



Paradox[®] 9

ObjectPAL[™] Reference Guide

Copyright © 1999 COREL CORPORATION and COREL CORPORATION LIMITED. All rights reserved.

For more complete copyright information please refer to the About section in the Help menu of the software.

TABLE OF CONTENTS

Introduction	1	Chapter 2:	
What's in the Paradox 9 ObjectPAL Reference Guide	1	Object type reference	39
ObjectPAL Level	1	ActionEvent type	39
Syntax notation	2	User-defined constants	40
ObjectPAL prototypes	2	actionClass method	40
Required elements	2	id method	41
Informational elements	3	setId method	41
Alternate syntax	3	AddinForm type	42
Using ObjectPAL in calculated fields	4	AnyType type	43
Example of using conditional logic and ObjectPAL methods	5	blank method/procedure	44
Derived methods	6	dataType method	45
		fromHex procedure	45
		isAssigned method	46
		isBlank method	47
		isFixedType method	47
		toHex procedure	48
		unAssign method	48
		view method	49
Chapter 1:		Application type	49
Basic language elements	7	Array type	50
; { } (comments) keyword	7	addLast method	51
= (Assignment/Comparison operator) keyword	8	append method	52
const keyword	9	contains method	53
disableDefault keyword	10	countOf method	54
doDefault keyword	10	empty method	54
enableDefault keyword	11	exchange method	55
for keyword	11	fill method	55
forEach keyword	12	grow method	56
if keyword	13	indexOf method	57
iif keyword	14	insert method	58
loop keyword	14	insertAfter method	58
method keyword	15	insertBefore method	59
passEvent keyword	16	insertFirst method	59
proc keyword	16	isResizeable method	60
quitLoop keyword	17	remove method	60
return keyword	18	removeAllItems method	61
scan keyword	18	removeItem method	62
switch keyword	19	replaceItem method	63
try keyword	21	setSize method	63
type keyword	22	size method	64
uses keyword	22	view method	64
ObjectPAL uses block	23	Binary type	65
DLL uses block	28	clipboardErase method	65
Calling external routines	32	clipboardHasFormat procedure	65
Using C++	32	enumClipboardFormats method	66
Passing by value	33	readFromClipboard method	66
Passing by pointer	34	readFromFile method	67
Returning values	35	size method	68
Notes on Graphic and Binary data (HANDLE)	35	writeToClipboard method	69
var keyword	35	writeToFile method	69
while keyword	36	Currency type	70
Reserved Keywords	37	currency procedure	71
Built-in object variables	37	Database type	73

beginTransaction method	74	setDestDelimitedFields method	114
close method	76	setDestDelimiter method	114
commitTransaction method.	76	setDestFieldNamesFromFirst method	115
delete method/procedure	77	setDestSeparator method	116
enumFamily method/procedure	77	setKeyviol method	116
getMaxRows method	78	setProblems method	117
isAssigned method	79	setSource method	117
isSQLServer method	79	setSourceCharSet method	118
isTable method/procedure	80	setSourceDelimitedFields method	119
open method/procedure.	81	setSourceDelimiter method	119
rollBackTransaction method	82	setSourceFieldNamesFromFirst method	120
setMaxRows method	82	setSourceRange method	121
transactionActive method	83	setSourceSeparator method	122
DataTransfer type	83	transferData method	123
appendASCIIIFix procedure	84	Date type	123
appendASCIIVar procedure	85	date method	124
dlgExport procedure	85	dateVal procedure	125
dlgImport procedure.	85	today procedure.	125
dlgImportASCIIIFix procedure	86	DateTime type	126
dlgImportASCIIVar procedure	87	dateTime method	127
dlgImportSpreadsheet procedure	87	day method	127
dlgImportTable procedure	88	daysInMonth method	128
empty method.	89	dow method	128
enumSourcePageList method	89	dowOrd method	129
enumSourceRangeList method	90	doy method	129
exportASCIIIFix procedure	90	hour method	130
exportASCIIVar procedure	91	isLeapYear method	130
exportParadoxDOS procedure	92	milliSec method	131
exportSpreadsheet procedure	93	minute method	131
getAppend method	94	month method	132
getDestCharSet method	94	moy method	133
getDestDelimitedFields method	95	second method	133
getDestDelimiter method	96	year method	134
getDestFieldNamesFromList method	97	DDE type	134
getDestName method	97	close method	135
getDestSeparator method	98	execute method	136
getDestType method	99	open method.	136
getKeyviol method	100	setItem method	137
getProblems method	100	DynArray type	138
getSourceCharSet method	101	contains method.	139
getSourceDelimitedFields method	102	empty method	140
getSourceDelimiter method	103	getKey method	141
getSourceFieldNamesFromFirst method.	104	removeItem method	142
getSourceName method	104	size method	142
getSourceRange method	105	view method	143
getSourceSeparator method	106	ErrorEvent type	144
getSourceType method	106	User-defined error constants	144
importASCIIIFix procedure	107	reason method	145
importASCIIVar procedure.	108	setReason method	145
importSpreadsheet procedure	109	Event type	146
loadDestSpec method	111	errorCode method	146
loadSourceSpec method	111	getTarget method	147
setAppend method	112	isFirstTime method	148
setDest method	112	isPreFilter method	149
setDestCharSet method.	113	isTargetSelf method.	149

reason method	150	create method	193
setErrorCode method	150	delayScreenUpdates	193
setReason method	151	deliver method	195
FileSystem type	152	design method	195
accessRights method	153	disableBreakMessage procedure	196
clearDirLock procedure	153	disablePreviousError procedure	196
copy method	154	dmAddTable method/procedure	197
delete method	154	dmAttach method/procedure	197
deleteDir method	155	dmBuildQueryString method/procedure	198
drives method	156	dmEnumLinkFields method/procedure	199
enumFileList method	157	dmGet method/procedure	200
existDrive method	158	dmGetProperty method/procedure	201
findFirst method	158	dmHasTable method/procedure	203
findNext method	160	dmLinkToFields method/procedure	203
freeDiskSpace method	161	dmLinkToIndex method/procedure	206
freeDiskSpaceEx method	162	dmPut method/procedure	208
fullName method/procedure	162	dmRemoveTable method/procedure	208
getDir method	163	dmResync method/procedure	208
getDrive method	163	dmSetProperty method/procedure	210
getFileAccessRights procedure	164	dmUnlink method/procedure	211
getValidFileExtensions procedure	164	enumDataModel method/procedure	212
isAssigned method	165	Property Names for enumDataModel	212
isDir procedure	165	enumSource method	213
isFile procedure	166	enumSourceToFile	214
isFixed method	167	enumTableLinks method/procedure	215
isRemote method	167	enumUIObjectNames method	216
isRemovable method	168	enumUIObjectProperties method	217
isValidDir procedure	168	formCaller procedure	217
isValidFile procedure	169	formReturn procedure	218
makeDir method	169	getFileName method/procedure	219
name method	170	getPosition method/procedure	220
privDir procedure	170	getProtoProperty method/procedure	220
rename method	171	getSelectedObjects	221
setDir method	171	getStyleSheet method/procedure	222
setDirLock procedure	172	getTitle method/procedure	223
setDrive method	173	hide method/procedure	223
setFileAccessRights procedure	174	hideToolBar procedure	224
setPrivDir procedure	175	isAssigned method	224
setWorkingDir procedure	177	isCompileWithDebug method	225
shortName method	179	isDesign method/procedure	226
size method	180	isMaximized method/procedure	227
splitFullFileName procedure	180	isMinimized method/procedure	227
startUpDir procedure	182	isToolBarShowing procedure	228
time method	183	isVisible method/procedure	228
totalDiskSpace method	183	keyChar method	228
totalDiskSpaceEx method	184	keyPhysical method	229
windowsDir procedure	184	load method	231
windowsSystemDir procedure	185	maximize method/procedure	232
workingDir procedure	185	menuAction method/procedure	233
Disk errors	186	methodDelete method	233
Form type	188	methodEdit method	234
action method/procedure	189	methodGet method	234
attach method	190	methodSet method	234
bringToTop method/procedure	191	minimize method/procedure	235
close method/procedure	192	mouseDouble method	235
		mouseDown method	236

mouseEnter method	237	enumSourceToFile method.	274
mouseExit method	238	execMethod	275
mouseMove method.	239	isAssigned method	276
mouseRightDouble method	239	methodEdit method	276
mouseRightDown method	240	open method.	277
mouseRightUp method.	241	Logical type	278
mouseUp method	242	logical procedure	278
moveTo method	243	LongInt type	279
moveToPage method/procedure	244	bitAnd method	280
open method.	244	bitIsSet method	281
openAsDialog method	246	bitOR method	282
postAction method	247	bitXOR method	283
run method	248	LongInt procedure	284
save method	248	Mail type	284
saveStyleSheet method.	249	addAddress method	284
selectCurrentTool method	249	addAttachment method	285
setCompileWithDebug method	250	addressBook method	286
setIcon method/procedure	252	addressBookTo method	286
setMenu method.	252	empty method	287
setPosition method/procedure	253	emptyAddresses method	288
setProtoProperty method/procedure	254	emptyAttachments method	288
setSelectedObjects method	254	enumInbox method.	289
setStyleSheet method/procedure	255	getAddress method	290
setTitle method/procedure	255	getAddressCount method	290
show method/procedure	256	getAttachment method.	291
showToolBar procedure	256	getAttachmentCount method	291
wait method	256	getMessage method.	292
windowClientHandle method/procedure	257	getMessageType method	292
windowHandle method/procedure	257	getSender method	293
writeText method	258	getSubject method	293
Graphic type	258	logoff method	294
readFromClipboard	259	logoffDlg method	294
readFromFile	260	logon method	295
writeToClipboard method	260	logonDlg method	295
writeToFile method	261	readMessage method	296
KeyEvent type.	261	send method	297
Keys and virtual key codes.	262	sendDlg method.	297
char method	263	setMessage method.	298
charAnsiCode.	264	setMessageType method	299
isAltKeyDown method	264	setSubject method	299
isControlKeyDown method	265	Memo type	300
isFromUI method	265	memo procedure	300
isShiftKeyDown method	266	readFromClipboard method	301
setAltKeyDown method	266	readFromFile method	302
setChar method	267	writeToClipboard method	303
setControlKeyDown method	267	writeToFile method	303
setShiftKeyDown method	268	writeToRTFFile method.	304
setVChar method	269	readFromRTFFile method	304
setVCharCode	269	Menu	305
vChar	269	addArray method	305
vCharCode.	271	addBreak method	306
Library type	272	addPopUp method	306
close method.	272	addStaticText method	307
create method	273	addText method.	308
enumSource method	273	contains method.	311

count method	312	exp method	353
empty method	313	floor method	353
getMenuChoiceAttribute procedure	313	fraction method	353
getMenuChoiceAttributeById procedure	314	fv method	354
hasMenuChoiceAttribute procedure	316	ln method	355
menuSetLimit procedure	317	log method	355
isAssigned method	317	max procedure	356
remove method	318	min procedure	356
removeMenu procedure	318	mod method	357
setMenuChoiceAttribute procedure	319	number procedure	357
setMenuChoiceAttributeById procedure	321	numVal procedure	358
show method	323	pmt method	358
MenuEvent type	324	pow method	359
User-defined constants	325	pow10 method	360
data method	325	pv method	360
id method	325	rand procedure	361
isFromUI method	327	round method	362
menuChoice method	330	sin method	362
reason	331	sinh method	363
setData	331	sqrt method	363
setId	332	tan method	364
setReason	332	tanh method	364
MouseEvent type	332	truncate method	365
getPosition method	333	OLE type	365
getObjectHit method	334	canLinkFromClipboard method	366
isControlKeyDown	335	canReadFromClipboard method	367
isFromUI method	335	edit method	368
isInside method	336	enumServerClassNames method	369
isLeftDown method	336	enumVerbs method	369
isMiddleDown method	337	getServerName method	371
isRightDown method	337	insertObject method	372
isShiftKeyDown	338	isLinked method	375
setControlKeyDown method	338	linkFromClipboard method	375
setInside method	339	readFromClipboard method	375
setLeftDown method	339	updateLinkNow method	375
setMiddleDown method	340	writeToClipboard method	376
setMousePosition method	341	OleAuto type	377
setRightDown method	341	attach method	377
setShiftKeyDown method	342	close method	378
setX method	343	enumAutomationServers procedure	378
setY method	344	enumConstants method	379
x method	344	enumConstantValues method	379
y method	344	enumControls procedure	380
getDestination method	345	enumEvents method	380
reason method	346	enumMethods method	381
setReason method	346	enumObjects method	381
Number type	347	enumProperties method	382
abs method	349	enumServerInfo procedure	382
acos method	349	first method	383
asin method	350	invoke procedure	384
atan method	350	next method	384
atan2 method	351	open method	384
ceil method	351	openObjectTypeInfo method	385
cos method	352	openTypeInfo method	385
cosh method	352	registerControl procedure	386

unregisterControl procedure	386	readFromFile method	426
version method	387	readFromString method	426
Point type	388	removeCriteria method	428
distance method	389	removeRow method	428
isAbove method	389	removeTable method	429
isBelow method	390	setAnswerFieldOrder method	429
isLeft method	390	setAnswerName method	430
isRight method	390	setAnswerSortOrder method	430
point procedure	391	setCriteria method	431
setX method	392	setLanguageDriver method	432
setXY method	392	setQueryRestartOptions method	432
setY method	393	setRowOp method	434
x method	393	wantInMemoryTCursor method	434
y method	393	writeQBE method/procedure	435
PopUpMenu	394	Record type	436
addArray method	394	view method	437
addBar method	395	Report type	438
addBreak method	396	attach method	439
addPopUp method	396	close method.	440
addSeparator method	398	currentPage method	440
addStaticText method	399	design method	441
addText method.	400	enumUIObjectNames method.	441
isAssigned method	402	enumUIObjectProperties method	442
show method	403	load method	443
switchMenu procedure	404	moveToPage method	443
Query type	404	open method	444
appendRow method	405	publishTo	445
appendTable method	405	print method	446
checkField method	406	run method	447
checkRow method	407	setMenu method.	447
clearCheck method	408	Script type	448
createAuxTables method	408	attach method	449
createQBEString method	409	create method	449
enumFieldStruct method	409	load method	450
executeQBE method/procedure	410	methodEdit method.	451
getAnswerFieldOrder method	413	run method	451
getAnswerName method	413	Session type	452
getAnswerSortOrder method	414	addAlias method/procedure	452
getCheck method	414	addPassword method/procedure.	454
getCriteria method	415	addProjectAlias method/procedure	454
getQueryRestartOptions method	416	advancedWildcardsInLocate procedure.	456
getRowID method	416	blankAsZero method/procedure	457
getRowNo method	417	close method.	457
getRowOp method	417	enumAliasLoginInfo method	458
getTableID method	418	enumAliasNames method/procedure	459
getTableNo method	418	enumDatabaseTables method/procedure	461
hasCriteria method	419	enumDriverCapabilities procedure	461
insertRow method	420	enumDriverInfo procedure	466
insertTable method	420	enumDriverNames method/procedure	467
isAssigned method	421	enumDriverTopics procedure	467
isCreateAuxTables method.	421	enumEngineInfo procedure	468
isEmpty method.	422	enumFolder procedure.	469
isExecuteQBELocal method	423	enumOpenDatabases method/procedure	470
isQueryValid method	423	enumUsers procedure	471
query keyword	424	getAliasPath method/procedure	472

getAliasProperty method	472
getNetUserName method/procedure	474
ignoreCaseInLocate procedure	474
isAdvancedWildcardsInLocate procedure	475
isAssigned method	476
isBlankZero method/procedure	476
isIgnoreCaseInLocate procedure	476
loadProjectAliases procedure	476
lock procedure	477
open method	478
removeAlias method/procedure	479
removeAllPasswords method/procedure	480
removePassword method/procedure	480
removeProjectAlias procedure	481
retryPeriod method/procedure	481
saveCFG method/procedure	482
saveProjectAliases procedure	482
setAliasPassword method/procedure	483
setAliasPath method/procedure	483
setAliasProperty method	484
setRetryPeriod method/procedure	484
unlock procedure	485
SmallInt type	486
bitAND method	487
bitISet method	488
bitOR method	489
bitXOR method	490
int procedure	490
smallInt procedure	491
SQL type	491
executeSQL method/procedure	492
isAssigned method	493
readFromFile method	494
readFromString method	495
wantInMemoryTCursor method	497
writeSQL method/procedure	497
StatusEvent type	498
reason method	499
setReason method	499
setStatusValue method	500
statusValue meth	500
String type	502
advMatch method	503
ansiCode procedure	505
breakApart method	506
chr procedure	507
chrOEM procedure	507
chrToKeyName procedure	508
fill procedure	508
format procedure	509
isEmpty	516
ignoreCaseInStringCompares procedure	516
isIgnoreCaseInStringCompares procedure	517

Volume 2

isSpace method	518
keyNameToChr procedure	518
keyNameToVKCode procedure	518
lower method	519
lTrim method	519
match method	520
oemCode procedure	521
readFromClipboard method	521
rTrim method	522
search method	522
searchEx method	523
size method	523
sizeEx method	524
spacespace method	524
string procedure	525
strVal procedure	525
subStr method	525
toANSI method	526
toOEM method	526
upper method	527
vrCodeToKeyName method	528
writeToClipboard method	528
System type	529
beep procedure	530
close procedure	530
compileInformation procedure	531
constantNameToValue procedure	532
constantValueToName procedure	532
cpuClockTime procedure	533
debug procedure	533
deleteRegistryKey method	534
desktopMenu procedure	535
dlgAdd procedure	535
dlgCopy procedure	536
dlgCreate procedure	536
dlgDelete procedure	537
dlgEmpty procedure	537
dlgNetDrivers procedure	537
dlgNetLocks procedure	538
dlgNetRefresh procedure	539
dlgNetRetry procedure	539
dlgNetSetLocks procedure	539
dlgNetSystem procedure	540
dlgNetUserName procedure	540
dlgNetWho procedure	540
dlgRename procedure	541
dlgRestructure procedure	541
dlgSort procedure	542
dlgSubtract procedure	542
dlgTableInfo procedure	543
enableExtendedCharacters procedure	543
enumDesktopWindowHandles procedure	544
enumDesktopWindowNames procedure	544

enumEnvironmentStrings procedure	545	helpQuit procedure	577
enumExperts procedure	546	helpSetIndex procedure	577
enumFonts procedure	547	helpShowContext procedure	578
enumFormats procedure	547	helpShowIndex procedure	579
enumFormNames procedure	548	helpShowTopic procedure	580
enumPrinters procedure	548	helpShowTopicInKeywordTable procedure	580
enumRegistryKeys method	549	isErrorTrapOnWarnings procedure	581
enumRegistryValueNames procedure	550	isMousePersistent method	582
enumReportNames procedure	550	isTableCorrupt method	582
enumRTLClassNames procedure	551	message procedure	583
enumRTLConstants procedure	551	msgAbortRetryIgnore procedure	583
enumRTLErrors method	552	msgInfo procedure	584
enumRTLMethods procedure	553	msgQuestion procedure	584
enumWindowHandles procedure	554	msgRetryCancel procedure	585
enumWindowNames procedure	554	msgStop procedure	585
errorClear procedure	555	msgYesNoCancel procedure	586
errorCode procedure	555	pixelsToTwips procedure	586
errorHasErrorCode method	556	play procedure	587
errorHasNativeErrorCode method	556	printerGetInfo procedure	587
errorLog procedure	557	printerGetOptions procedure	588
errorMessage procedure	558	PrinterOptionInfo record structure	589
errorNativeCode method	558	printerSetCurrent procedure	590
errorPop procedure	558	printerSetOptions procedure	591
errorShow procedure	559	projectViewerClose procedure	592
errorTrapOnWarnings procedure	559	projectViewerIsOpen procedure	592
commands and programs	559	projectViewerOpen procedure	592
executeString method	560	readEnvironmentString procedure	593
exit procedure	561	readProfileString procedure	594
expertsDir procedure	561	resourceInfo procedure	594
fail procedure	561	runExpert procedure	595
FileBrowser	562	searchRegistry procedure	596
fileBrowserEx procedure	562	sendKeys procedure	597
formatAdd procedure	565	sendKeysActionID method	601
formatDelete procedure	565	setDefaultPrinterStyleSheet procedure	602
formatExist procedure	566	setDefaultScreenStyleSheet procedure	603
formatGetSpec procedure	566	setDesktopPreference procedure	603
formatSetCurrencyDefault procedure	567	setMouseScreenPosition procedure	604
formatSetDateDefault procedure	567	setMouseShape procedure	604
formatSetDateTimeDefault procedure	567	setMouseShapeFromFile method	605
formatSetLogicalDefault procedure	568	setRegistryValue method	606
formatSetLongIntDefault procedure	569	setUserLevel procedure	607
formatSetNumberDefault procedure	569	sleep procedure	608
formatSetSmallIntDefault procedure	570	sound procedure	609
formatSetTimeDefault procedure	570	startWebBrowser procedure	610
formatStringToDate procedure	571	sysInfo procedure	610
formatStringToDateTime method	572	Language identifiers	612
formatStringToNumber procedure	572	tracerClear procedure	613
formatStringToTime method	573	tracerHide procedure	613
getDefaultPrinterStyleSheet procedure	573	tracerOff procedure	614
getDefaultScreenStyleSheet procedure	573	tracerOn procedure	614
getDesktopPreference procedure	574	tracerSave procedure	615
getLanguageDriver procedure	574	tracerShow procedure	615
getMouseScreenPosition procedure	574	tracerToTop procedure	615
getRegistryValue method	575	tracerWrite procedure	615
getUserLevel procedure	576	twipsToPixels procedure	616
helpOnHelp procedure	576	version procedure	616

