410
  converting to ANSI code 420
  converting to character 420
vkCodeToKeyName procedure 430

W

wait method
  close method/procedure and 553
  Form type 217
  formReturn procedure and 187, 217
  TableView type 553
warning errors 457
wasLastClicked method 747
wasLastRightClicked method 747
while statements 54
  loop statements and 35
  for loops versus 55
width specifications 412
wildcards
  advanced 369, 382
  advMatch method and 405, 652
  copy method and 142
  delete method and 143
  file searches and 148
  match method and 422
WIN.INI file 167, 479, 490
windowClientHandle method/procedure 218
windowHandle method/procedure 218
windows
  arranging 172
  closing 173, 358, 551
  constants 211
  dialog boxes 212
  displaying 191, 192, 199, 216
  handle 218
  handle (client) 218
  hiding 189, 216
  listing open 447, 452
  maximized 196
  minimized 191, 199, 216
  opening 210
  position of 188, 215
  Restore command 216
  size of 190, 196
  titles 188
Windows (Microsoft) program
  calling convention 51
  DDE protocol 120
  entering from Paradox 458
  Help application 470
  MenuEvents and 273, 278
  reporting directory of 167
  virtual characters 231–233, 783–787
  window handles 218
windowsDir procedure 167
windowsSystemDir procedure 167
winGetMessageID procedure 488
winPostMessage procedure 489
winSendMessage procedure 489
WM_CHAR message, Windows 12
WM_KEYDOWN message, Windows 12
working directory 94
workingDir procedure 168
write locks
  add method and 555
  cAverage method and 562
  cCount method and 563
  cMax method and 564
  compact method and 567
  copy method and 568
  cSamVar method and 573
  cStd method and 574
  cVar method and 577
  dropIndex method and 579
  empty method and 581
  lock method and 384, 531, 622
  lockStatus method and 623, 716
  records 622, 623, 715, 745
  reIndexAll method and 536, 635
  removing 390, 547, 624, 648
  sortTo method and 643
  unlock method and 648
writeEnvironmentString procedure 490
writeLine method 663
writeProfileString procedure 490
writeQBE method/procedure 104, 353
writeString method 664
writeToClipboard method
  Graphic type 222
  OLE type 328
writeToFile method
  Binary type 91
  Graphic type 222
  Memo type 251

X

x method
  MouseEvent type 294
  Point type 335

Y

y method
    MouseEvent type 295
    Point type 335
year method 118
Yes buttons, in dialog boxes 476

Z

zero, versus blank value 65, 68

PARADOX

FOR WINDOWS

B O R L A N D

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-8400. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom ▪ Part # PDX1110WW21775 ▪ BOR 2702