winGetMessageID procedure	616	setIndex method	679
winPostMessage procedure	617	setRange method	680
winSendMessage procedure	617	setReadOnly method	681
writeEnvironmentString procedure	618	showDeleted method	682
writeProfileString procedure	619	sort keyword.	683
Table type	620	subtract method/procedure	684
add method/procedure	620	tableRights method/procedure	684
attach method	622	type method	685
cAverage method/procedure	623	unAttach method	686
cCount method/procedure.	623	unlock method	687
cMax method/procedure	624	unProtect method/procedure.	688
cMin method/procedure	625	usesIndexes method	688
cNpv method/procedure	626	Using ranges and filters	689
compact method	627	TableView type	689
copy method/procedure	628	action method	690
create keyword	628	close method.	691
createIndex method	637	isAssigned method	692
cSamStd method/procedure	639	moveToRecord method.	692
cSamVar method/procedure	640	open method.	693
cStd method/procedure	641	wait method.	694
cSum method/procedure	642	Tcursor type	695
cVar method/procedure	642	add method	696
delete method/procedure	643	aliasName method	697
dropGenFilter method	644	atFirst method	698
dropIndex method	645	atLast method	698
empty method/procedure	646	attach method	699
enumFieldNames method	647	attachToKeyViol method	701
enumFieldNamesInIndex method	647	bot method	702
enumFieldStruct method	648	cancelEdit method	703
enumIndexStruct method	650	cAverage method	703
enumRefIntStruct method.	652	cCount method	704
enumSecStruct method	653	close method.	705
familyRights method	655	cMax method	705
fieldName method/procedure	656	cMin method.	706
fieldNo method/procedure	656	cNpv method.	707
fieldType method/procedure	657	compact method	707
getGenFilter method	659	copy method.	708
getRange method	661	copyFromArray method	709
index keyword	662	copyRecord method.	710
isAssigned method	664	copyToArray method	711
isEmpty method/procedure	665	createIndex method.	712
isEncrypted method/procedure	665	cSamStd method.	715
isShared method/procedure	666	cSamVar method	715
isTable method/procedure.	667	cStd method	716
lock method	668	cSum method.	717
nFields method/procedure	669	currRecord method	717
nKeyFields method/procedure	669	cVar method	718
nRecords method/procedure	670	deletesRecord method	719
protect method/procedure.	671	didFlyAway method.	720
reIndex method	672	dmAttach method	721
reIndexAll method	672	dropGenFilter method	721
rename method/procedure	673	dropIndex method	722
restructure method.	674	edit method	723
setExclusive method	676	empty method	724
setGenFilter method	677	end method	725

endEdit method	725	nKeyFields method	773
enumFieldNames method	726	nRecords method	774
enumFieldNamesInIndex method	726	open method	775
enumFieldStruct method	727	postRecord method	776
enumIndexStruct method	729	priorRecord method	777
enumLocks method	731	qLocate method	777
enumRefIntStruct method	731	recNo method	778
enumSecStruct method	733	recordStatus method	779
enumTableProperties method	735	reIndex method	780
eot method	736	reIndexAll method	781
familyRights method	737	seqNo method	781
fieldName method	737	setBatchOff method	782
fieldNo method	738	setBatchOn method	782
fieldRights method	738	setFieldValue method	784
fieldSize method	739	setFlyAwayControl method	784
fieldType method	740	setGenFilter method	785
fieldUnits2 method	741	setRange method	787
fieldValue method	742	showDeleted method	788
forceRefresh method	744	skip method	789
getGenFilter method	745	sortTo method	790
getIndexName method	746	subtract method	791
getLanguageDriver method	746	switchIndex method	792
getLanguageDriverDesc method	747	tableName method	793
getRange method	747	tableRights method	793
handle method	748	type method	794
home method	749	unDeleteRecord method	795
initRecord method	750	unlock method	795
insertAfterRecord method	750	unlockRecord method	796
insertBeforeRecord method	751	update method	797
insertRecord method	752	updateRecord method	798
instantiateView method	753	TextStream type	798
isAssigned method	754	advMatch method	799
isEdit method	754	close method	801
isEmpty method	755	commit method	801
isEncrypted method	756	create method	802
isInMemoryTCursor method	756	end method	803
isOnSQLServer method	757	eof method	804
isOpenOnUniqueIndex method	758	home method	804
isRecordDeleted method	758	isAssigned method	804
isShared method	759	open method	805
isShowDeletedOn method	760	position method	806
isValid method	760	readChars method	807
isView method	761	readLine method	807
locate method	761	setPosition method	809
locateNext method	762	size method	809
locateNextPattern method	763	writeLine method	810
locatePattern method	764	writeString method	810
locatePrior method	766	Time type	811
locatePriorPattern method	767	time procedure	811
lock method	768	Toolbar type	812
lockRecord method	769	addButton method	813
lockStatus method	769	attach method	814
moveToRecNo method	770	create method	815
moveToRecord method	771	empty method	815
nextRecord method	772	enumToolbarNames method	816
nFields method	773		

getPosition method	816	insertBeforeRecord method	860
getState method	817	insertRecord method	861
hide method	817	isAssigned method	862
isVisible method	818	isContainerValid procedure	862
remove method	819	isEdit method	863
removeButton method	819	isEmpty method	863
setPosition method	820	isLastMouseClickedValid method	864
setState method	820	isLastMouseRightClickedValid method	864
show method	821	isRecordDeleted method	865
unAttach method	822	keyChar method	865
showApplicationBar method	822	keyPhysical method	865
isAppBarVisible method	823	killTimer method	866
TimerEvent type	823	locate method	867
UIObject type	825	locateNext method	867
action method	826	locateNextPattern	868
atFirst method	827	locatePattern method	870
atLast method	828	locatePrior method	871
attach method	828	locatePriorPattern method	872
bringToFront method	831	lockRecord method	873
broadcastAction method	831	lockStatus method	874
cancelEdit method	833	menuAction method	875
convertPointWithRespectTo method	833	methodDelete method	875
copyFromArray method	834	methodEdit method	877
copyToArray method	835	methodGet method	877
copyToToolBar method	836	methodSet method	878
create method	837	mouseClick method	878
currRecord method	838	mouseDouble method	878
delete method	838	mouseDown method	879
deleteRecord method	839	mouseEnter method	879
dropGenFilter method	840	mouseExit method	880
edit method	840	mouseMove method	880
empty method	841	mouseRightDouble method	881
end method	843	mouseRightDown method	881
endEdit method	843	mouseRightUp method	882
enumFieldNames method	843	mouseUp method	882
enumLocks method	844	moveTo method	883
enumObjectNames method/procedure	845	moveToRecNo method	884
enumSource method	845	moveToRecord method	884
enumSourceToFile method	847	nextRecord method	885
enumUIClasses procedure	847	nFields method	885
enumUIObjectNames method/procedure	848	nKeyFields method	886
enumUIObjectProperties method/procedure	848	nRecords method	886
execMethod method/procedure	849	pixelsToTwips method	887
forceRefresh method	851	postAction method	887
getBoundingBox method	852	postRecord method	888
getGenFilter method	852	priorRecord method	889
getHTMLTemplate method	853	pushButton method	890
getPosition method	854	recordStatus method	890
getProperty method	855	resync method	891
getPropertyAsString method	856	rgb procedure	891
getRange method/procedure	856	sendToBack method	892
getRGB procedure	857	setGenFilter method	893
hasMouse method	858	setPosition method	895
home method	858	setProperty method	896
insertAfterRecord method	859	setRange method/procedure	897
		setTimer method	898

skip method	899
switchIndex method	900
twipsToPixels method	901
unDeleteRecord	901
unlockRecord method	902
view method	902
wasLastClicked method	903
wasLastRightClicked method	903
ValueEvent.	904
newValue method	905
setNewValue method	905

Appendix A:
ObjectPAL constants 907

Appendix B:
Properties and property values 1001

Introduction

These two volumes provide a reference guide to the ObjectPAL language. They include a complete language reference on ObjectPAL basic language elements, methods, and procedures in the ObjectPAL run-time library, and ObjectPAL constants. All the ObjectPAL types are organized alphabetically, listing the methods associated with each type and examples for each method.

These reference manuals are arranged 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" type section.

What's in the Paradox 9 ObjectPAL Reference Guide

This chapter explains how to set the ObjectPAL level, and describes the syntax notation used in prototypes throughout this reference.

Chapter 1, "Basic language elements," describes the ObjectPAL basic language elements, which include keywords for creating control structures, for declaring methods, procedures, and variables, and for other tasks not bound to specific object types.

Chapter 2, "Object type reference," presents the methods and procedures for each ObjectPAL type. For example, the methods and procedures for working with tables are grouped in the Table Type section, and the methods and procedures for working with disk files are grouped in the FileSystem Type section.

Types are presented in alphabetical order, from ActionEvent to ValueEvent, and within each type, the methods, procedures, and structures are listed alphabetically. When you search for a particular method or procedure, make sure you know what object you are working with and what you want to do with that object. You can also search for the methods by name using the Index.

The appendices at the end of the book list all the Constants and Properties of ObjectPAL.

Note

- The Paradox 9 ObjectPAL Reference Guide consists of two separate books. The pagination in the second book continues from the first book.

ObjectPAL Level

By setting a property in the ObjectPAL Preference dialog box, you can configure the ObjectPAL Integrated Development Environment (IDE) to show you everything in the language, or to select a subset of essential elements (the default setting). The ObjectPAL level affects the IDE only: ObjectPAL code executes identically at either level, and you can use advanced elements in code even when the level is set to Beginner.

To set the ObjectPAL level

- 1 Click Tools, Settings, Developer Preferences.
- 2 On the General page, in the ObjectPAL level panel, choose one of the following:

- Beginner—to display a subset of the language
- Advanced—to display everything

Syntax notation

The following table displays the ObjectPAL syntax notation:

Convention	Sample	Meaning
Bold font	beep()	Required element (method name or parentheses). Type the bold font convention exactly as shown. Parentheses are required, even if the method takes no arguments.
Bold italic font	<i>tableName</i>	Required element (argument). Replace with a variable, expression, or literal value.
[] (Square brackets)	[, fieldName]	Informational element indicating an optional argument.
* (Asterisk)	[, fieldName] *	Informational element indicating a repeatable argument. You choose whether to repeat this argument.
{ } (Braces and bar)	{ Yes No }	You must choose one of the values separated by the vertical bar.

ObjectPAL prototypes

Prototypes are syntax statement that are presented for each ObjectPAL method and procedure. An ObjectPAL prototype consists of required elements (displayed in **bold** or *bold italic* type) and informational elements (displayed in normal type).

In the following prototype, the method name (**sample**), the argument (*argOne*), and the parentheses are required. The argument *argTwo* is optional, and remaining code is made up of informational elements.

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

In ObjectPAL code, the following statements are valid:

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

```
; Two arguments, the return value is not used.
sample(custName, custAddress)
```

Required elements

In ObjectPAL syntax, required elements are displayed in **bold** or *bold italic* type. In the following prototype the required elements are: the method name (**load**), the parentheses, and the argument (*formName*). The rest of the prototype consists of informational elements.

```
load ( const formName String ) Logical
```

Required element	Description
Name	The name of the method or procedure

Parentheses	Parentheses are required, even if the method or procedure takes no arguments
Argument	Unless an argument is enclosed in square bracket (which makes it optional) it must be included. An argument can be a variable, an expression, or a literal (hard—coded) value. Arguments are separated by commas.

Informational elements

Informational elements are not essential to the ObjectPAL syntax that you type for a method or procedure. Instead, arguments describe how the method or procedure works. The following table describes ObjectPAL informational elements:

Element	Description
Square brackets	<p>Square brackets indicate an optional argument. For example, the square brackets in the following prototype indicate that you don't have to include <code>formTitle</code> in the syntax:</p> <pre>attach ([const formTitle String]) Logical</pre> <p>There is one exception to this rule: when an argument is an <code>arrayidh_pglos_array</code> (or <code>DynArray</code>), the syntax for the argument shows square brackets following the <code>Array</code> (or <code>DynArray</code>) keyword. For instance, the following syntax indicates that <code>enumPrinters</code> takes a resizable array as an argument:</p> <pre>enumPrinters (var printers Array[] String) Logical</pre>
Keywords	<p>Keywords displayed in normal type provide information about the arguments for a method or procedure. An argument preceded by the keyword <code>var</code> is passed by reference. An argument preceded by the keyword <code>const</code> is passed as a constant. An argument itself, without either keyword, is passed by value.</p> <p>The keyword that follows each argument specifies its data type (e.g., <code>String</code>, <code>Number</code>, <code>Table</code>, or <code>Logical</code>).</p> <p>If a method or procedure returns a value, the keyword at the end of the syntax line specifies its data type. Most ObjectPAL methods and procedures return values.</p>
Asterisks	<p>An asterisk (<code>*</code>) indicates that an argument can be repeated. For example, the following syntax indicates that <code>message</code> takes one required argument, <code>reqTxt</code>, and one or more optional arguments, represented by <code>optTxt</code>.</p> <pre>message (const reqTxt String [, const optTxt String] *)</pre>

Alternate syntax

ObjectPAL supports an alternate syntax. The standard syntax uses dot notation to specify an object, a method name, and one or more arguments. In the following prototype, **object** is an object name or UIObject variable, **methodName** represents the name of the method, and *argument* represents one or more arguments:

```
object.methodName ( argument [ , argument ] )
```

ObjectPAL's alternate syntax does not use dot notation. Instead, it specifies the object as the first argument to the method. For example,

```
methodName ( object , argument [ , argument ] )
```

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)
```

For clarity and consistency, use standard syntax when possible; however, the alternate syntax is occasionally necessary when defining the calculation for a calculated field. See specific calculation methods for more details.

Using ObjectPAL in calculated fields

A calculated field can use any of the following elements:

- literal values
- variables declared within the scope of the calculated field, and have an assigned value.
- object properties
- basic language elements
- custom methods attached to other objects or to the field itself (you must declare a UIObject variable within the scope of the calculated field and use an attach statement to associate the variable with a UIObject).
- any method or procedure in the ObjectPAL run—time library (RTL) that returns a value (including a Logical value)
- special functions (e.g., Sum and Avg) provided specifically for use in calculated fields

The following table describes these elements.

Element	Comments
5	literal value
a	literal value
x	variable (must be declared within the scope of the calculated field, and have an assigned value)
x + 5	simple expression (rules for working with variables apply)
self.Name	property (displays the field's name as a String)
theBox.Color	property (displays an integer value representing the object's color)
iif(State.Value = "CA", 0.075, 0)	basic language element iif (the value of the calculated field depends on the value of the State field object)
uio.objCustomMethod()	custom method attached to another object (must return a value)
tc.open("orders.db")	RTL method (displays True if the open succeeds; otherwise, it displays False. TCursor must be declared within the scope of the field)
Avg([DIVEITEM.Sale Price])	special function (operates on the Sale price field of the Diveitem table). The table must be in the form's data model. Quotes are not used, but spaces are allowed.

```
tc.cAverage("Sale Price")
```

RTL method (TCursor must be declared and opened previously) If a field name contains spaces, quotes are required.

Example of using conditional logic and ObjectPAL methods

When you design forms or reports, it is sometimes desirable to use calculated fields that contain conditional statements. For example, if the state is California, multiply the Amount field by 10; otherwise multiply the Amount field by 5. One of the possible applications of the `iif()` keyword is to use it to provide conditional functionality in calculated fields.

By using the `iif()` keyword to create a conditional statement in a calculated field, you can evaluate a field value to see if it meets a condition, then return a value based on the condition. In a calculated field, the `iif()` keyword can be used within another expression, for example, in combination with operators and numeric methods.

You can use certain ObjectPAL methods as part of your field calculation. Most methods that involve numeric or alphanumeric strings are available in calculated fields. Any ObjectPAL expression that evaluates to a single value is valid in a calculated field.

To use ObjectPAL in a calculated expression, type the ObjectPAL method directly into the text box for the calculated field in the Define Field Object dialog box.

The syntax for the `iif()` keyword is:

```
iif(Condition, ValueIfTrue, ValueIfFalse)
```

Condition is any expression that evaluates to a logical value of True or False; ValueIfTrue is the value returned if Condition evaluates to True; and ValueIfFalse is the value returned if the Condition evaluates to False.

The following five examples use conditional expressions in a calculated field:

Example 1

Suppose you want a sales representative to visit all the customers in the Customer table. One sales representative (named Elliot) will visit those customers in California, and another (named Dolores) will visit all customers outside of California. You can create a calculated field that returns a different value (Elliot or Dolores) based on the contents of each record's State/Prov field. Use the `iif()` ObjectPAL keyword to create the expression

```
iif([CUSTOMER.State/Prov]="CA", "Elliot", "Dolores")
```

This expression tells Paradox to return the string "Elliot" when the field value is CA, and to return the string "Dolores" when the field value is anything else.

Example 2

You can also use calculated fields to print spaces between fields when appropriate. For example, use the following procedure to print a space after the Zip/Postal Code only when the Zip/Postal Code contains a value:

```
iif([CUSTOMER.Zip/postal code] = "", "", " ")
```

Example 3

You can define a calculated field that prints a comma only when the City field contains a value. Use this technique to produce an address that contains punctuation only when appropriate.

```
iif([CUSTOMER.City] = "", "", ",")
```

Example 4

This example, based on the sample Orders table, compares the Amount Paid and the Balance Due to determine which is greater, and then display one of two messages, depending on which value is greater.

```
if([ORDERS.Amount Paid] = [ORDERS.Balance Due], "This is a preferred customer.",  
"This customer has a balance due.")
```

If the Amount Paid is greater than or equal to the Balance Due, the field reads "This is a preferred customer." Otherwise, it displays "This customer has a balance due."

Example 5

Suppose you had an employee table that had a DOB field for Date of Birth. You could use the following expression to see if today was their birthday:

```
if(month([EMPLOYEE.DOB]) = month(today()) AND day([EMPLOYEE.DOB]) =  
day(today()), "Happy Birthday!", "")
```

If the month value of the employee's date of birth is the same as the current month, and the day value of the employee's date of birth is the same as the current day, then it is the employee's birthday — display message. Otherwise, it is not the employee's birthday — do not display anything.

- The **month()** method returns the numeric month value of a date. Its syntax is month(Date).
- The **day()** method returns the numeric day value of a date. Its syntax is day(Date).
- The **today()** procedure returns the current date.

Note

- ObjectPAL supports an alternate syntax that can be useful when defining a calculated field.

Derived methods

Many object types include methods derived from similar methods defined for another type. For example, the Script type includes methods derived from the Form type. The diagram below displays the methods for the Script type when scripts were introduced in Paradox 5.0. In version 5.0, the Script type included eleven methods: seven methods derived from the Form type, and four methods defined specifically for the Script type. The derived methods are listed as Form methods, but the information applies equally to the Script type.

When methods are derived from other types, the ObjectPAL online Help displays information about the original method only. For example, when you request help on the save method, Help displays information about the save method defined for the Form type. The information that applies to forms also applies to scripts.

The online Help topic for each type includes information on its methods that are derived from other types.

Methods for the Script Type in version 9.0

Form	←	Script
deliver		attach
enumSource		create
enumSourceToFile		load
methodDelete		run
methodGet		
methodSet		
save		

Basic language elements

You can use basic language elements to assign values, call functions from dynamic link libraries (DLLs), and to build control structures like **if...then...else...endIf** loops, **while...endWhile** loops, and **switch...case...endSwitch** structures. You can also use the basic language elements to declare methods, procedures, constants, variables, and data types. Most of these elements are not bound to specific object types; they work for all object types.

; (comments)	method
{ } (comments)	passEvent
= (equals)	proc
= (assignment)	quitLoop
const	return
disableDefault	scan
doDefault	switch
enableDefault	try
for	type
forEach	uses
if	var
iif	while
loop	

; { } (comments) keyword

Designates the beginning of a comment, which is text that is ignored by the compiler. The comment extends from the comment operator (;) to the end of the current line or extends from the open brace ({) to the closing brace (}) (not end at the end of the line).

Syntax

```
; Comments
or
{Comments ...
...More Comments}
```

Description

Comments are useful for documenting code.

Note

- Multiple levels of braces are not supported. The comment will end at the first closing brace (}), regardless of the number of open braces ({} in the comment.
- You can also use two forward slashes with a semicolon as a comment operator (;//)

Example

The following example demonstrates the comment operator (;):

```
var
  x AnyType; declares the variable x of AnyType
endvar
x = 25      ; x gets a value of 25
; Comments that begin with the comment operator (;) extend only to
; the end of the current line.
```

The following example demonstrates the comment braces { } operator:

```
var
  x AnyType {declares the variable x of AnyType}
endvar
x = 25      {x gets a value of 25}
{Comments that begin with the comment braces operator extend from the opening brace to
the closing brace, regardless of the number of lines occupied.}
```

= (Assignment/Comparison operator) keyword

Syntax

itemSpec = *expression*

Description

Normally, in an expression, the = is a comparison operator that tests whether the two operands are equal. Otherwise, the = operator 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.

When you use = with numbers, you can assign a numeric value to a field or variable. For example, the following code assigns the value 1.5 to *i*.

```
i=1.5
```

You can also use Hex values, like those used in C++ or Borland Delphi, to make numeric assignments. The following lines of code assign 11 to *i*.

```
i = 0x0B
```

```
i = 11
```

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* all of the information about *fieldOne*:

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

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

```
ui = fieldTwo
```

Example

The following example shows various uses of = both as a comparison operator and as an assignment. MyTable is a table frame or multi-record object on the form which contains the fields myField and fieldOne. bigBox, bigCircle, smallBox, and smallCircle are UI Objects on the form which are contained within each other. amountField is a field UI Object on the form.

```

method pushButton(var eventInfo Event)
var
  x      AnyType
  ar     Array[5] AnyType
  w      Logical
  y, z   SmallInt
  tempAmountField  Number
  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

```

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.

Note

- You declare constants in a **const...endConst** block in ObjectPAL code or in the Const window in the Object Explorer.

Example

The following example demonstrates how `const` declares a value.

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

disableDefault keyword

Disables the default code for a built-in event method.

Syntax

```
disableDefault
```

Description

disableDefault prevents an event's built-in code from executing for the current call to a built-in event method. Normally, the built-in code executes implicitly at the end of a method, just before the **endMethod** keyword. 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 Bar.

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

doDefault keyword

Executes the default code for a built-in event 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; the other statements are ignored.

Generally, if you attach code to an object's built-in **open** method, you should call **doDefault** before calling any other method or procedure. The call to **doDefault** executes the built-in code, ensuring the object is completely opened and initialized.

Example 1

The following example demonstrates the effect of a call to **doDefault**. In the following method, the button pushes in, waits two seconds and 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

```

Example 2

The following example shows how to call **doDefault** when you attach code to an object's built-in **open** method. The following code is attached to the built-in **open** method of an unbound field object named *greetingFld*. The code calls **doDefault** to execute the built-in code and then sets the value of the field object.

```

greetingFld::open
method open(var eventInfo Event)
    doDefault
    self.Value = "Hello " + getNetUserName()
endMethod

```

enableDefault keyword

Enables the default code for a built-in event 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

In the following example, default behavior is disabled and custom methods **doOpen()** or **doQuit()** are called if the respective conditions apply. Otherwise, the default behavior is enabled.

```

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

```

for keyword

Executes a sequence of statements a specified number of times.

Syntax

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

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 keywords 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.

The arguments *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 values can be any data type represented by AnyType, except Point, Memo, Graphic, String, OLE, and Binary. Also, *counter* must be a literal value or a single-valued variable; it can't be an array element or record field value.

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 continues as long as the value of *counter* is greater than or equal to the value of *endVal*. In either case, when 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

The following example demonstrates 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
```

forEach keyword

Repeats the specified statement sequence in elements within a dynamic array, or DynArray.

Syntax

```
forEach VarName in DynArrayName  
    Statements  
endForEach
```

Description

forEach walks through the elements in a DynArray. The argument *VarName* is a String variable used as a placeholder for the DynArray indexes. The argument *DynArrayName* is a DynArray variable that identifies the DynArray to walk through. If *DynArrayName* does not exist, the **forEach** statement causes an error when the method is compiled. The *Statements* clause represents the one or more ObjectPAL statements that are to be executed for each index in the DynArray.

Generally, 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 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.

Do not call **removeItem** or **empty** to modify a DynArray in a **forEach** loop.

Example

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

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

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 recommended that you 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. As in the following code, 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

```

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 use other expressions. **iif** is especially useful in calculated fields on forms or reports where **if...endIf** statements are illegal.

Example

The following example demonstrates how an **iif** keyword returns a value.

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

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** loops and returns to the beginning of the structure. Otherwise, **loop** causes an error.

Example

The following example shows how the **loop** keyword passes control to the nearest **for** statement.

```

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

```

method keyword

Defines an ObjectPAL method.

Syntax

```

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

```

Description

method marks the beginning of a method. 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 the **method** keyword.

Also optional are one or more parameter descriptions (up to a maximum of 29) represented in the prototype by *parameterDesc*, where each description takes the following 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 differ in the following ways:

- 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.

Example

```

method pushButton (var eventInfo Event)
var
  txt String
  myNum Number

```

```

endVar
myNum = 123.321
txt = String(myNum)
msgInfo("myNum = ", txt)
endMethod

```

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 clicked, 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

```

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 must provide the following:

- the procedure name, in *ProcName*
- parentheses, even if the procedure has no arguments
- zero or more parameter descriptions (up to a maximum of 29) represented in the prototype by *parameterDesc*, where each description takes the following 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

Notes

- You declare procedures in a **proc...endProc** block in ObjectPAL code or in the Proc window in the Object Explorer.

Procedures and methods are similar. The key differences are that 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.

- The scope of a procedure depends on where it is declared.

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
```

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 a **for**, **scan**, or **while** structure.

Example

In the following example, **quitLoop** is used in a **for** loop to determine whether an array has any unassigned elements:

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

return keyword

Returns control to 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, whether or not the method or procedure is declared to return a value. The following rules apply to **return**:

- If **return** is executed within the body of a procedure, the procedure is exited.
- If **return** is executed within a method (but outside of 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

The following example adds one 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 event method, a **return** statement executes the built-in code unless you explicitly disable the code. 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
```

scan keyword

Scans the TCursor and executes ObjectPAL instructions.

Syntax

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

The colon is required, even if you omit the **for** keyword.

Description

scan searches *tcVar* (TCursor) and executes *Statements* (ObjectPAL instructions) for each record. **scan** always begins at the first record of the table and examines each record in sequence. When statements in the **scan** loop change an indexed field, that record moves to its sorted position in the table; It's possible, therefore, 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 other records are skipped. If the table is empty or if no records meet the condition, the **scan** has no effect.

scan allows you to prototype a statement sequence for a single record of a table and then place that sequence inside a **scan** loop to apply it to 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 active 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

The following 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()
```

switch keyword

Executes a specified set of statement sequences.

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

The following example creates an array of 100 random numbers and then uses the bubble sort algorithm to sort the numbers 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
```

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 **try...onFail** block builds error recovery into an application.

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

The program calling the System procedure **fail** causes a trial to fail by 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 within several procedure calls. 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 of 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, existing actions are canceled, and resources are closed.

By default, a **try...onFail** block traps critical errors only. Use **errorTrapOnWarnings** if you want a **try...onFail** block to also trap warnings.

Example

The following example attempts 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
```

type keyword

Declares data types.

Syntax

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

Description

Using **type**, you can define new data types (based on existing ObjectPAL types). Once defined, you can use these types to declare variables in methods.

Note

- You declare data types in a **type...endType** block in ObjectPAL code, or in the Type window on the Methods page of the Object Explorer.

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

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

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

Example

The following example declares a **record** *Employee* that you can use to declare variables in methods and procedures. 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.

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

uses keyword

Declares external ObjectPAL methods, types, constants, or dynamic link library (DLL) routines to use in a method or procedure.

Syntax

```
uses ObjectPAL
  [ "fileName" ]*
endUses
```

Syntax for declaring DLL routines:

```
uses LibraryName
  [ routineName ( [parameterList] ) [returnType] [[callingConvention ["linkName"]]] ]*
senduses
```

Note

- While the syntax shown above is different from the **uses** block syntax in version 5.0, any existing **uses** blocks will continue to work as before.

Description

The **uses** block, declared in an object's Uses window, makes methods, constants, and type definitions available to the object's methods and procedures. An ObjectPAL uses block is different from a DLL uses block, which is why they are discussed separately. A Uses window may contain multiple ObjectPAL or DLL **uses** blocks.

Changes to uses keyword

The **uses** keyword can now be used to specify types, methods and constants from an ObjectPAL form or library. You can use all of the types, methods, and constants in a specific library by specifying the filename of the form or library. You don't have to separately name each of the types, constants, and methods you want to use.

The syntax for specifying a DLL in a **uses** block now includes an optional calling convention that lets you control the type of call made to the DLL.

Note that Paradox for Windows 95, Windows 98, and Windows NT requires 32-bit DLL's. Any DLL compiled for 16-bit use (such as with Windows 3.1) will no longer work.

Note

- Uses can specify a path in the uses name. For example:

```
Uses "c:\\program files\\corel\\suite 9\\testing.dll"
endUses

or

Uses ObjectPal "c:\\program files\\corel\\suite 9\\testing.dll"
endUses
```

ObjectPAL uses block

To use methods, constants, or type definitions stored in an ObjectPAL library or attached to a form, write a **uses** block in an object's Uses window.

Syntax

```
uses ObjectPAL
  [ "fileName" ]*
endUses
```

Description

The keyword **ObjectPAL** indicates that you are referencing ObjectPAL forms or libraries rather than a dynamic link library (DLL).

Specify the filename of each form or library name to reference. You may use an alias or path in each specified filename. Each filename must be surrounded by quotation marks and must include the file extension .FSL or .LSL. Each form or library that you reference must be in the .FSL or .LSL format when the **uses** block is compiled.

You must open a form or library before calling a method from it; however, you can use constants and type definitions without opening the form or library.

Every form or library that you want to reference must be explicitly named in the **uses** block. You cannot, for example, have a form FORM1.FSL, with a **uses** block that references LIBRARY1.LSL, that in turn has a **uses** block that references LIBRARY2.LSL, and then use the constants, types, or method declarations defined in LIBRARY2.LSL in the code in FORM1.FSL. (In this case, you would add the **uses** block for FORM1.FSL shown below to use the constants, types, and methods from both LIBRARY1.LSL and LIBRARY2.LSL).

```
Uses ObjectPAL
  "LIBRARY1.LSL" "LIBRARY2.LSL"
endUses
```

Constants and type definitions defined in the **const** and **type** sections of a library are available for other forms, libraries, or scripts to access through a **uses** statement. All methods defined in a library are available after a library variable has been attached to the library containing the methods.

Constants and type definitions defined in the **const** and **type** sections at the *form level only* are available for other forms, libraries, or scripts to access through a **uses** statement. All methods defined on all objects of a form are available to be called after a form variable has been attached to the form containing the methods.

Procedures and variables in external forms or libraries are not available. If you need to access variables in libraries, use methods in the library to get and set the values of library variables. Then you can call those methods from your forms, libraries, or scripts to share global values.

When your code is compiled or saved, it reads the constants, types, or method declarations from the .FSL or .LSL files named in **uses** blocks. Delivered forms or libraries (.FDL and .LDL files) do not have the information required for this step, so you must have the .FSL or .LSL files available when you make changes to your code.

After you deliver your code, it will run without the .FSL or .LSL files it references. After the code is saved, it will run without the .FSL or .LSL files, as long as you don't make changes to your code.

When you change constant or type information in a form or library that other forms, libraries, or scripts reference, all the forms, libraries, or scripts need to be recompiled to use the changed values. To recompile your code, make sure you have the Show Developer Menus check box enabled in the Developer Preferences dialog box. For each library, or script, open the file in Design mode, click Program, Compile, then File, Save.

Example 1

The following example calculates interest rates by referencing an ObjectPAL library. The library, named MATHLIB.LSL, contains the method **calcInterest**, which takes two arguments: *intRate* and *nPeriods*. It returns the interest calculated.

The following code, attached to a button's Uses window, reads the declaration for the **calcInterest** method from MATHLIB.LSL so the button can use it.

```
uses ObjectPAL
  "mathlib.lsl"
endUses
```

The following code, attached to a button's built-in **pushButton** method, opens the library, reads the values of two fields on a tableframe, calls **calcInterest**, and then displays the results.

```

method pushButton(var eventInfo Event)
var
    mathLib    Library
    iRate      Number
    nPeriods   SmallInt
    interest   Number
endVar
if mathLib.open("mathlib.lsl") then
    iRate = mortgage.intRate.value
    nPeriods = mortgage.nYears.value * 12
    interest = mathLib.calcInterest(iRate, nPeriods)
    interest.view("Interest")
endif
endMethod

```

In the following example, dot notation specifies where to find the **calcInterest** method. The following statement looks in the library represented by the Library variable *mathLib*.

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

The concept for calling a method attached to another form is the same. Use dot notation to specify the form used to search for the method. The following example assumes that the Form variable *codeForm* has been previously declared, and that the form has been opened and referenced in a **uses** block.

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

Note

- With previous versions of Paradox, the **uses** block was used to declare external methods to call. The declarations are now read directly from the form or library that you are calling. You no longer have to maintain multiple copies of method declarations as they change, and Paradox reports parameter mismatches when you compile your code rather than later as your code is run.

Example 2

The following example references an ObjectPAL library named PARTS.LSL. The example shows how the **uses** block allows you to share constants, type declarations, and method declarations from forms and libraries.

The library PARTS.LSL contains a **const** block, a **type** block, and a method using the constants and type definitions.

```

const
    DefaultPartName = "N/A"
    DefaultPartNumber = "000-00"
    DefaultPricePerUnit = 1.00
endConst

type
    PartRecordType = Record
        PartName      String
        PartNumber     String
        QtyOnHand      LongInt
        QtyOnOrder     LongInt
        PricePerUnit   Currency
    endRecord
endType

method NewPart(var newPartRecord PartRecordType)
    newPartRecord.PartName = DefaultPartName
    newPartRecord.PartNumber = DefaultPartNumber
    newPartRecord.QtyOnHand = 0

```

```

    newPartRecord.QtyOnOrder = 0
    newPartRecord.PricePerUnit = DefaultPricePerUnit
endMethod

```

The following code, attached to a button's Uses window, declares *DefaultPartName*, *DefaultPartNumber*, *DefaultPricePerUnit* and *PartRecordType* from the library and declares *NewPart* so the button can use them:

```

Uses ObjectPAL
    "parts.lsl"
endUses

```

The following code, attached to a button's built-in **pushButton** method, opens the library and calls the method with a *PartRecordType* variable. Note that *PartRecordType* is a type defined in the library and is declared automatically by the **uses** block.

```

method pushButton(var eventInfo Event)
    var
        partsLib    Library
        partRecord   PartRecordType
    endVar

    if partsLib.open("parts") then
        partsLib.newPart(partRecord)
    endif
endMethod

```

Example 3

The following example references an ObjectPAL library named WINAPI.LSL. The example shows how to create a *Reference Library*, that is, a library that is only accessed at compile time for constant, type, and method declarations. An ObjectPAL Reference Library contains no ObjectPAL code, only definitions.

Certain data structures, constants, and method declarations that you develop in Paradox applications can apply to several projects. The **uses** block allows applications to access centralized libraries that have been created solely for the purpose of defining the types, constants, and method declarations used. Changes to types and constants automatically propagate to all projects referencing the information (after the projects are recompiled to include the change). An ObjectPAL Reference Library is similar to a header file (.H) in the C and C++ programming languages.

The following code is attached to the Uses window in WINAPI.LSL. It declares calls made to the Windows Application Programming Interface (API). These calls should not change, so you should have them defined in a single file that also does not change, where they can be referenced whenever needed.

```

Uses User32
    GetWindowText(hwin CLONG, title CPTR, nMaxLength CLONG) CLONG [STDCALL
"GetWindowTextA"]
    GetActiveWindow() CHANDLE [STDCALL "GetActiveWindow"]
    MessageBox(hwin CLONG, text CPTR, title CPTR, flags CLONG) CLONG [STDCALL
"MessageBoxA"]
EndUses

```

The following code is attached to the Const window in WINAPI.LSL. It assigns a constant used in the *MessageBox* call to the Windows API.

```

Const
    MB_OK = 0
EndConst

```

The following code, attached to a **pushButton** method, calls the functions from the Windows API defined in WINAPI.LSL:


```

uses ObjectPAL
    "winapi.lsl"
enduses

method pushButton(var eventInfo Event)
var
    windowHandle    LongInt
    windowTitle     String
endvar

    windowTitle = fill(" ", 80)    ; reserve 80 characters for title
    windowHandle = GetActiveWindow()
    if GetWindowText(windowHandle, windowTitle, 80) > 0 then
        MessageBox(0, windowTitle, "Title of Active Window", MB_OK)
    endif

endmethod

```

Other objects (forms, libraries, or scripts) can also access WINAPI.LSL with a **uses** block and declare USER32.DLL as the Windows functions in the system dynamic link library (DLL). It is not necessary to have WINAPI.LSL present at run time in either source (.LSL) or delivered (.LDL) form.

Example 4

The following example references an ObjectPAL library named PARTSHDR.LSL. The example demonstrates how the **uses** block enables you to share constants, type declarations, and method declarations from forms and libraries. It also demonstrates how to use a Reference Library, and that you may need to use multiple **uses** blocks to declare all the information you need.

The library PARTSHDR.LSL contains a **const** block and a **type** block. It defines some global constants and types that are to be used by several other forms and libraries. PARTSHDR.LSL is considered a Reference Library because Paradox only needs to reference the information it contains at compile time.

```

const
    DefaultPartName = "N/A"
    DefaultPartNumber = "000-00"
    DefaultPricePerUnit = 1.00
endConst

type
    PartRecordType = Record
        PartName      String
        PartNumber     String
        QtyOnHand      LongInt
        QtyOnOrder     LongInt
        PricePerUnit   Currency
    endRecord
endType

```

The library PARTS.LSL declares the **NewPart** method. It declares constants and type declarations through a **uses** block that references PARTSHDR.LSL.

```

uses ObjectPAL
    "partshdr.lsl"
endUses

method NewPart(var newPartRecord PartRecordType)
    newPartRecord.PartName = DefaultPartName
    newPartRecord.PartNumber = DefaultPartNumber
    newPartRecord.QtyOnHand = 0

```

```

    newPartRecord.QtyOnOrder = 0
    newPartRecord.PricePerUnit = DefaultPricePerUnit
endMethod

```

The following code is attached to a button's Uses window. It declares *DefaultPartName*, *DefaultPartNumber*, *DefaultPricePerUnit*, and *PartRecordType* from PARTSHDR.LSL and *NewPart* from PARTS.LSL so the button can use them:

```

Uses ObjectPAL
    "partshdr.lsl" "parts.lsl"
endUses

```

Even though PARTS.LSL has a **uses** block that references PARTSHDR.LSL, the **uses** block for this button must explicitly include the reference to PARTSHDR.LSL. An indirect reference is not sufficient. Every object that needs to declare constants, type definitions, or methods from external forms or libraries must declare the forms or libraries directly in its own **uses** block or have a definition included in the **uses** block of one of its containers.

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

```

method pushButton(var eventInfo Event)
    var
        partsLib    Library
        partRecord   PartRecordType
    endVar
    if partsLib.open("parts") then
        partsLib.newPart(partRecord)
    endif
    partRecord.view() ; display the record to show the changed values
endMethod

```

DLL uses block

To use routines stored in a dynamic link library (DLL), write a DLL **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. No matter where you write it, the basic structure (shown in the following example) is the same:

Syntax

```

uses libraryName
[ routineName ( [parameterList] ) [returnType] [[callingConvention ["linkName"]]] ]*
endUses

```

Description

The required elements in a DLL **uses** block are *libraryName* and an optional list of routines. Each routine must be specified with a *routineName* and the left and right parentheses. All other arguments are optional.

The argument *libraryName* specifies the DLL filename. Paradox assumes a file extension of .DLL or .EXE.

Each routine that you declare must include a *routineName*, the name you use in your ObjectPAL code to call the external routine.

The optional *parameterList* specifies zero or more argument names and data types.

If the routine returns a value, *returnType* specifies the return value's data type.

The *callingConvention* for a DLL call can be PASCAL, STDCALL, or CDECL.

The *linkName* argument is the name of the routine as it is defined in the DLL. It is dependent on the calling convention and is case sensitive in Windows 95, and Windows 98, and Windows NT.

Windows searches for the DLL *libraryName* in this order:

- 1 the current directory
- 2 the Windows directory (folder). You can use the FileSystem procedure **windowsDir** to find the path to this directory (typically, it's C:\WINDOWS).
- 3 the Windows system directory (folder). You can use the FileSystem procedure **windowsSystemDir** to get the path to this directory (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 (e.g., a DLL loaded automatically by Windows), you can use *libraryName* to specify the DLL's module name instead of the filename. Consult your programming language's documentation for more information about DLL module names.

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

The routines must fit one of the following descriptions:

- Routines written in assembly language, C, C++, or Pascal and stored in a Windows 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.
- Routines from the Windows API (Application Programming Interface). The Windows system is made up of several DLL's. You can use Paradox to access routines within the DLLs that comprise the Windows system.

Declare a **uses** block in an object's Uses window, and within that window, declare one **uses** block for each DLL you want to use. You don't have to declare every routine the DLL contains, just the ones you want to use. Once declared, routines are available to all methods attached to that object, to all objects that object contains, and to forms or libraries that reference the declarations through an ObjectPAL **uses** block.

In a **uses** block, declare the data types of parameters and return types using the following keywords:

Data type	Uses keyword Pascal type	ObjectPAL type	C/C++ type
16-bit integer	CWORD	SmallInt	short (short int)
			Integer

32-bit integer	CLONG	LongInt	long (long int)	Longint
Natural integer (*)	CLONG (*)	LongInt (*)	int	Integer
64-bit floating-point number	CDOUBLE	Number	double	Double
80-bit floating-point number	CLONGDOUBLE	Number	long double	Extended
String pointer	CPTR	String	char *	Pchar
Binary or graphic data	CHANDLE	Binary, GraphicHANDLE (Windows)		Thandle

The size of a natural integer is dependent upon the compiler you use to create your DLL. With Windows 95, Windows 98, and Windows NT, natural integers in C and Pascal are 32-bit integers, and map into CLONG. If your compiler uses 16-bit integers and then the arguments map into CWORD, and you must declare the arguments as CWORD.

The ObjectPAL keywords CWORD, CLONG, CDOUBLE, CLONGDOUBLE, CPTR, and CHANDLE are valid only within a DLL **uses** block. Don't use them anywhere else. They are used by Paradox to convert between the more complex (and powerful) ObjectPAL data types and the corresponding data types in C or Pascal.

Note

- Do not modify any passed CPTR. If you change the contents of a string passed as a CPTR, the string must not grow beyond the size it had when it was passed to your routine.

Example 1

The following example references a dynamic link library (DLL) named MYSTUFFEDLL. To use a DLL routine in a method, declare variables to use as arguments and 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 = 3155111
thatNum = 5535345
myResult = doSomething(thisNum, thatNum) ; call the routine, return a result
endMethod
```

In this example, notice how the variables in the method are declared as LongInt and Number, and the arguments in the **uses** block are correspondingly declared as CLONG and CDOUBLE.

Example 2

The following example uses routines from MINMAX.DLL, written using a 32-bit Pascal compiler. The code for the dynamic link library (DLL) is as follows:

```
library MinMax;

function Min(x, y: integer): integer; stdcall; export;
begin
```

```

    if x < y then
        result := x
    else
        result := y;
end;

function Max(x, y: integer): integer; stdcall; export;
begin
    if x > y then
        result := x
    else
        result := y;
end;

exports
    Min, Max;

begin
end.

```

The following ObjectPAL code uses the routines in the DLL. The code for the Uses window appears first, followed by the code that modifies a button's **pushButton** method:

```

; the following goes in a button's Uses window
uses
    MinMax ; load routines from MINMAX.DLL
    Min (x CLONG, y CLONG) CLONG [STDCALL]
    Max (x CLONG, y CLONG) CLONG [STDCALL]
endUses

```

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

```

method pushButton(var eventInfo Event)
var
    x, y, z LongInt
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

```

Example 3

The following example shows how to use ObjectPAL to call a function from the Windows application programming interface (API). It calls the Windows API function `MessageBox` to display a dialog box.

The following code is attached to a button's Uses window:

```

Uses USER32 ; The MessageBox function is in
              ; the Windows system DLL USER32.DLL
              ; usually found in C:\WINDOWS\SYSTEM
    MessageBoxA(hWnd CLONG, lpText CPTR, lpCaption CPTR, wType CLONG) CLONG
endUses

```

The following code is attached to a button's built-in **pushButton** method. It calls `MessageBox`, passing it zero for the window handle ensuring that it's not connected to any particular window. The code also passes text for the message and the caption and another zero to signify an OK-style message box. The return value is ignored.

```

method pushButton(var eventInfo event)
    MessageBoxA(0,

```

```

        "Your message here",
        "Your caption here",
        0)
endMethod

```

For more information on the parameters for this and other Windows API function calls, see the Windows API reference .

Calling external routines

Previous versions of Windows (3.1 and earlier) and Paradox used the Pascal calling convention (PASCAL). Windows 95, Windows 98, and Windows NT use a different calling convention. Paradox supports this calling convention, STDCALL, PASCAL, and the C calling convention, CDECL. Paradox defaults to STDCALL.

Convention	Push order	Restore stack	Link name	Used by
PASCAL	Left first	Callee	Uppercase	Pascal
CDECL	Right first	Caller	'_' prepended	C/C++
STDCALL	Right first	Callee	No change	Windows 95, Windows 98, Windows NT

When you declare routines to be called from a dynamic link library (DLL), you must match the calling convention that the routines were declared with. All calls to functions in the Windows 95 Application Programming Interface (API) or the Windows NT API are case-sensitive and require the use of the STDCALL calling convention.

Calls to functions written in Pascal should be declared with the PASCAL calling convention, and calls to C functions should be declared CDECL, unless the routines were explicitly declared to use a different convention when the DLL was compiled. For example, a C routine might be declared `__stdcall`, in which case you would declare it STDCALL in the **uses** block.

If you do not include a link name in the declaration, the routine name will be used in the call with any changes listed in the Link name column in the table above.

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

All C and C++ functions that you want ObjectPAL to call must be exported in the .DEF file, or tagged with `_export` in the function declaration.

Using C++

Calling dynamic link library (DLL) modules written in C++ requires either the use of a C linkage specification or the mangled name in the **uses** block.

To specify a C++ function with C linkage, the modules must be in one of the following forms:

```

extern "C" declaration
extern "C" { declarations }

```

For example, if a C module contains these functions:

```

char *SCopy(char*, char*);
void ClearScreen(void);

```

they must be declared in a C++ module in one of the following ways to have a C linkage.

```
extern "C" char *SCopy(char*, char*);
extern "C" void ClearScreen(void);
```

or

```
extern "C" {
char *SCopy(char*, char*);
void ClearScreen(void);
}
```

Otherwise, you can specify the mangled name of the routine to call. The mangled name can be found by using a dumping file on the .OBJ file produced by your compiler.

For example, if a Borland C++ module (named MyLib) contains the function

```
int __cdecl MyFunction(int arg)
```

then you can use this **uses** block to declare the DLL routine.

```
uses MyLib
MyFunction(CLONG arg) CLONG [CDECL "@MyFunction$qi"]
enduses
```

All C or C++ functions that you want to call from ObjectPAL must be exported, either by using a .DEF file or the `_export` modifier. See your C or C++ compiler documentation for more information on exporting functions when creating DLLs.

Example

The following example shows how to use ObjectPAL to call a function from the Windows application programming interface (API). It calls the Windows API function `MessageBox` to display a dialog box.

The following code is attached to a button's **Uses** window:

```
Uses USER32 ; The MessageBox function is in
; the Windows system DLL USER32.DLL
; usually found in C:\WINDOWS\SYSTEM
MessageBoxA(hWnd CLONG, lpText CPTR, lpCaption CPTR, wType CLONG) CLONG
endUses
```

The following code is attached to a button's built-in **pushButton** method. It calls `MessageBox`, passing it zero for the window handle ensuring that it's not connected to any particular window. The code also passes text for the message and the caption and another zero to signify an OK-style message box. The return value is ignored.

```
method pushButton(var eventInfo event)
  MessageBoxA(0,
    "Your message here",
    "Your caption here",
    0)
endMethod
```

For more information on the parameters for this and other Windows API function calls, see the Windows API reference .

Passing by value

The following table presents the syntax you should use when 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 with any C compiler, pass values by pointer.

It is assumed that these ObjectPAL variables have been declared: `si` SmallInt, `li` LongInt, `nu` Number, `st` String, `gr` Graphic, and `bi` Binary

C data type	C syntax	In uses block	ObjectPAL call
long double	void __stdcall cproc(long double value)	cproc(numvar CLONGDOUBLE)	cproc(si) cproc(li) cproc(nu)
double	void __stdcall cproc(double value)	cproc(numvar CDOUBLE)	cproc(si) cproc(li) cproc(nu)
long int	void __stdcall cproc(long int value)	cproc(longvar CLONG)	cproc(si) cproc(li)
short int	void __stdcall cproc(short int value)	cproc(shortvar CWORD)	cproc(si)
int	void __stdcall cproc(int value)	cproc(longvar CWORD)	cproc(si)
(String)	void __stdcall cproc(char * value)	cproc(stringvar CPTR)	cproc(st)
(Graphic)	void __stdcall cproc(HANDLE value)	cproc(bitmapvar CHANDLE)	cproc(gr)
(Binary)	void __stdcall cproc(HANDLE value)	cproc(binaryvar CHANDLE)	cproc(bi)

Passing by pointer

When ObjectPAL passes information to a C procedure that takes pointers to information, the pointer points directly to the corresponding value in the ObjectPAL object. Variables in ObjectPAL are treated as objects internally. For example, if you want an int * and you pass a LongInt, you will get a pointer that points directly to the integer value inside the LongInt object. You can then modify the value of the LongInt using the pointer in your DLL. This could, however, corrupt ObjectPAL by overwriting memory (writing past the bounds of the memory pointer). **Use caution when using pointers.**

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.

The following table presents the syntax you should use when passing various data types by pointer to a C procedure, with the assumption that these ObjectPAL variables have been declared: si SmallInt, li LongInt, nu Number, st String, gr Graphic, and bi Binary.

C data type	C syntax	In USES block	ObjectPAL call
long double *	void __stdcall cproc(long double * value)	cproc(numvar CPTR)	cproc(nu)
long int *	void __stdcall cproc(long int * value)	cproc(longvar CPTR)	cproc(li)
int *	void __stdcall cproc(int * value)	cproc(longvar CPTR)	cproc(li)
short int *	void __stdcall cproc(short int * value)	cproc(shortvar CPTR)	cproc(si)
char *	void __stdcall cproc(char * value)	cproc(strvar CPTR)	cproc(st)

Returning values

The following table presents the syntax for data type value that have been returned from a C procedure. The assumption is that these ObjectPAL variables have been declared: si SmallInt, li LongInt, nu Number, and st String

C data type	C syntax	In USES block	ObjectPAL call
long double	long double __stdcall cproc(void)	cproc() CLONGDOUBLE	nu = cproc()
double	double __stdcall cproc(void)	cproc() CDOUBLE	nu = cproc()
long int	long int __stdcall cproc(void)	cproc() CLONG	li = cproc()
short int	short int __stdcall cproc(void)	cproc() CWORD	si = cproc()
char *	char * __stdcall cproc(void)	cproc() CPTR	st = cproc()

Notes on Graphic and Binary data (CHANDLE)

Graphic and Binary data are passed via CHANDLE{bmc emdash.bmp}a handle to Windows memory. In C use the HANDLE typedef by including it inside code like this:

```
void __stdcall cproc(HANDLE value)
{
    // declare ptr to point to Global Memory Block
    huge *ptr = (huge *) GlobalLock(value);

    // ... make use of ptr here
    // ... DO NOT use 'GlobalFree(value);'
    GlobalUnlock(value);
}
```

For a Binary variable, HANDLE is a handle to memory that holds the information in the binary BLOB. There is no header information. As with any strings you pass, you can read or modify the data, but you cannot change its size.

For a Graphic variable, HANDLE is a Windows bitmap handle that you can use as you would any other bitmap HANDLE.

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.

A variable's scope depends on the block in which it is declared.

Note

- You declare variables in a **var...endVar** block in ObjectPAL code or in the Var window on the Methods page of the Object Explorer.

Example

The following example demonstrates how the var keyword declares a variable.

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

while keyword

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

Syntax

```
while Condition
  [ Statements ]
endWhile
```

Description

while evaluates the logical expression *Condition*. If *Condition* is False, the *Statements* are not executed. If the *Condition* is True, the *Statements* between *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* is 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** variable to force control back to the top of the **loop**, skipping the statements between **loop** and **endWhile**. You can also use **quitLoop** to exit the loop or nest **while** statements to any level.

while and **for** are 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

The following 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.nextRecord()
  n = n + 1
  namesArray.grow(1)
```

```
namesArray[n] = myNames."Last name"
endWhile
```

Reserved 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.

Generally, 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 event methods.

Keywords

active	endMethod	key	return
and	endProc	lastMouseClicked	scan
array	endQuery	lastMouseRightClicked	secStruct
as	endRecord	like	self
case	endScan	loop	sort
caseInsensitive	endSort	maintained	step
const	endSwitch	method	struct
container	endSwitchMenu	not	subject
create	endTry	ObjectPAL	switch
database	endType	of	switchMenu
descending	endUses	on	tag
disableDefault	endVar	onFail	then
doDefault	endWhile	or	to
dynArray	for	otherwise	try
else	forEach	passEvent	type
enableDefault	from	primary	unique
endConst	if	proc	uses
endCreate	iif	query	var
endFor	in	quitLoop	where
eEndForEach	index	record	while
endIf	indexStruct	refIntStruct	with
endIndex	is	retry	without

Built-in object variables

ObjectPAL provides built-in object variables that you can use to refer to UIObjects. These variables are particularly useful for creating generalized code. For example, when the following statement executes, it sets the color of the active object (the object that has focus). You don't have to specify the object by name.

```
active.Color = Blue
```

The built-in object variables are:

- active
- container
- lastMouseClicked

- lastMouseRightClicked
- self
- subject

Object type reference

ActionEvent type

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

The only built-in event method that is triggered by an ActionEvent is **action**. Typically, when you work with ActionEvents, you'll also work with ObjectPAL 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(UserError) ; UserError is a constant.
endif
endMethod
```

The action constants are grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use user-defined action constants.

The following table displays the methods for the ActionEvent type:

Methods for the ActionEvent type

Event	←	ActionEvent
errorCode		actionClass
getTarget		id
isFirstTime		setId
isPreFilter		
isTargetSelf		
reason		
setErrorCod		
setReason		

User-defined constants

ActionEvent

You can define your own action constants, but you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the `IdRanges` constants `UserAction` and `UserActionMax` to represent the minimum and maximum values allowed.

For example, suppose that you want to define two action constants, `ThisAction` and `ThatAction`. In a `Const` window, define values for your custom constants as follows:

```
Const
    ThisAction = 1
    ThatAction = 2
EndConst
```

Then, to use one of these constants, add it to `UserAction`. For example,

```
method action(var eventInfo ActionEvent)
    if eventInfo.id() = UserAction + ThisAction then
        doSomething()
    endif
endMethod
```

By adding `UserAction` to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, use the value of `UserActionMax`. One way to check the value is with a **message** statement:

```
message(UserActionMax)
```

In Paradox, the difference between `UserAction` and `UserActionMax` is 2047. That means the largest value you can use for an action constant is `UserAction + 2047`.

actionClass method

ActionEvent

Returns the class number of an `ActionEvent`.

Syntax

```
actionClass ( ) SmallInt
```

Description

actionClass returns an integer value representing an `ActionEvent` class. Use `ActionClasses` constants to find out which class the integer value represents.

Example

The following 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(UserError)
beep()
message("Sorry. Can't make changes to this field.")
endif
endif
endMethod
```

id method**ActionEvent**

Returns the ID number of an ActionEvent.

Syntax

```
id ( ) SmallInt
```

Description

id returns the ID number of an ActionEvent. ObjectPAL defines constants for these ID numbers (for example, DataBeginEdit), so you don't have to remember numeric values.

The action constants are grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use user-defined action constants.

Example

The following 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(UserError) ; don't start Edit mode
        msgStop("Sorry", "View only – can't edit this form")
    endif
endif
endMethod
```

setId method**ActionEvent**

Specifies an ActionEvent.

Syntax

```
setId ( const actionId SmallInt )
```

Description

setId specifies the ActionEvent represented by the constant *actionId*. ObjectPAL provides constants (e.g., DataNextRecord) for ActionEvents so you don't have to remember numeric values.

The action constants grouped as follows:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

setId method

You can also use user-defined action constants.

Example

In the following example, the Toolbar 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
```

AddinForm type

An add-in form is an external dynamic link library (DLL) that a third-party developer has provided. Not all DLLs can be used in Paradox. To use a DLL in Paradox, it must be designed so that it permits proper communication between it and Paradox. For more information on a specific add-in and whether it can be used in Paradox, contact the third-party developer who created it. For information on developing an add-in for Paradox, refer to the Paradox Developer Help for Paradox Add-Ins.

If an add-in DLL has been designed for use in Paradox, you can use the ObjectPAL AddinForm type methods to open and close the forms that the DLL contains and to obtain and set published form properties. An add-in DLL can also add menu options to the Paradox menus.

Before an add-in form can be used in Paradox, it must be registered.

Methods of the AddinForm type are similar to methods of the Form type.

Methods in the AddinForm type

Form	←	AddinForm
attach		The AddinForm type consists of derived methods from the Form type.
bringToTop		
close		
closeQuery		
enumForms		
getPosition		
getPropertyAsInteger		
getPropertyAsNumber		
getPropertyAsString		

getTitle
 hide
 isAssigned
 isMaximized
 isMinimized
 isVisible
 maximize
 menuAction
 minimize
 open
 postMessage
 sendMessage
 setPosition
 setProperty
 setTitle
 show
 wait
 windowHandle

AnyType type

An AnyType variable can store any one of the data types listed in the following table.

Type	Description
AnyType	Any basic data type
Binary	Machine-readable data
Currency	Used to manipulate currency values
Date	Calendar data
DateTime	Calendar and clock data combined
Graphic	A bitmap image
Logical	True or False
LongInt	Used to represent large integer values
Memo	Holds a large amount of text
Number	Floating-point values
OLE	A link to another application
Point	Information about a location on the screen
SmallInt	Used to represent relatively small integer values

blank method/procedure

String	Letters
Time	Clock data

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

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.)

Methods for the AnyType type

AnyType

blank
dataType
fromHex
isAssigned
isBlank
isFixedType
toHex
unAssign
view

blank method/procedure

AnyType

Returns a blank value.

Syntax

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

Description

blank 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

The following example assumes that a form has a table frame bound to the *Lineitem* table and a button named *thisButton*. When a user presses *thisButton*, the code scans the Qty field in *Lineitem* and replaces non-blank 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

```

dataType method

AnyType

Returns a string representing the data type of a variable.

Syntax

```
dataType ( ) String
```

Description

dataType returns a string representing the data type of a variable or expression: Binary, Currency, Date, DateTime, Graphic, Logical, LongInt, Memo, Number, OLE, 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

The following 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 and 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

```

fromHex procedure

AnyType

Converts a hexadecimal number to a decimal number.

Syntax

```
fromHex ( const value String ) LongInt
```

isAssigned method

Description

fromHex converts a hexadecimal number to a decimal number. The *value* must range from 0x00000000 to 0xFFFFFFFF.

Example

In the following example, the **pushButton** method for a button named *convertHex* converts a hexadecimal string variable to a decimal number.

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

    var
        s String
        li LongInt
    endVar

    ;Hexadecimal value to convert.
    s = "0x0756B5B3"
    s.view("Hex value to convert")
    li = fromHex(s)
    li.view("0x0756B5B3") ; Displays 123123123.
endMethod
```

isAssigned method

AnyType

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just AnyType.

Example

The following 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 is attached 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
```

isBlank method**AnyType**

Reports whether an expression has a blank value.

Syntax

```
isBlank ( ) Logical
```

Description

isBlank 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 example uses **isBlank** to test various values and displays the results in a dialog box. This code is attached to a button's **pushButton** method.

```
; 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
```

isFixedType method**AnyType**

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

Syntax

```
isFixedType ( ) Logical
```

Description

isFixedType returns True if the variable has been declared using a **var...Endvar** block; otherwise, it returns False. **isFixedType()** returns false for an 'ANYTYPE' variable because these variables are dynamically allocated at runtime.

Example

The following example 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
```

toHex procedure

toHex procedure

AnyType

Converts a decimal number to a hexadecimal number.

Syntax

```
toHex ( const value LongInt ) String
```

Description

toHex converts a decimal number to a hexadecimal number.

Example

In the following example, the **pushButton** method for a button named *convertDecimal* converts a long integer value to a hexadecimal string.

```
; convertDecimal::pushButton
method pushButton(var eventInfo Event)
    var
        s String
        li LongInt
    endVar

    li = 123123123
    li.view("Value to convert")
    s = toHex(li)
    s.view("123123123") ; Displays 0x0756B5B3.
endMethod
```

unAssign method

AnyType

Sets a variable's state to unAssigned.

Syntax

```
unAssign ( )
```

Description

unAssign 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
```

view method

AnyType

Displays the value of a variable in a dialog box.

Syntax

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

Description

view displays the value of a variable in a modal dialog box. 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 summarizes the AnyType variables that can be displayed, and those which the user can modify.

Type	Can be viewed	Can be modified
Binary	no	no
Currency	yes	yes
Date	yes	yes
DateTime	yes	yes
Graphic	no	no
Logical	yes	yes
LongInt	yes	yes
Memo	no	no
Number	yes	yes
OLE	no	no
Point	yes	yes
SmallInt	yes	yes
String	yes	yes
Time	yes	yes

Application type

An Application variable provides a handle for working with the desktop window of the active Paradox application. You can use an Application variable in your code to control the size, position, and appearance of the desktop, and change the working directory and the private directory at run time.

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 active Paradox desktop only; you can, however, use Session variables to open multiple channels to the database engine (see the Session type).

view method

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

```
; downSize::pushButton
method pushButton(var eventInfo Event)
var
    thisApp      Application
endVar
thisApp.maximize() ; Maximize the desktop.
endMethod
```

The following table displays the methods for the Application type, which are derived methods from the Form type.

Methods for the Application type

Form	←	Application
bringToTop		The Application type consists of derived methods from the Form type.
GetPosition		
getTitle		
hide		
isMaximized		
isMinimized		
isVisible		
maximize		
minimize		
setIcon		
setPosition		
setTitle		
show		
windowClientHandle		
windowHandle		

Array type

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. To use arrays in methods, you must declare them by specifying a name, size (number of items), and a data type for the items. An array is not derived from Anytype. Each element in the array is derived from the Anytype class.

Notes

- In ObjectPAL, array items are counted beginning with 1, not with 0, as in some other languages.
- ObjectPAL also supports dynamic arrays. For more information, see the method and procedures for DynArray.

The following table displays the methods for the Array type, including several derived methods from the AnyType type.

Methods for the Array type

AnyType	←	Array
blank		addLast
dataType		append
isAssigned		contains
isBlank		countOf
isFixedType		empty
		exchange
		fill
		grow
		indexOf
		insert
		insertAfter
		insertBefore
		insertFirst
		isResizeable
		remove
		removeAllItems
		removeItem
		replaceItem
		setSize
		size
		view

addLast method

Array

Inserts an item at the end of a resizeable array.

Syntax

```
addLast ( const value AnyType )
```

Description

addLast 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, 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:

append method

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

Example

The following example adds an element to a resizable 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 should be attached in the **Var** window for *thisButton*:

```
; thisButton::Var
var
  ar Array[] SmallInt  ; declare ar as a resizable 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
```

append method

Array

Appends the contents of one array to another.

Syntax

```
append ( const newArray Array[ ] String )
```

Description

append attaches the items of *newArray* to a resizable array. The array grows to make room for the added items.

Example

The following example creates two resizable arrays, *addMe* and *baseArray*, and loads them with numeric values. The following example appends the *addMe* array to the *baseArray* array and 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

```

contains method

Array

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

Syntax

```
contains ( const value AnyType ) Logical
```

Description

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

Example

The following example defines and loads a resizable 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    ; resizable 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 box
    dogs.view("dogs is initialized with these values")

```

countOf method

```
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 box  
    dogs.view("dogs size: " + strVal(dogs.size()))  
else  
    ; "Bandit" must already exist  
    msgInfo("Once is enough", "The dogs array already contains Bandit.")  
endif  
  
endMethod
```

countOf method

Array

Counts the occurrences of a value in an array.

Syntax

```
countOf ( const value AnyType ) LongInt
```

Description

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

Example

The following example contains code which should be attached to a button's **pushButton** method. It creates and loads a fixed-size array and 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
```

empty method

Array

Removes all items from an array.

Syntax

```
empty ( )
```

Description

empty removes all items from an array. A fixed-size array stays the same size, and all items become unassigned. A resizable array is reset to a size of 0.

Example

The following example shows how empty functions for a fixed-size array. The code immediately following declares a fixed-size 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-size 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-size 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
```

exchange method**Array**

Swaps the contents of two cells in an array.

Syntax

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

Description

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

Example

See the example for **indexOf**.

fill method**Array**

Fills an array with a value.

grow method

Syntax

```
fill ( const value AnyType )
```

Description

fill assigns a value to every item of an array.

Example

The following example creates a fixed-size 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
```

grow method

Array

Increases the size of a resizable array.

Syntax

```
grow ( const increment LongInt )
```

Description

grow appends *increment* cells to a resizable array or removes cells if the value of *increment* is negative. If you try to remove more cells than the array contains, an error occurs.

Example

The following example uses **grow** to increase and decrease the size of a resizable 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
```

indexOf method

Array

Returns the position of an item in an array.

Syntax

```
indexOf ( const value AnyType ) LongInt
```

Description

indexOf 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 the field, the last menu item selected is the first in the list of menu choices. The following code should be added 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, values are assigned 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
```

insert method

Array

Inserts one or more empty cells into an array.

Syntax

```
insert ( const index LongInt [ , const numberOfItems LongInt ] )
```

Description

insert add the number of empty cells specified by *numberOfItems* empty cells into a resizable 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 into a resizable array at two locations 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
```

insertAfter method

Array

Inserts an item into an array after a specified item.

Syntax

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

Description

insertAfter places *insertedItem* after the first occurrence of *keyItem* in a resizable array. If *keyItem* is not found, *insertedItem* is not inserted, and the indexes do not change. If *insertedItem* is inserted, indexes of subsequent items increase by one.

Example

The following example loads a resizable 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
```


insertBefore method

Array

Inserts an item into an array before a specified item.

Syntax

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

Description

insertBefore searches a resizable array for *keyItem* and inserts *insertedItem* at *keyItem*'s position. Indexes of *keyItem* (and subsequent items) are increased by one. If *keyItem* is not found, *insertedItem* is not inserted, and the indexes do not change.

Example

The following example adds an element to a resizable array using **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
```

insertFirst method

Array

Inserts an item at the beginning of an array.

Syntax

```
insertFirst ( const value AnyType )
```

Description

insertFirst inserts *value* at the beginning of a resizable array. Indexes of subsequent items are increased by one.

Example

The following example creates a resizable array and 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 –
```

isResizable method

```
                ; 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
```

isResizable method

Array

Reports whether an array can be resized.

Syntax

```
isResizable ( ) Logical
```

Description

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

Example

The following example verifies whether 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.isResizable() = True then ; if array can be resized
    myArray.grow(5) ; add 5 cells to it
else
    msgStop("Problem", "Array cannot be resized.")
endif
endMethod
```

remove method

Array

Removes one or more items from an array.

Syntax

```
remove ( const index SmallInt [ const numberOfItems SmallInt ] )
```

Description

remove deletes the number of items specified by *numberOfItems* items (or one item if *numberOfItems* is not specified) from an array. Indexes of subsequent items are decreased by *numberOfItems* (or one if *numberOfItems* is not specified).

Example I

The following example removes a single item from a resizable 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 displays myZoo[lion, bear]
myZoo.view("myZoo size: " + strVal(myZoo.size()))

endMethod

```

Example 2

The following example shows how to use **remove** to eliminate more than one element from a resizable 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

```

removeAllItems method

Array

Removes all occurrences of an array item.

Syntax

```
removeAllItems ( const value AnyType )
```

Description

removeAllItems deletes all occurrences of value from an array. Indexes of subsequent items are decreased by one.

Example

The following example shows how **removeAllItems** works with a resizable 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"

```

removeItem method

```
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
```

removeItem method

Array

Deletes a specified item from an array.

Syntax

```
removeItem ( const value AnyType )
```

Description

removeItem deletes the first occurrence of value from an array. Indexes of subsequent items are decreased by one.

Example

The following example uses **removeItem** to eliminate an item from a resizable 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
```

replaceItem method

Array

Overwrites an item in an array with another item.

Syntax

```
replaceItem ( const keyItem AnyType, const newItem AnyType )
```

Description

replaceItem searches an array for *keyItem*, and replaces the first occurrence of *keyItem* with *newItem* (provided that *newItem* is a valid array element). If *keyItem* is not found, *newItem* is not inserted.

Example

The following example replaces an item in a resizable array and displays the original 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
```

setSize method

Array

Specifies the size of an array.

Syntax

```
setSize ( const size LongInt )
```

Description

setSize saves space for *size* items in a resizable array. If **setSize** makes the array smaller, the array is truncated.

Example

The following example declares a resizable array in the variable declaration section and then uses **setSize** to initialize the size of the array to three elements. The code fills each element of the array and then executes **setSize** again to resize the array to two elements. Making the array smaller (shown in a dialog box) eliminates 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
```

size method

```
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
```

size method

Array

Returns the number of items in an array.

Syntax

```
size ( ) LongInt
```

Description

size returns the number of items in an array, even if one or more elements are blank.

Example

See the example for **setSize**.

view method

Array

Displays the contents of an array in a dialog box.

Syntax

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

Description

view displays the contents of an array in a modal dialog box. 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 box can not be changed interactively. For more information, see AnyType.

Example

The following example displays the contents of an array in a dialog box without a custom title and then with a custom title. 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

```

Binary type

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. You can refer to this variable in your code to move binary data back and forth between a disk file and a binary field in a table or from a disk file or a table to a method or procedure.

The Binary type includes several derived methods from the AnyType type.

Methods for the Binary type

AnyType	←	Binary
blank		clipboardErase
dataType		clipboardHasFormat
isAssigned		enumClipboardFormats
isBlank		readFromClipboard
isFixedType		readFromFile
		size
		writeToClipboard
		writeToFile

clipboardErase method

Binary

Clears the Windows Clipboard.

Syntax

```
clipboardErase ( )
```

Description

clipboardErase clears the Windows Clipboard on the user's system.

Example

See the example for **clipboardHasFormat**.

clipboardHasFormat procedure

Binary

Reports whether a format name is on the Windows Clipboard.

enumClipboardFormats method

Syntax

```
clipboardHasFormat ( const formatName String ) Logical
```

Description

clipboardHasFormat returns True if the format name *formatName* is on the Windows Clipboard on a user's system; otherwise, it returns False.

Example

In the following example, the **pushButton** method for a button named *clearClipboard* checks the Windows Clipboard for a Corel Form Object and if it is there, clears the Clipboard.

```
;btnClearClipboard::pushButton
method pushButton(var eventInfo Event)
  var
    b    Binary
  endVar

  if clipboardHasFormat("Corel Form Object") then
    b.clipBoardErase()
    message("Clipboard cleared")
  else
    message("Corel form object not on Clipboard")
  endIf
endMethod
```

enumClipboardFormats method

Binary

Creates an array listing the formats on the Windows Clipboard.

Syntax

```
enumClipboardFormats ( var formatNames Array[ ] String ) SmallInt
```

Description

enumClipboardFormats creates an array *formatNames* that lists the formats on the Windows Clipboard on the user's system. You must declare the array before you call this method.

Example

The following example writes the Clipboard format names to an array named *ar*, then displays *ar* in a view dialog box.

```
;btnShowClipboard :: pushButton
method pushButton(var eventInfo Event)
  var
    b    Binary
    ar    Array[] String
  endVar

  b.enumClipboardFormats( ar )
  ar.view("Formats in Windows Clipboard")
endmethod
```

readFromClipboard method

Binary

Reads a binary object from the Clipboard.

Syntax

```
readFromClipboard ( const clipboardFormat String ) Logical
```


Description

readFromClipboard reads a binary object *clipboardFormat* from the Clipboard. If the Clipboard contains a Binary object that can be copied to a Binary variable, **readFromClipboard** returns True. If the Clipboard is empty or does not contain a valid Binary object, **readFromClipboard** returns False.

Example

See the example for **writeToClipboard**.

readFromFile method**Binary**

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

Syntax

```
readFromFile ( const fileName String ) Logical
```

Description

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

Example

The following example creates a pop-up menu listing the SQL files stored in the private directory. When the user chooses a file from the menu, this code calls **readFromFile**. **readFromFile** reads the query, assigns it to an SQL variable, executes the query, and stores the results in a TCursor. The code then passes the TCursor to a custom procedure (assumed to be defined elsewhere) for additional processing.

```
method pushButton(var eventInfo Event)
var
    myAlias,
    aliasTableName,
    sqlFileName,
    sqlFileSpec      String
    aliasNamTC,
    answerTC          TCursor
    sqlPop            PopUpMenu
    db                Database
    sqlFS             FileSystem
    sqlFileAr         Array[] String
    sqlVar            SQL
endVar
; initialize variables
myAlias = "itchy"
aliasTableName = ":PRIV:aliasNam.db"
sqlFileSpec = ":PRIV:*.SQL"

enumAliasNames(aliasTableName) ; create a table of aliases

aliasNamTC.open(aliasTableName)
if aliasNamTC.locate("DBName", myAlias) then
    db.open(myAlias) ; use alias to get database handle to server
else
    msgStop("Stop",
        "The alias " + myAlias +
        " has not been defined.")
    return ; exit the method
endif
```

size method

```
; build a pop-up menu listing SQL files in the target directory
if sqlFS.findFirst(sqlFileSpec) then
  sqlFS.enumFileList(sqlFileSpec, sqlFileAr)
  sqlPop.addArray(sqlFileAr)
  sqlFileName = sqlPop.show() ; variable stores user's menu choice
  ; read and execute the SQL file chosen by the user
  sqlVar.readFromFile(sqlFileName)
  if sqlVar.executeSQL(db,answerTC) then
    doSomething(answerTC) ; call custom proc to process data
  else
    errorShow("readFromFile failed")
  endif
endif

else
  msgStop("File not found:", sqlFileSpec)
endif

endMethod
```

size method

Binary

Returns the number of bytes in a Binary variable.

Syntax

```
size ( ) LongInt
```

Description

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

Example

The following example tests the **size** of each Binary field in a table. If there's enough free disk space, the code writes the data to a disk file. Assume that SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, and 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
```

writeToClipboard method

Binary

Writes a binary object to the Clipboard.

Syntax

```
writeToClipboard ( const clipboardFormat String ) Logical
```

Description

writeToClipboard writes (copies) a binary object to the Windows Clipboard. Specify the Clipboard format to use with the parameter *clipboardFormat*. **writeToClipboard** returns True if successful; otherwise, it returns False.

Example

In the following example, a form contains two buttons. The button named *btnStoreClip* stores the native portion of the Windows Clipboard in a file called *NATIVE.CLP*. The second button, *btnRetrieveClip*, retrieves the clip from the file and writes it to the Clipboard.

The following code is attached to *btnStoreClip*.

```
;btnStoreClip :: pushButton
method pushButton(var eventInfo Event)
  var
    b   Binary
  endVar

  if not b.readFromClipboard("Native") then
    msgInfo("Instructions", "First copy something to Clipboard.")
  endIf

  b.writeToFile("Native.clp")
endmethod
```

The following code is attached to *btnRetrieveClip*.

```
;btnRetrieveClip :: pushButton
method pushButton(var eventInfo Event)
  var
    b   Binary
  endVar

  if not b.readFromFile("Native.clp") then
    beep()
    message("File does not exist")
  endIf

  b.writeToClipboard("Native")
endmethod
```

writeToFile method

Binary

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

Syntax

```
writeToFile ( const fileName String ) Logical
```

Description

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

writeToFile method

Example

The following example tests the size of each Binary field in a table. If there's enough free disk space, the code writes the data to a disk file. Assume that SOUNDS.DB is a Paradox table with the following structure: SoundName, A32; SoundData, and 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
```

Currency type

Currency values can range from ± 3.4E-308 to ± 1.1E308 (precise to 18 decimal places). The number of decimal places displayed depends on the user's Control Panel settings. However, the values in a table are stored up to 18 decimal places.

The following table lists the methods for Currency type, including several derived methods from the Number and AnyType types.

Methods for the Currency type

AnyType	←	Number	←	Currency
blank		abs		currency
dataType		acos		
isAssigned		asin		
isBlank		atan		
isFixedType		atan2		
view		ceil		
		cos		
		cosh		
		exp		
		floor		
		fraction		

fv
 ln
 log
 max
 min
 mod
 number
 numVal
 pmt
 pow
 pow10
 pv
 rand
 round
 sin
 sinh
 sqrt
 tan
 tanh
 truncate

currency procedure

currency

Converts a value's data type to Currency.

Syntax

```
currency ( const value AnyType ) Currency
```

Description

currency casts *value* as a Currency. The date, time and datetime types will return 0.00 when converted to currency in this way.

Example 1

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

```

; 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

```

currency procedure

Example 2

In the following example, the **pushButton** method for the *watchPrecision* button calculates a number using variables of the **Number** type, then performs the same calculation with the values cast as **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
```

Note

- Although the currency type will accept Anytype, not all data types will produce a valid currency value.

Database type

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 working directory. To work with tables stored in another database, declare a Database variable and use an **open** statement to create a handle to the database. You can also specify the full path to a table each time you wanted to use it, but code that uses Database variables is easier to maintain.

The following example demonstrates how to use **open** and an alias to specify which database to open:

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

Paradox now recognizes two databases: the default database and CustomerInfo. The custInfo variable is a *handle* to the *CustomerInfo* database and can be used 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 the CustomerInfo database), and you want to find out if these files are tables. The following example uses *custInfo* as a handle for the CustomerInfo database and tests ORDERS.DB:

```
var
    custInfo Database
endVar
addAlias("CustomerInfo", "Standard", "c:\\Corel\\Paradox\\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 custInfo 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, the following syntax gives you a handle for the default database, which you can pass to a custom method that requires a database handle.

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

Note

- The default database type refers to the WORK alias (your working directory).

Methods for the Database type

Database

beginTransaction	isAssigned	transactionActive
close	isSQLServer	
commitTransaction	isTable	
delete	open	
enumFamily	setMaxRows	
getMaxRows	rollBackTransaction	

beginTransaction method**Database**

Starts a transaction.

Syntax

```
beginTransaction ( [ const isoLevel String ] ) Logical
```

Description

beginTransaction starts a transaction on a database that supports transactions, such as Interbase, Microsoft SQL, and most other SQL databases.

The optional argument *isoLevel* specifies an isolation level to use when transactions are supported on SQL databases. If you do not specify an isolation level, the highest (most isolated) isolation level supported by the server is used. The following table lists values for *isoLevel* from lowest to highest isolation level.

isoLevel value	Description
DirtyRead	The transaction reads uncommitted changes made by other transactions.
ReadCommitted	Changes made by other transactions affect data read by the current transaction.
RepeatableRead	Data previously read in the current transaction is not affected by changes made by other transactions.

The **beginTransaction** method returns True if successful; otherwise, it returns False. While the transaction is active, statements that operate on tables associated with the specified database (except passthrough SQL statements) are included as part of the transaction. Only one transaction is allowed for each database.

Example

The following example processes a withdrawal of cash from an automatic teller machine. The call to **beginTransaction** starts a transaction consisting of three operations: debiting the customer's account, debiting the cash on hand, and dispensing cash to the customer. The result of each operation is stored in a dynamic array. When all of the operations are completed, this code checks each item in the DynArray and calls **commitTransaction** (if all items are True) or **rollbackTransaction** (if an item is False).

This example uses **beginTransaction**, **commitTransaction**, **rollbackTransaction**, **transactionActive**, **enumAliasNames**, and **getAliasProperty**.

```
method pushButton(var eventInfo Event)
var
    db                Database
    opResult          DynArray[] Logical
    Element           AnyType
    All_OK            Logical
    serverType,
    myAlias,
    custID            String
    aliasNamTC        TCursor
    xAmount           Currency
    xDate             Date
    xTime             Time
endVar

; initialize variables
myAlias = "ITCHY"
custID = "RHALL001"
```



```

xAmount = Currency(120.00)
xDate = today() ; returns current date
xTime = time() ; returns current time

; use alias to get database handle to server
if not db.open(myAlias) then
    errorShow("Could not open the database.")
    return ; exit the method
endif

if db.transactionActive() then
    db.commitTransaction() ; commit any previous transaction
endif

db.beginTransaction() ; begin a transaction

; execute the operations for this transaction
; debitAccount, debitCashOnHand, and dispenseCash
; are custom procs assumed to be defined elsewhere
; after calling debitAccount and debitCashOnHand, the code
; calls transactionActive to check the transaction status
; before calling dispenseCash

opResult["Debit customer account"] =
    debitAccount(custID, xAmount)
opResult["Debit cash on hand"] =
    debitCashOnHand(xAmount, xDate, xTime)

; the following if...then...else block is not required
; it's included to show one way to use transactionActive

if db.transactionActive() then ; make sure everything is OK
    msgInfo("Transaction Status", "In a Transaction")
else
    errorShow("NOT in a Transaction")
    return
endif

opResult["Dispense cash"] = dispenseCash(xAmount)

All_OK = True ; initialize to True

forEach element in opResult ; Check operation results
    if opResult[element] = False then
        All_OK = False
        quitLoop
    endif
endForEach

; inform user of transaction status
if All_OK then
    if db.commitTransaction() then
        msgInfo("Transaction Status", "Transaction committed.")
    else
        errorShow("Transaction NOT committed")
    endif
else
    if msgQuestion("Transaction failed",
        "View results?") = "Yes" then
        opResult.view("Operation results")
    endif
endif

```

close method

```
        if db.rollbackTransaction() then
            msgInfo("Transaction Status",
                "Transaction rolled back.")
        else
            errorShow("Transaction NOT rolled back.")
        endIf
    endIf
endMethod
```

close method

Database

Closes a database.

Syntax

```
close ( ) Logical
```

Description

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

Example

The following example 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
```

commitTransaction method

Database

Commits all changes within a transaction.

Syntax

```
commitTransaction ( ) Logical
```

Description

commitTransaction commits all changes made within a transaction on a database that supports transactions, such as Paradox, dBASE, and most SQL databases.

commitTransaction returns True if successful; otherwise, it returns False. This method does not check the results of the operations in the transaction; instead, you must evaluate the results and decide whether to commit the transaction or roll it back.

Example

See the **beginTransaction** example.

delete method/procedure

Database

Deletes a table from a database.

Syntax

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

Description

delete 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 the file extension is not standard or not supplied, you can use the optional argument *tableType* to specify the type of table to delete (e.g., "Paradox" or dBASE). If *tableType* is not specified or not standard, delete removes the Paradox table. If you use Syntax 2, you can use the argument *tableVar* to specify a Table variable. This method uses the name and type of table described by the Table variable, but does not use its database association. In either case, the deletion can't 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 the following 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.dbf"
myDb.open("megadata")
if isTable(tableName) then
    myDb.delete(tableName, "dBASE") ; removes OldTable.dbf from megadata
endif
endMethod
```

enumFamily method/procedure

Database

Lists the files in a table family.

Type

Database

Syntax

```
enumFamily ( var members DynArray[ ] String, const tableName String )
```

 Logical

getMaxRows method

Description

enumFamily lists the files in the table family of the table *tableName*. It assigns values to a dynamic array, or DynArray, named *members* that you pass as an argument. The value of *tableName* must include a file extension if the table name includes one. For example, if you specify "ORDERS" as the value, this method does not list the table family for ORDERS.DB; instead, **enumFamily** looks for a table named ORDERS.

The DynArray's indexes represent the full filenames of the table family members, and the corresponding value is one of the following strings:

Blobfile	SecondaryIndex2
Form	Table
Index	Unknown
Report	ValCheck
SecondaryIndex	

Example

The following example copies the family information from the *Orders* table to a dynamic array *dyn*. A **forEach** loop then displays each element of the family information in a dialog box.

```
;btnFamilyInfo :: pushButton
method pushButton(var eventInfo Event)
  var
    dyn      DynArray[] String
    sElement String
  endVar

  enumFamily(dyn, "ORDERS.DB")

  forEach sElement in dyn
    msgInfo(sElement, dyn[sElement])
  endForEach
  ; You could also do dyn.view().
endmethod
```

getMaxRows method

Database

Retrieves the setting of setMaxRows (the maximum number of rows that are returned from an SQL server).

Syntax

```
getMaxRows() LongInt
```

Description

getMaxRows retrieves the setting on the maximum number of rows that are returned from an SQL server in response to a query. Use **setMaxRows** to set the maximum number of returns.

Example

The following example puts a 1000 record limit on the query if the maximum is set to less than 1000.

```
var myQBE Query
endvar
if getMaxRows() 1000 then
  setMaxRows(1000)
endif
myQBE = Query
```

```

Customer      |Customer No |Name |
               | Check      |A..  |
endQuery

```

isAssigned method

Database

Reports whether a Database variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Example

In the following example, a form has been created with an unassigned field named *coRating* and a button named *showRating*. The code attached to *showRating*'s **pushButton** method uses **isAssigned** to determine whether the Database variable *db* is assigned. If a value has now been assigned to the variable *db*, a database alias is 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. The following code is 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:\\Core1\\Paradox\\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

```

isSQLServer method

Database

Reports whether a Database is opened on an SQL server.

Syntax

```
isSQLServer ( ) Logical
```

Description

isSQLServer returns True if the Database variable is open on an SQL server; otherwise, it returns False.

isTable method/procedure

Example

In the following example, a Database variable is opened on an alias. The code then determines if the Database variable points to an SQL server, and displays the results.

```
; showRating::pushButton
method pushButton(var eventInfo Event)
    var
        db Database
    endVar

    db.open(":fred:")

    if db.isSQLServer() then
        msgInfo(":FRED:", "Is on a SQL server.")
    else
        msgInfo(":FRED:", "Is not on a SQL server.")
    endIf
endMethod
```

isTable method/procedure

Database

Reports whether a database contains a table.

Syntax

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

Description

isTable returns True if the specified table is found in the database; otherwise, it returns False.

If you use Syntax 1, you can specify the table's name and type in the arguments *tableName* and *tableType*. If you use Syntax 2, you can specify a Table variable in *tableVar*. This method uses the table name and type described by the Table variable, and not the database association.

Example

The following example uses **isTable** to determine whether the Orders table exists in a given database. The 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
```

open method/procedure

Database

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
5. open ( [ const aliasName String, ] [ const ses Session, ]
        [ const parms DynArray ] ) Logical

```

Description

open opens a database. In Syntax 1, where no arguments are given, **open** opens the default database. In Syntax 2, you specify in *aliasName* a database to open in the current session. Syntax 3 opens the default database in the session specified in *ses*. Use Syntax 4 to open a specified database in a specified session. In Syntax 5, the *parms* argument represents a list of parameters and values to use when opening a database on an SQL server. The items in the parameter list correspond to the fields in the Alias Manager dialog box for a given alias. The items vary depending on the server you're connecting to; see your server documentation for more information.

open returns True if it opens the specified database; otherwise, it returns False.

Notes

- If you use Syntax 2, 4, or 5, *aliasName* must be a valid alias in the current session or the session. The colons around the alias name are optional.
- Syntaxes 3, 4, and 5 require that a valid session variable has been opened; the current session is assumed in Syntaxes 1 and 2.
- When you use Syntax 5, the settings in the *parms* dynamic array override previously set values, both in code and interactively. For example, if the OPEN MODE parameter was previously set to READ/WRITE, the following statement would set it to READ ONLY when you open the database.

```
dbParmsDA["OPEN MODE"] = "READ ONLY"
```

- When you use *parms* to specify parameters, the Alias Manager dialog box does not open.

Example

For the following example, the **pushButton** method for *thisButton* opens four databases in the current session.

```

; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    dDb, myDb, pDb, rDb Database
    dbParmsDA DynArray[] AnyType
    currSes Session
endVar

```

rollBackTransaction method

```
currSes.open()           ; get a handle to the current session

dDb.open()               ; associate dDb with the default database
myDb.open("custInfo")    ; associate myDb with the Custinfo database
                          ; (custInfo is an alias defined elsewhere)
pDb.open("PRIV")         ; associate pDb with the Private directory

; specify parameters for SQL database
dbParmsDA["OPEN MODE"] = "READ/WRITE"
dbParmsDA["Password"]  = "tycobb"

rDb.open("remote", currSes, dbParmsDA) ; (remote is an alias defined elsewhere)
endMethod
```

rollBackTransaction method

Database

Rolls back or undoes all changes within a transaction, on a server that supports transactions.

Syntax

```
rollbackTransaction ( ) Logical
```

Description

rollbackTransaction undoes the effects of all operations within a transaction. This method returns True if successful; otherwise, it returns False.

Example

See the **beginTransaction** example.

setMaxRows method

Database

Sets the maximum number of rows that can be retrieved by one query.

Syntax

```
setMaxRows ( const maxRows LongInt ) Logical
```

Description

setMaxRows sets the maximum number of rows that can be returned from an SQL server in response to a single query. The argument *maxRows* is a long integer that specifies the maximum number of rows returned. **setMaxRows** returns True if the maximum number of rows specified by *maxRows* is successfully set; otherwise, it returns False.

setMaxRows resemble the Borland Database Engine (BDE) configuration option MAX ROWS. MAX ROWS is set in the BDE Configuration file's DB OPEN section and sets the maximum number of rows that the SQL driver will attempt to fetch for any single SQL statement that is sent to the server. If a request exceeds the maximum specified by MAX ROWS, Paradox generates a DBIERR_ROWFETCHLIMIT error.

Notes

- The maximum specified with the **setMaxRows** method can exceed that specified by the MAX ROWS BDE configuration option.
- If no **setMaxRows** method is issued or if the *maxRows* argument is set to -1, Paradox imposes no limit on rows. If present, the BDE MAX ROWS limit is imposed.

Example

The following example puts a 1000 record limit on the query if the maximum is set to less than 1000. This example assumes the database is open.

```
var
    myQBE Query
    myDatabase Database
endvar
myDatabase.open("Work")
if mydatabase.getmaxrows() then
else
    mydatabase.setmaxrows(1000)
endif
myQBE = Query
    Customer      |Customer No |Name |
                  | Check      |A.. |
endQuery
```

transactionActive method**Database**

Reports whether a transaction is active in the specified database.

Syntax

```
transactionActive ( ) Logical
```

Description

transactionActive reports whether a transaction is active in the specified database. Because Paradox allows only one active transaction for each database, call **transactionActive** to determine whether a transaction is active before beginning a transaction.

Example

See the **beginTransaction** example.

DataTransfer type

The DataTransfer type contains methods and procedures that create, delete, import, and export data.

Methods for the DataTransfer type**DataTransfer**

appendASCIIFix	dlgImportASCIIFix
AppendASCIIVar	dlgImportASCIIVar
dlgExport	dlgImportSpreadsheet
dlgImport	dlgImportTable
empty	getDestSeparator
enumSourcePageList	getDestType
enumSourceRangeList	getKeyviol
getSourceFieldNamesFromFirst	getProblems
getSourceName	getSourceCharSet
getSourceRange	getSourceDelimitedFields
getSourceSeparator	getSourceDelimiter
getSourceType	setDest

appendASCIIIFix procedure

importASCIIIFix	setDestCharSet
importASCIIVar	setDestDelimitedFields
importSpreadsheet	setDestDelimiter
loadDestSpec	setDestFieldNamesFromFirst
loadSourceSpec	setDestSeparator
setAppend	setKeyviol
exportASCIIIFix	setProblems
exportASCIIVar	setSource
exportParadoxDOS	setSourceCharSet
exportSpreadsheet	setSourceDelimitedFields
getAppend	setSourceDelimiter
getDestCharSet	setSourceFieldNamesFromFirst
getDestDelimitedFields	setSourceRange
getDestDelimiter	setSourceSeparator
getDestFieldNamesFromFirst	transferData
getDestName	

appendASCIIIFix procedure

DataTransfer

Adds fixed format ASCII data from a file to a table.

Syntax

```
appendASCIIIFix ( const fileName String, const tableName String, const specTableName
String [ , const ANSI Logical ] ) Logical
```

Description

appendASCIIIFix adds data from the fixed format ASCII file specified by *fileName* to the table specified by *tableName*. This method uses the layout specified in *specTableName*.

The following table illustrates the structure of the file specified with *specTableName* :

Field name	Type & size	Description
Field Name	A 25	Name of the field to import
Type	A 4	Field type to import. The Type must be a valid Paradox or dBASE field specification.
Start	S	Column number where the field value begins
Length	S	Field size

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example imports ASCII fixed text to Paradox (short form):

```
appendASCIIIFix("NewRecords.txt", "TimeCards.db", "ImpSpec.db")
```

appendASCIIVar procedure**DataTransfer**

Adds delimited ASCII data from a file to a table.

Syntax

appendASCIIVar (const *fileName* String, const *tableName* String [, const *separator* String, const *delimiter* String, const *allFieldsDelimited* Logical, const *ANSI* Logical]) Logical

Description

appendASCIIVar appends data from the delimited ASCII file specified by *fileName* to the table specified by *tableName*. **appendASCIIVar** uses the options specified by *separator*, *delimiter*, *allFieldsDelimited*, and *ANSI*.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example imports ASCII Delimited Text to Paradox:

```
appendASCIIVar("NewRecords.txt", "TimeCards.db")
```

dlgExport procedure**DataTransfer**

Displays the Export <table> as: dialog box.

Syntax

dlgExport (const *tableName* String [, const *fileName* String])

Description

dlgExport displays the Export <table> as: dialog box with the specified table and file names already filled in. *tableName* specifies the name of the table to export. The type of export file is determined by the extension of the *fileName*.

ObjectPAL code suspends execution until the user closes the dialog box. ObjectPAL has no control over this dialog box once it is displayed; it is up to the user to close the dialog box.

Example

The following example displays the Export As dialog box for the ORDERS.DB table.

```
method pushButton ( var eventInfo Event )
    var
        tableName String
    endVar

    tableName = "orders.db"
    ; invoke the Export<tablename> As dialog box
    dlgExport ( tableName )
endMethod
```

dlgImport procedure**DataTransfer**

Displays the Import Data dialog box.

Syntax

dlgImport (const *fileName* String [, const *tableName* String])

dlgImportASCIIFix procedure

Description

dlgImport displays the Import Data dialog box with the specified file and table names displayed as the default. Paradox opens text files and reads first few lines to determine whether the file contains delimited or fixed-length text. Text files are files with the *.TXT extension or with unknown extensions.

Example

The following example displays the Import Data dialog box. The target table name defaults to the name of the source file, with a .DB extension. The target file type is Paradox, unless another type has been specified for the table.

```
method pushButton(var eventInfo Event)
    ;the following line displays the Import Data dialog box
    dlgImport("Customer.txt")
endmethod
```

dlgImportASCIIFix procedure

DataTransfer

Displays the Import Data dialog box.

Syntax

```
dlgImportASCIIFix ( const fileName String )
```

Description

dlbImportASCIIFix displays the Import Data dialog box with the specified *fileName* displayed as the default, and the import file type set to fixed-length ASCII. *fileName* specifies the name of the source file and the target table for the imported data. If you specify a file extension, Paradox uses it to locate the appropriate file.

The target table's extension depends on its table type. The default type for Paradox is .DB; for dBASE tables the default type is .DBF. Dates and numbers are formatted according to your settings in the Windows Control Panel.

ObjectPAL code suspends execution until the user closes the Import Data dialog box.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example displays the Import Data dialog box and imports data from the ORDERS.TXT text file to the ORDERS.DB table:

```
method pushButton ( var eventInfo Event )
    var
        fileName String
    endVar

    fileName = "orders.txt"

    ; invoke the Import Data dialog box
    ; by default, Paradox will use ORDERS.TXT as the source file
    ; and ORDERS.DB as the target table
    dlgImportASCIIFix ( fileName )
endMethod
```

dlgImportASCIIVar procedure**DataTransfer**

Displays the Import Data dialog box.

Syntax

```
dlgImportASCIIVar ( const fileName String )
```

Description

dlgImportASCIIVar displays the Import Data dialog box with the specified *fileName* displayed as the default, and the import file type set to delimited ASCII text. *fileName* specifies the name of the source file and the target table for the imported data.

The target table's extension depends on its table type. The default type for Paradox is .DB; for dBASE tables the default type is .DBF. Dates and numbers are formatted according to your settings in the Windows Control Panel.

ObjectPAL code suspends execution until the user closes the Import Data dialog box.

The default settings include: fields separated by commas; fields delimited by quotes; only text fields delimited; and the OEM character set used.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example displays the Import Data dialog box and imports data from the ORDERS.TXT text file to the ORDERS.DB table:

```
method pushButton ( var eventInfo Event )
    var
        fileName String
    endVar

    fileName = "orders.txt"

    ; invoke the Import Data dialog box.
    ; by default, Paradox will use ORDERS.TXT as the source file
    ; and ORDERS.DB as the target table.
    dlgImportASCIIVar ( fileName )
endMethod
```

dlgImportSpreadsheet procedure**DataTransfer**

Displays the Import Data dialog box.

Syntax

```
dlgImportSpreadsheet ( const fileName String )
```

Description

dlgImportSpreadsheet displays the Import Data dialog box with the specified *fileName* displayed as the default. *fileName* specifies the name of the source file, its spreadsheet type, and the name of the target table.

The target table's file extension depends on its table type. The default type for Paradox is .DB; for dBASE tables the default type is .DBF. Dates and numbers are formatted according to your settings in the Windows Control Panel.

Paradox uses the file extensions that you specify to identify the spreadsheet type of the source file. The following table displays the file extensions and their spreadsheet types:

dlgImportTable procedure

Extension	Format
WB1, WB2, WB3	Quattro Pro Win
WQI	Quattro Pro DOS
WKQ	Quattro
WKI	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

ObjectPAL code suspends execution until the user closes the Import Data dialog box.

The default settings specify that the From cell is the first cell of the spreadsheet's first page. The To cell is the last cell of the spreadsheet's last page, and the Use First Row Of Data As Field Names check box is enabled.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example instructs the Import Data dialog box to import data from a Quattro Pro for Windows spreadsheet (ORDERS.WB1) to a Paradox table (ORDERS.DB):

```
method pushButton ( var eventInfo Event )
    var
        fileName String
    endVar

    fileName = "orders.wb1"

    ; invoke the Import Data dialog box
    ; by default, Paradox will use ORDERS.WB1 as the source file
    ; and ORDERS.DB as the target table
    dlgImportSpreadsheet ( fileName )
endMethod
```

dlgImportTable procedure

DataTransfer

Displays the Import Data dialog box.

Syntax

```
dlgImportTable ( const tableName String )
```

Description

dlgImportTable displays the Import Data dialog box with the specified *tableName* displayed as the import source.

Example

The following example displays the Import Data dialog box and imports data from the dBASE table ORDERS.DBF to the Paradox table ORDERS.DB.

```
method pushButton ( var eventInfo Event )
    var
        tblName String
    endVar
```

```
tblName = "orders.dbf"

; invoke the Import Data dialog box
; by default, Paradox will use ORDERS.DBF as the source file
; and ORDERS.DB as the target table
dlgImportTable ( tblName )
endMethod
```

empty method

DataTransfer

Deletes the data from a structure.

Syntax

```
empty ( )
```

Description

empty re-initializes a structure by deleting its data while leaving its form intact. **empty** can initialize Mail variable structures and tables but cannot initialize forms, databases, or reports.

Example

The following example specifies a DataTransfer data type. This structure is used with the transferData method. This example assumes that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method cmTransfer() is within the scope of the variable (dt).

```
method cmTransfer() ;this example completes a DataTransfer

    dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
                                           ; to CUSTOMER.TXT
    dt.setSourceSeparator("/") ; specifies the forward slash "/" character
                               ; to separate each field
    dt.setSourceDelimiter("'") ; specifies the single quote to surround
                               ; the fields
    dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
                                                ; quote (delimiter) surrounds
                                                ; only text fields of the
                                                ; source file
    dt.setSourceCharSet(DTANSI) ; specifies that the character set used
                               ; when creating the source file
                               ; was the ANSI character set

    dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
                                           ; row of the source file as
                                           ; field names
    dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
    dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
                        ; any problems importing the source file
    dt.transferData() ; executes the data transfer. In this case it
                    ; imports the CUSTOMER.TXT file as NEWCUST.DB.
    dt.empty() ; empties the dt variable structure to set it up for
              ; a new transfer.

endmethod
```

enumSourcePageList method

DataTransfer

Copies the list of spreadsheet pages in a string array.

enumSourceRangeList method

Syntax

```
enumSourcePageList ( var pages Array[] String )
```

Description

enumSourcePageList compiles a list of pages and copies it into a string array called *pages*. This method requires you to set filenames and types to an existing spreadsheet. **enumSourcePageList** only applies when the source file is a spreadsheet.

Example

The following example copies the pages of the LEDGER.WB3 spreadsheet into an array and displays the results:

```
method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
    arPage  Array[] String
  endVar
  dt.setSource("ledger.wb3")
  dt.enumSourcePageList(arPage)
  arPage.view()
endMethod
```

enumSourceRangeList method

DataTransfer

Compiles a list of named ranges into a string array.

Syntax

```
enumSourceRangeList ( var ranges Array[] String )
```

Description

enumSourceRangeList copies the list of named ranges into a string array named *ranges*. This method requires you to set filenames and types to an existing spreadsheet. **enumSourceRangeList** only applies when the source file is a spreadsheet.

Example

The following example compiles a list of the pages of the LEDGER.WB3 spreadsheet, and displays the list.

```
method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
    arRange Array[] String
  endVar
  dt.setSource("ledger.wb3")
  dt.enumSourceRangeList(arRange)
  arRange.view()
endMethod
```

exportASCIIFix procedure

DataTransfer

Exports data from the specified table to an ASCII file in which all of the fields are the same length.

Syntax

```
exportASCIIFix ( const tableName String, const fileName String, const specTableName
String [ , const ANSI Logical ] ) Logical
```


Description

exportASCIIIFix exports data from the specified table to an ASCII file in which all of the record's fields are the same length. This method duplicates the function of the Export Data dialog box.

tableName specifies the source table and *fileName* specifies the target file. If the target file does not exist, this procedure creates it using the layout specified *specTableName*. *specTableName* is the name of a table that specifies the layout for the imported data. The following table illustrates the structure of the file specified with *specTableName* :

Field name	Type & size	Description
Field Name	A 25	Name of the field to import
Type	A 4	Field type to import. The Type must be a valid Paradox or dBASE field specification.
Start	S	Column number where the field value begins
Length	S	Field size

exportASCIIIFix can use the same *specTableName* as **importASCIIIFix**. For export operations, the table type determines the field type. More recent versions of Paradox will recognize tables made with versions 5.0 and earlier, but the reverse is not true.

For each field you export, *specTableName* contains a Start position (the column where the field value begins) and a Length (the number of characters in the field). *specTableName* operates like EXPORT.DB, which is created when you use the Export Data dialog box to export a table interactively.

ANSI (optional) specifies whether to use the ANSI or OEM character set. Set *ANSI* to True to use the ANSI character set, or to False to use the OEM character set.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example exports data from the ORDERS.DB table to the ORDERS.TXT text file. The code then reads the export format from the ORDEREXP.DB table and exports the data using the ANSI character set.

```
method pushButton ( var eventInfo Event )
    exportASCIIIFix ( "orders.db", "orders.txt", "orderexp.db", True )
endMethod
```

exportASCIIVar procedure**DataTransfer**

Exports data from a specified table to a delimited ASCII file. A delimited ASCII text files is one of variable fixed length.

Syntax

```
exportASCIIVar ( const tableName String, const fileName String [ , const separator
String, const delimiter String, const allFieldsDelimited Logical, const ANSI Logical ]
) Logical
```

Description

exportASCIIVar exports data from a table to a delimited ASCII file. If the file does not exist, exportASCIIVar creates it. This method duplicates the function of the Export Data dialog box.

exportParadoxDOS procedure

tableName specifies the source table, and *fileName* specifies the target file. *separator* (optional) specifies the character that surrounds field values in the target file. You can choose a comma or any other single character, including special characters. *delimiter* (optional) specifies the character that defines the limits of field values in the target. Leave the delimiter string empty if you do not want to define limits. The *allFieldsDelimited* (optional) string is marked True if data from all field types is delimited, and False if data from only text, alphanumeric, or character field types is delimited.

Note

Paradox cannot export memo (Paradox or dBASE), formatted memo, graphic, OLE, or binary fields to delimited text.

ANSI (optional) specifies whether to use the ANSI or OEM character set. Set *ANSI* to True to use the ANSI character set, or to False to use the OEM character set.

The following table displays the default settings for optional arguments:

- separator “,” (comma)
- delimiter “\” (double quote)
- allFieldsDelimited False
- ANSI False

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example exports data from the ORDERS.DB table to the ORDERS.TXT text file. In this example, tabs delimit field values, percent signs enclose each value, only text fields are delimited, and the ANSI character set is used.

```
method pushButton ( var eventInfo Event )
    exportASCIIVar ( "orders.db", "orders.txt", "\t", "%", False, True )
endMethod
```

The following code exports Paradox to ASCII Delimited Text (medium form):

```
var
    dt DataTransfer
endVar

dt.setSource("TimeCards.db")
dt.setDest("Records.txt", DTAsciiVar)
dt.TransferData()
```

The following code exports Paradox to ASCII Delimited Text (short form):

```
ExportASCIIVar("TimeCards.db", "NewRecords.txt")
```

exportParadoxDOS procedure

DataTransfer

Exports data from a Paradox for Windows or a dBASE table to a Level 4 Paradox for DOS table.

Syntax

```
exportParadoxDOS ( const tableName String, const fileName String ) Logical
```

Description

exportParadoxDOS exports data from a Paradox for Windows or a dBASE table to a Level 4 Paradox for DOS table. This method duplicates the function of the Table Export dialog box.

Note

exportParadoxDOS cannot export Bytes (type Y) fields because they are excluded from the destination file. This method does not export OLE and Binary fields when you export a dBASE table to Paradox for DOS format.

tableName specifies the source table and *fileName* specifies the target file. If you include a file extension with the filename, it must be *.DB.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example exports data from a dBASE table named ORDERS.DBF to a Paradox for DOS table named ORDERS.DB:

```
method pushButton ( var eventInfo Event )
    if not exportParadoxDOS ( "orders.dbf", "orders" ) then
        errorShow ( "Export to Paradox DOS failed." )
    endif
endMethod
```

exportSpreadsheet procedure**DataTransfer**

Exports data from a table to a spreadsheet file.

Syntax

```
exportSpreadsheet ( const tableName String, const fileName String [ , const
makeRowHeaders Logical ] ) Logical
```

Description

exportSpreadsheet exports the data from a table to a spreadsheet file, duplicating the function of the Export Data dialog box. If the spreadsheet file does not exist, this method creates it. The spreadsheet type is determined by the file extension. When you export data to a spreadsheet, Paradox converts each record to a row and each field to a column. If a value is wider than the column, the full value is partially hidden.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

If a date in the original table is beyond the acceptable range in the spreadsheet, the date is exported as an ERROR.

tableName specifies the source table and *fileName* specifies the target file. *makeRowHeaders* (optional) specifies whether the table's column headers correspond to the spreadsheet's rows. The *makeRowHeaders* string returns True (default) if column headers are used as labels, and False if they are not.

Note

The file extension in *fileName* specifies the format of the spreadsheet file. The following table displays file extensions and their spreadsheet formats:

getAppend method

Extension	Format
WB1, WB2, WB3	Quattro Pro Win
WQ1	Quattro Pro DOS
WKQ	Quattro
WK1	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

Example

The following example exports data from the ORDERS.DB table to a Quattro Pro for Windows file. The table's field names are used as labels in the spreadsheet file.

```
method pushButton ( var eventInfo Event )  
    exportSpreadsheet ( "orders.db", "orders.wb1", True )  
endMethod
```

getAppend method

DataTransfer

Retrieves the True or False value set by setAppend.

Syntax

```
getAppend ( ) Logical
```

Description

getAppend retrieves the True or False value set by **setAppend**. **getAppend** applies only when the destination is a table.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification, and then call the transferData method.

```
var  
    dt DataTransfer  
endVar  
dt.SetSource ( "MYFILE.TXT" )  
if dt.getSourceType ( ) = DTASCIIIFixed Then  
    dt.loadDestSpec ( "SpecTable" )  
endif  
dt.setDest ( "Existing Data.db" )  
if not dt.getAppend ( ) then  
    dt.setAppend ( True )  
endif  
dt.transferData ( )
```

getDestCharSet method

DataTransfer

Retrieves the value set by setDestCharSet.

Syntax

```
getDestCharSet ( ) SmallInt
```

Description

getSourceCharSet retrieves the value set by **setDestCharSet**. The value is one of the two **DataTransferCharset** constants: **dtOEM** or **dtANSI**. **getSourceCharSet** applies only when the source or destination is a fixed or delimited ASCII text file.

Example

The following example uses the **transferData** method to export **ORDERS.DB** to **ORDINFO.TXT**. It uses **setDestCharSet** to specify the use of the ANSI character set. To specify the use of the OEM character set, use the **DT OEM** constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
    var
        dt      DataTransfer
    endVar
    dt.setDest("ordinfo.txt", DTASCIIVar)
    dt.setSource("orders.db")

    ;Specify the single quote (') to surround the fields.
    ;The delimited fields will be text fields only.
    dt.setDestDelimiter("'")
    dt.setDestDelimitedFields(DTDelimJustText)

    ;Specify the tab character to separate the fields.
    dt.setDestSeparator("\t")

    ;Set the first row of the ORDINFO.TXT to be the field names
    dt.setDestFieldNamesFromFirst(True)

    ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
    ;character set.
    if dt.getDestCharSet()= dtOEM then
        dt.setDestCharSet(DTAnsi)
    endif
    ;run Export
    dt.transferData()
endMethod
```

getDestDelimitedFields method**DataTransfer**

Retrieves the value set by **setDestDelimitedFields**.

Syntax

```
getDestDelimitedFields ( ) SmallInt
```

Description

getDestDelimitedFields retrieves the value set by the **setDestDelimitedFields** method. The value is one of the two **DataTransferDelimitCode** constants: **DtDelimAllFields** or **DtDelimJustText**.

getDestDelimitedFields only applies when the destination is a delimited ASCII text file.

Example

The following example exports the **ORDERS.DB** table into an ASCII delimited text file. The single quote character is specified as a delimiter.

```
method pushButton(var eventInfo Event)
    var
        dt      DataTransfer
    endVar
    dt.setDest("ordinfo.txt", DTASCIIVar)
```

getDestDelimiter method

```
dt.setSource("orders.db")

;Specify the single quote (') to surround the fields.
;The delimited fields will be text fields only.
msgInfo("Info", "current delimiter is "+dt.getDestDelimiter())
dt.setDestDelimiter("")
dt.setDestDelimitedFields(DTDelimJustText)

;Specify the tab character to separate the fields.
dt.setDestSeparator("\t")

;Set the first row of the ORDINFO.TXT to be the field names
dt.setDestFieldNamesFromFirst(True)

;Set the character set of the destination file ORDINFO.TXT to be the ANSI
;character set.
dt.setDestCharSet(DTAnsi)
;run Export
dt.transferData()
endMethod
```

getDestDelimiter method

DataTransfer

Retrieves the value set by setDestDelimiter.

Syntax

```
getDestDelimiter ( ) String
```

Description

getDestDelimiter retrieves the value set by the **setDestDelimiter** method. **getDestDelimiter** only applies when the destination is a delimited ASCII text file.

Example

The following example uses the transferData method to export ORDERS.DB to ORDINFO.TXT. It uses setDestDelimitedFields to delimit surrounding text fields only. To delimit all fields, use the DTDelimAllFields constant with setDestDelimitedFields.

```
method pushButton(var eventInfo Event)
var
    dt      DataTransfer
endVar
dt.setDest("ordinfo.txt", DTASCIIVar)
dt.setSource("orders.db")

;Specify the single quote (') to surround the fields.
;The delimited fields will be text fields only.
dt.setDestDelimiter("")
dt.setDestDelimitedFields(DTDelimJustText)

;Specify the tab character to separate the fields.
dt.setDestSeparator("\t")

;Set the first row of the ORDINFO.TXT to be the field names
dt.setDestFieldNamesFromFirst(True)

;Set the character set of the destination file ORDINFO.TXT to be the ANSI
;character set.
```

```

dt.setDestCharSet(DTAnsi)

;run Export
dt.transferData()
endMethod

```

getDestFieldNamesFromList method

DataTransfer

Retrieves the True or False value set by setDestFieldNamesFromFirst.

Syntax

```
getDestFieldNamesFromFirst ( ) Logical
```

Description

getDestFieldNamesFromFirst retrieves the True or False value set by **setDestFieldNamesFromFirst**. **getDestFieldNamesFromFirst** only applies when the source file is a spreadsheet or a delimited text file.

Example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. The **setDestFieldNamesFromFirst** is used to create the first row of the text file with field names.

```

method pushButton(var eventInfo Event)
var
    dt      DataTransfer
endVar
dt.setDest("ordinfo.txt", DTASCIIVar)
dt.setSource("orders.db")

;Specify the single quote (') to surround the fields.
;The delimited fields will be text fields only.
dt.setDestDelimiter("'")
dt.setDestDelimitedFields(DTDelimJustText)

;Specify the tab character to separate the fields.
dt.setDestSeparator("\t")

If dt.getDestFieldNamesFromFirst() Then
    msgInfo("Info", "SetDestFieldNamesFromFirst is On")
else
    msgInfo("Info", "Setting DestFieldNamesFrom First to On")
endif
;Set the first row of the ORDINFO.TXT to be the field names
dt.setDestFieldNamesFromFirst(True)

;Set the character set of the destination file ORDINFO.TXT to be the ANSI
;character set.
dt.setDestCharSet(DTAnsi)

;run Export
dt.transferData()
endMethod

```

getDestName method

DataTransfer

Retrieves the destination filename.

getDestSeparator method

Syntax

```
getDestName ( ) String
```

Description

getDestName retrieves the destination filename.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Export to text

```
var
    dt DataTransfer
endVar
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "The current source is " + dt.getSourceName())
dt.SetDest ( "NEWFILE.TXT",dtASCIIVar)
msgInfo("Info", "The current destination is " + dt.getDestName())
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

getDestSeparator method

DataTransfer

Retrieves the value set by setDestSeparator.

Syntax

```
getDestSeparator ( ) String
```

Description

getDestSeparator retrieves the value set by the **setDestSeparator** method. **getDestSeparator** only applies when the destination is a delimited ASCII text file.

Example

The following examples specify a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from spreadsheet

```
var
    dt DataTransfer
endVar
; Source file MYFILE.WB2 is a Quattro Pro Windows spreadsheet

dt.setSource ( "MYFILE.WB2" )
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator())
; Destination file is a Paradox table NEWFILE.DB

dt.SetDest ( "NEWFILE.DB" )
msgInfo("Info", "The current destination separator is " + dt.getDestSeparator())
dt.setProblems ( True )
dt.transferData ( )
```

Import from text

```
var
    dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator())
dt.setSource ( "SRCFILE.TXT")
```



```

MsgInfo("Info", "The current destination separator is " + dt.getDestSeparator())
dt.SetDest ( "NEWFILE.db" )
if dt.getSourceType ( ) = DTASCIIIFixed Then
    dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setAppend ( True )
dt.transferData ( )

```

Export to text

```

var
    dt DataTransfer
endVar
MsgInfo("Info", "The current source separator is " + dt.getSourceSeparator())
dt.setSource ( "CUSTOMER.db" )
MsgInfo("Info", "The current destination separator is " + dt.getDestSeparator())
dt.SetDest ( "NEWFILE.TXT",dtASCIIIFixed )
dt.setDestSeparator ( ";" )
dt.transferData ( )

```

getDestType method**DataTransfer**

Retrieves the destination file type constant.

Syntax

```
getDestType ( ) SmallInt
```

Description

getDestType retrieves the destination file type constant. The file type constant is one of the **DataTransferFileType** constants.

Example

The following example specifies a **DataTransfer** data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from spreadsheet

```

var
    dt DataTransfer
endVar
MsgInfo("Info", "the current dest type is " + string(dt.getDestType()))
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )

```

Import from text

```

var
    dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIIFixed Then
    dt.loadDestSpec ( "SpecTable" )
EndIf
MsgInfo("Info", "the current dest type is " + string(dt.getDestType()))
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )

```

getKeyviol method

Export to text

```
var
    dt DataTransfer
endVar
dt.setSource ( "ANSWER.db" )
msgInfo("Info", "the current dest type is " + string(dt.getDestType()))
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

getKeyviol method

DataTransfer

Retrieves the True or False value set by setKeyviol.

Syntax

```
getKeyviol ( [ const tableName String, var count LongInt ] ) Logical
```

Description

getKeyviol retrieves the True or False value set by the **setKeyviol** method. The argument *tableName* is the name of the Key Violations table, and *count* is the number of key violations in the table.

getKeyviol method applies only when the destination is a table.

Example

The following example retrieves the key violations from the kvTbl file.

```
method pushButton(var eventInfo Event)
    var
        dt DataTransfer
        kvTbl, probTbl String
        kvNum, probNum Longint
    endVar
    dt.setSource("MYFILE.TXT")
    dt.LoadSourceSpec("SPECFILE.DB")
    dt.setDest("MYFILE.DB")
    if isTable("MYFILE.DB ") then
        dt.setAppend(True)
    endIf
    if msgQuestion("Import Option",
        "Would you like to produce auxilliary tables?") = "Yes" then
        dt.setKeyviol(True)
        dt.setProblems(True)
    endIf
    dt.transferData()
    if dt.getKeyviol(kvTbl, kvNum) then
        msgInfo("Import Status",
            "# Key violations = "+string(kvNum)+
            "\nKeyviol table name = " + kvTbl)
    endIf
    if dt.getProblems(probTbl, probNum) then
        msgInfo("Import Status",
            "# Record errors = "+string(probNum) +
            "\nProblem table name = " + probTbl)
    endIf
endMethod
```

getProblems method

DataTransfer

Retrieves the True or False value set by setProblems.

Syntax

```
getProblems ( [ var tableName String, var count LongInt ] ) Logical
```

Description

getProblems retrieves the True or False value set by the **setProblems** method. The *tableName* argument specifies the name of the problems table, and count specifies number of problems.

getProblems applies only when the destination is a table.

Example

The following example retrieves the problems from the probTbl file.

```
method pushButton(var eventInfo Event)
  var
    dt                DataTransfer
    kvTbl, probTbl    String
    kvNum, probNum    Longint
  endVar
  dt.setSource("MYFILE.TXT")
  dt.LoadSourceSpec("SPECFILE.DB")
  dt.setDest("MYFILE.DB")
  if isTable("MYFILE.DB ") then
    dt.setAppend(True)
  endIf
  if msgQuestion("Import Option",
    "Would you like to produce auxilliary tables?") = "Yes" then
    dt.setKeyviol(True)
    dt.setProblems(True)
  endIf
  dt.transferData()
  if dt.getKeyviol(kvTbl, kvNum) then
    msgInfo("Import Status",
      "# Key violations = "+string(kvNum)+
      "\nKeyviol table name = " + kvTbl)
  endIf
  if dt.getProblems(probTbl, probNum) then
    msgInfo("Import Status",
      "# Record errors = "+string(probNum) +
      "\nProblem table name = " + probTbl)
  endIf
endMethod
```

getSourceCharSet method**DataTransfer**

Retrieves the value set by setSourceCharSet.

Syntax

```
getSourceCharSet ( ) SmallInt
```

Description

getSourceCharSet retrieves the value set by **setSourceCharSet**. The value is one of the two DataTransferCharset constants: dtOEM or dtANSI. **getSourceCharSet** applies only when the source is a fixed or delimited ASCII text file.

Example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **getSourceCharSet** to determine the source file's character set and **setDestCharSet** to set the

getSourceDelimitedFields method

destination file to the same character set. To specify the OEM character set, use the DTOEM constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
    var
        dt          DataTransfer
        stChrSt String
    endVar
    dt.setDest("ordinfo.txt", DTASCIIVar)
    dt.setSource("orders.db")

    ;Specify the single quote (') to surround the fields.
    ;The delimited fields will be text fields only.
    dt.setDestDelimiter("'")
    dt.setDestDelimitedFields(DTDelimJustText)

    ;Specify the tab character to separate the fields.
    dt.setDestSeparator("\t")

    ;Set the first row of the ORDINFO.TXT to be the field names
    dt.setDestFieldNamesFromFirst(True)

    ; set the destination character set to be the same as the source.
    if dt.getSourceCharSet() = dtAnsi then
        dt.setDestCharSet(DTAnsi)
        stChrSt = "ANSI"
    else
        dt.setDestCharSet(DTOEM)
        stChrSt = "OEM"
    endif
    ; since the result of getSourceCharSet is a SmallInt, convert the result
    ; to a string which represents ANSI or OEM
    msgInfo("Info", "the character set is: " + stChrSt)
    ;run Export
    dt.transferData()
endMethod
```

getSourceDelimitedFields method

DataTransfer

Retrieves the value set by setSourceDelimitedFields.

Syntax

```
getSourceDelimitedFields ( ) SmallInt
```

Description

getSourceDelimitedFields retrieves the value set by the **setSourceDelimitedFields** method. The value is one of the two DataTransferDelimitCode constants: DtDelimAllFields or DtDelimJustText. **getSourceDelimitedFields** only applies when the source is a delimited ASCII text file.

Example

The following example uses **getSourceDelimitedFields** to determine which fields are delimited.

```
method pushButton(var eventInfo Event)
    var
        dt          DataTransfer
    endVar
    dt.setSource("iesimpld.txt")
```

```

;The following lines check to see what Paradox determined as the type of
;of the source file. If it is delimited, Paradox determines the separator,
;delimiter and which fields are delimited.
switch
  case dt.getSourceType() = DTASCIIVar :
    fldType = "Delimited"
    fldDelimiter = dt.getSourceDelimiter()
    if dt.getSourceDelimitedFields() = DTDelimAllFields then
      fldDelimitedFields = "All"
    else
      fldDelimitedFields = "Text"
    endIF
    fldSeparator = dt.getSourceSeparator()
  case dt.getSourceType() = DTASCIIFixed :
    fldType = "Fixed"
  otherwise :
    msgInfo("Hello","File missing or not text.")
endSwitch
endMethod

```

getSourceDelimiter method

DataTransfer

Retrieves the value set by setSourceDelimiter.

Syntax

```
getSourceDelimiter ( ) String
```

Description

getSourceDelimiter retrieves the value set by the **setSourceDelimiter** method.

getSourceDelimiter only applies when the source is a delimited ASCII text file.

Example

The following example uses **getSourceDelimiter** to display the delimiter used in the source file. This confirms that the delimiter is set correctly and allows you to specify a new delimiter if necessary.

```

method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
  endVar
  dt.setSource("iesimpld.txt")

```

```

;The following lines check to see what Paradox determined as the type of
;of the source file. If it is delimited, Paradox determines the separator,
;delimiter and which fields are delimited.
switch
  case dt.getSourceType() = DTASCIIVar :
    fldType = "Delimited"
    fldDelimiter = dt.getSourceDelimiter()
    if dt.getSourceDelimitedFields() = DTDelimAllFields then
      fldDelimitedFields = "All"
    else
      fldDelimitedFields = "Text"
    endIF
    fldSeparator = dt.getSourceSeparator()
  case dt.getSourceType() = DTASCIIFixed :
    fldType = "Fixed"
  otherwise :
    msgInfo("Hello","File missing or not text.")

```

getSourceFieldNamesFromFirst method

```
endSwitch  
endMethod
```

getSourceFieldNamesFromFirst method

DataTransfer

Retrieves the True or False value set by setSourceFieldNamesFromFirst.

Syntax

```
getSourceFieldNamesFromFirst ( ) Logical
```

Description

getSourceFieldNamesFromFirst retrieves the True or False value set by **setSourceFieldNamesFromFirst**. **getSourceFieldNamesFromFirst** only applies when the source is a spreadsheet or a delimited text file.

Example

The following example specifies a DataTransfer data type. This structure is used with the **transferData** method. This example assumes that the DataTransfer variable (dt) is declared within a Var ... EndVar statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

```
method cmTransfer() ;this example completes a DataTransfer  
  
    dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source  
                                           ; to CUSTOMER.TXT  
    dt.setSourceSeparator("/") ; specifies the forward slash "/" character  
                               ; to separate each field  
    dt.setSourceDelimiter("'") ; specifies the single quote to surround  
                               ; the fields  
    dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single  
                                                ; quote (delimiter) surrounds  
                                                ; only text fields of the  
                                                ; source file  
    dt.setSourceCharSet(DTANSI) ; specifies that the character set used  
                               ; when creating the source file  
                               ; was the ANSI character set  
    msgInfo("Info", "the current setting is " +  
string(dt.getSourceFieldNamesFromFirst()))  
    dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first  
                                           ; row of the source file as  
                                           ; field names  
    dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB  
    dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are  
                        ; any problems importing the source file  
    dt.transferData() ; executes the data transfer. In this case it  
                    ; imports the CUSTOMER.TXT file as NEWCUST.DB.  
    dt.empty() ; empties the dt variable structure to set it up for  
              ; a new transfer.
```

```
endmethod
```

getSourceName method

DataTransfer

Retrieves the source filename.

Syntax

```
getSourceName ( ) String
```

Description

getSourceName retrieves the source filename.

Example

The following example determines if the user has attempted to import data from the SYSTEM.INI file.

```
method gtSrcName()

var
    dt DataTransfer
    importSourceFile String
endVar

importSourceFile = "Your sourcename here"
importsourcefile.view("Import what file?")

dt.setSource(importSourceFile, dtAuto) ; allow Paradox to determine filetype
if dt.getSourceName() = "system.ini" then
    msgStop("No!", "This source file won't create useable data.")
    return
else
    dt.setDest("importSample", dtParadox7) ; import into Paradox 7 table
    dt.transferData ( )
endif
endMethod
```

getSourceRange method**DataTransfer**

Retrieves the range set by setSourceRange.

Syntax

```
getSourceRange ( ) String
```

Description

getSourceRange retrieves the range set by the **setSourceRange** method. **getSourceRange** only applies when the source is a spreadsheet.

Example

The following demonstrates the **setSourceRange** method. You can use this method to specify the range in a spreadsheet to import. **getSourceRange** accepts both named ranges and standard ranges.

```
method pushButton(var eventInfo Event)
var
    dt DataTransfer
endVar
dt.setSource("092595.wb2")

;Set the range to import from the spreadsheet.
;Either named range or specified range (ie. Page1:A1..Page3:AB10)
msgInfo("Info", "The Current range is " + dt.getSourceRange())
dt.setSourceRange("myRange")
dt.setSourceFieldNamesFromFirst(True)
dt.setDest("delme09.db")

;Prompt the user to verify range to import. getSourceRange returns the
;actual range notation.
view(dt.getSourceRange(),"Import Range")
dt.transferData()
endMethod
```

getSourceSeparator method**DataTransfer**

Retrieves the value set by **setSourceSeparator**.

Syntax

```
getSourceSeparator ( ) String
```

Description

getSourceSeparator retrieves the value set by the **setSourceSeparator** method. **getSourceSeparator** only applies when the source is a delimited ASCII text file.

Example

The following example uses **getSourceSeparator** to display the separator used in a field on the specified form. This confirms that the separator is set correctly and allows you to specify a new separator if necessary.

```
method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
  endVar
  dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox determined as the type of
;of the source file. If it is delimited, Paradox determines the separator,
;delimiter and which fields are delimited.
  switch
    case dt.getSourceType() = DTASCIIVar :
      fldType = "Delimited"
      fldDelimiter = dt.getSourceDelimiter()
      if dt.getSourceDelimitedFields() = DTDelimAllFields then
        fldDelimitedFields = "All"
      else
        fldDelimitedFields = "Text"
      endIf
      fldSeparator = dt.getSourceSeparator()
    case dt.getSourceType() = DTASCIIFixed :
      fldType = "Fixed"
    otherwise :
      msgInfo("Hello","File missing or not text.")
  endSwitch
endMethod
```

getSourceType method**DataTransfer**

Retrieves the source file type constant.

Syntax

```
getSourceType ( ) SmallInt
```

Description

getSourceType retrieves the source file type constant. The file type constant is one of the **DataTransferFileType** constants. The version part of the file type is irrelevant to the source and can be ignored.

Example

The following example uses **getSourceType** to determine the file type of the source file:


```

method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
  endVar
  dt.setSource("iesimpld.txt")

;The following lines check to see what Paradox determined as the type of
;of the source file. If it is delimited, Paradox determines the separator,
;delimiter and which fields are delimited.
  switch
    case dt.getSourceType() = DTASCIIIFix :
      fldType = "Delimited"
      fldDelimiter = dt.getSourceDelimiter()
      if dt.getSourceDelimitedFields() = DTDelimAllFields then
        fldDelimitedFields = "All"
      else
        fldDelimitedFields = "Text"
      endIF
      fldSeparator = dt.getSourceSeparator()
    case dt.getSourceType() = DTASCIIIFixed :
      fldType = "Fixed"
    otherwise :
      msgInfo("Hello","File missing or not text.")
  endSwitch
endMethod

```

importASCIIIFix procedure

DataTransfer

Imports data from a fixed record length ASCII text file to a table.

Syntax

```
importASCIIIFix ( const fileName String, const tableName String, const specTableName
String [ , const ANSI Logical ] ) Logical
```

Description

importASCIIIFix imports data from an ASCII file in which each record's fields are the same length to a table. If the target table exists, its contents are replaced with the imported data. If the table does not exist, this method creates it. **importASCIIIFix** duplicates the function of the Import Data dialog box.

The argument *fileName* specifies the source file and *tableName* specifies the target table. Dates and numbers are formatted according to your settings in the Windows Control Panel. The file extension specified in *tableName* identifies the table type of the target table. .DB specifies a Paradox table and .DBF specifies a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

The argument *specTableName* is the name of a table that specifies the layout for the imported data. The following table illustrates the structure specified in *specTableName* :

Field name	Type & size	Description
Field Name	A 25	Name of the field to import
Type	A 4	Field type to import. The Type must be a valid Paradox or dBASE field specification.
Start	S	Column number where the field value begins
Length	S	Field size

importASCIIVar procedure

ANSI (optional) specifies whether to use the ANSI or OEM character set. Set *ANSI* to True to specify the ANSI character set, and to False to specify the OEM (default) character set.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example imports data from the ORDERS.TXT text file to the ORDERS.DB table. The ORDERS.DB table structure is read from the ORDERIMPDB table and the OEM character set is used.

```
method pushButton ( var eventInfo Event )
    importASCIIIFix ( "orders.txt", "orders.db", "orderimp.db", False )
endMethod
```

importASCIIVar procedure

DataTransfer

Imports data from an ASCII text file with variable field length values to a table.

Syntax

```
importASCIIVar ( const fileName String, const tableName String [ , const separator
String, const delimiter String, const allFieldsDelimited Logical, const ANSI Logical ]
) Logical
```

Description

importASCIIVar imports data from an ASCII file to a table. The source file's variable length field values in each record may be delimited by an optionally specified character. If the target table exists, its contents are replaced with the imported data. If the table does not exist, this method creates it.

importASCIIVar duplicates the function of the Import Data dialog box.

The argument *fileName* specifies the source file and *tableName* specifies the target table. Dates and numbers are formatted according to your settings in the Windows Control Panel. The file extension specified in *tableName* identifies the table type of the target table. .DB specifies a Paradox table and .DBF specifies a dBASE table. If you omit the extension, the data is imported to a Paradox table by default.

delimiter (optional) specifies the character that defines the limits of field values in the target. Leave the delimiter string empty if you do not want to define limits. The *allFieldsDelimited* (optional) string is marked True if data from all field types is delimited, and False if data from only text, alphanumeric, or character field types is delimited. Paradox truncates strings longer than 255 characters when it imports them.

ANSI (optional) specifies whether to use the ANSI or OEM character set. Set *ANSI* to True to use the ANSI character set, or to False to use the OEM character set.

The following table displays the default settings for optional arguments:

separator	, (comma)
delimiter	" (double quote) text fields only
ANSI	False

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example imports data from the ORDERS.TXT text file to the ORDERS.DB table. In this example, commas delimit field values, values are not enclosed, and the ANSI character set is used.

```
method pushButton ( var eventInfo Event )
    importASCIIVar ( "orders.txt", "orders.db", ",", "", True, True )
endMethod
```

The following example imports ASCII delimited text to Paradox (long form):

```
method pushButton ( var eventInfo Event )
var
    dt DataTransfer
endVar

dt.setSource("orders.txt", DTAsciiVar)
dt.setDest("orders.db")
dt.setSourceDelimiter("")
dt.setSourceSeparator(",")
dt.setSourceCharSet(dtANSI)
dt.setSourceDelimitedFields(dtDelimAllFields)

dt.TransferData()
endMethod
```

The following example imports ASCII delimited text to Paradox (medium form):

```
method pushButton ( var eventInfo Event )
var
    dt DataTransfer
endVar

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.db")
dt.TransferData()

endMethod
```

The following example imports ASCII Delimited Text to Paradox (short form):

```
method pushButton ( var eventInfo Event )
ImportASCIIVar("NewRecords.txt", "TimeCards.db")
endMethod
```

importSpreadsheet procedure**DataTransfer**

Imports data from a spreadsheet file to a table.

Syntax

```
importSpreadsheet ( const fileName String, const tableName String, const fromCell
String, const toCell String [ , const getFieldNames Logical ] ) Logical
```

Description

importSpreadsheet imports the data from a spreadsheet file to a table. Paradox converts rows to records and columns to fields. If the table does not exist prior to importing, this method creates it.

importSpreadsheet duplicates the function of the Import Data dialog box.

fileName specifies the spreadsheet or source file, and *tableName* specifies the table that displays the imported data. *fromCell* specifies the upper-left cell and *toCell* specifies the lower-right cell of the imported block. *getFieldNames* specifies whether to format the top row of the spreadsheet as column

importSpreadsheet procedure

headers for the table. If you set *getFieldNames* to True Paradox creates column headers (default); If you set *getFieldNames* to False, Paradox does not.

Note

The file extension specified in *fileName* identifies the format of the spreadsheet file. The following table illustrates the file extensions and their spreadsheet formats:

Extension	Format
WBI, WB2, WB3	Quattro Pro Win
WQI	Quattro Pro DOS
WKQ	Quattro
WKI	Lotus 2.x
WKS	Lotus 1.A
XLS	Excel 3.0/4.0/5.0

Note

The file extension specified in *tableName* identifies the target table's type. .DB specifies a Paradox table (default) and .DBF specifies a dBASE table.

Paradox automatically assigns a field type to each column of data. The following table shows how Paradox determines a field's type:

Spreadsheet value	Paradox field type	dBASE field type
Label	Alpha	Character
Integer	Short	Float number (5,0)
Number	Number	Float number (20,4)
Currency	Money	Float number (20,4)
Date	Date	Date

The following rules determine which category a column falls into:

- A column containing a label (text) is converted to an alpha field (or character field for a dBASE table).
- A column containing both dates and numbers is converted to an alpha field (or character field for a dBASE table).
- A column containing only values formatted as currency is converted to a money field in a Paradox table.
- A column containing both currency and number (or integer) values is converted to a number field.

Paradox often imports dates and numbers from unedited spreadsheets as alpha fields. For example, spreadsheets often have rows of hyphens separating sections of numbers. Since only an alphanumeric field can have both numbers and hyphens, Paradox converts each spreadsheet column to an alpha field even though it contains mostly numbers.

To avoid conversion problems, edit the spreadsheet before importing it. Follow these steps:

1. Remove extraneous entries such as hyphens, asterisks, and exclamation points.
2. Ensure each column contains one type of data and uses one formatting option.
3. Place the titles you want to format as table column headings in the top row of the selected range. Paradox uses the first row that contains text to generate field names. If the spreadsheet does not contain column titles, set the `getFieldNames` parameter to `False`.

If the table does not have the format you want after you import it, restructure it in Paradox.

This method is part of the Data Transfer type, but in previous versions it was included in the System type.

Example

The following example imports data from a Quattro Pro for Windows file to the ORDERS.DB table. This example converts the first row of the spreadsheet file to column headers in the table.

```
method pushButton ( var eventInfo Event )
    importSpreadsheet ( "orders.wb1", "orders.db", "A:A1", "A:H25", True )
endMethod
```

loadDestSpec method

DataTransfer

Loads a fixed-length import file specification.

Syntax

```
loadDestSpec ( const tableName String )
```

Description

loadDestSpec loads a fixed-length import file specification. The argument *tableName* specifies the table to use as the pattern for the destination specification. **loadDestSpec** applies only when the destination is a fixed-length ASCII text file.

Example

The following examples specify a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from text

```
var
    dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIIFixed Then
    dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

loadSourceSpec method

DataTransfer

Loads a fixed-length import file specification.

Syntax

```
loadSourceSpec ( const tableName String )
```

setAppend method

Description

loadSourceSpec loads a fixed-length import file specification. The argument *tableName* specifies the table to use as the pattern for the source specification. **loadSourceSpec** applies only when the source is a fixed-length ASCII text file.

Example

The following examples specify a DataTransfer data type. Use this code to build an Import or Export specification, and then call the **transferData** method.

Import from text

```
var
    dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIIFixed Then
    dt.loadSourceSpec ( "SpecTable" )
EndIf
dt.setSource ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

setAppend method

DataTransfer

Appends data to the existing table.

Syntax

```
setAppend ( const AppendToTable Logical )
```

Description

setAppend appends data to the existing table when set to True, and overwrites the table when set to False. This method is ignored for new tables, and applies only when the destination is a table.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from text

```
var
    dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIIFixed Then
    dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

setDest method

DataTransfer

Specifies the file or table to receive imported data.

Syntax

```
setDest ( const destName String, [ const destType SmallInt ] )
```

Description

setDest specifies the name and type of the file or table that receives data. If no file type is specified, the file extension determines its type. The file type constant *destType* specifies one of the *DataTransferFileType* constants.

Example

The following example specifies a *DataTransfer* data type. Use this code to build an Import or Export specification, and then call the **transferData** method.

Import from spreadsheet

```
var
  dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

setDestCharSet method**DataTransfer**

Sets the file character set to dtOEM or dtANSI.

Syntax

```
setDestCharSet ( const CharSetCode SmallInt )
```

Description

setDestCharSet sets the file character set to one of the two *DataTransferCharset* constants: dtOEM or dtANSI. This method applies only when the destination is a fixed or delimited ASCII text file.

Example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. This example uses **setDestCharSet** to set the file character set to ANSI. To set the file character set to OEM, use the DTOEM constant with **setDestCharSet**.

```
method pushButton(var eventInfo Event)
  var
    dt      DataTransfer
  endVar
  dt.setDest("ordinfo.txt", DTASCIIVar)
  dt.setSource("orders.db")

  ;Specify the single quote (') to surround the fields.
  ;The delimited fields will be text fields only.
  dt.setDestDelimiter("'")
  dt.setDestDelimitedFields(DTDelimJustText)

  ;Specify the tab character to separate the fields.
  dt.setDestSeparator("\t")

  ;Set the first row of the ORDINFO.TXT to be the field names
  dt.setDestFieldNamesFromFirst(True)

  ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
  ;character set.
  dt.setDestCharSet(DTAnsi)

  ;run Export
```

setDestDelimitedFields method

```
dt.transferData()  
endMethod
```

setDestDelimitedFields method

DataTransfer

Sets the Delimited Fields setting to DtDelimAllFields or DtDelimJustText.

Syntax

```
setDestDelimitedFields ( const delimitCode SmallInt )
```

Description

setDestDelimitedFields sets the Delimited Fields setting. The argument *delimitCode* specifies one of the two DataTransferDelimitCode constants: DtDelimAllFields or DtDelimJustText.

setDestDelimitedFields only applies when the destination is a delimited ASCII text file.

Example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. It uses **setDestDelimitedFields** to delimit surrounding text fields only. To delimit all fields, use the DTDelimAllFields constant with **setDestDelimitedFields**.

```
method pushButton(var eventInfo Event)  
  var  
    dt      DataTransfer  
  endVar  
  dt.setDest("ordinfo.txt", DTASCIIVar)  
  dt.setSource("orders.db")  
  
  ;Specify the single quote (') to surround the fields.  
  ;The delimited fields will be text fields only.  
  dt.setDestDelimiter("'")  
  dt.setDestDelimitedFields(DTDelimJustText)  
  
  ;Specify the tab character to separate the fields.  
  dt.setDestSeparator("\t")  
  
  ;Set the first row of the ORDINFO.TXT to be the field names  
  dt.setDestFieldNamesFromFirst(True)  
  
  ;Set the character set of the destination file ORDINFO.TXT to be the ANSI  
  ;character set.  
  dt.setDestCharSet(DTAnsi)  
  
  ;run Export  
  dt.transferData()  
endMethod
```

setDestDelimiter method

DataTransfer

Specifies a character as the delimiter.

Syntax

```
setDestDelimiter ( const delimiterChar String )
```

Description

setDestDelimiter sets the delimiter to the character specified by *delimiterChar*. The default delimiter is a comma. **setDestDelimiter** only applies when the destination is a delimited ASCII text file.

Example

The following example exports the ORDERS.DB table into an ASCII delimited text file. The delimiter single quote character is the specified delimiter.

```
method pushButton(var eventInfo Event)
    var
        dt          DataTransfer
    endVar
    dt.setDest("ordinfo.txt", DTASCIIVar)
    dt.setSource("orders.db")

    ;Specify the single quote (') to surround the fields.
    ;The delimited fields will be text fields only.
    dt.setDestDelimiter("'")
    dt.setDestDelimitedFields(DTDelimJustText)

    ;Specify the tab character to separate the fields.
    dt.setDestSeparator("\t")

    ;Set the first row of the ORDINFO.TXT to be the field names
    dt.setDestFieldNamesFromFirst(True)

    ;Set the character set of the destination file ORDINFO.TXT to be the ANSI
    ;character set.
    dt.setDestCharSet(DTAnsi)

    ;run Export
    dt.transferData()
endMethod
```

setDestFieldNamesFromFirst method**DataTransfer**

Sets field names using the data in the first row of input.

Syntax

```
setDestFieldNamesFromFirst ( const namesFirst Logical )
```

Description

setDestFieldNamesFromFirst sets the first row of the destination file to be the field names of the table. Setting *namesFirst* to True creates the first row as field names and data will begin on the second row.

setDestFieldNamesFromFirst applies to both spreadsheets and delimited text files.

Example

The following example uses the **transferData** method to export ORDERS.DB to ORDINFO.TXT. The **setDestFieldNamesFromFirst** is used to set the field names using the data in the first row of the text file.

```
method pushButton(var eventInfo Event)
    var
        dt          DataTransfer
    endVar
    dt.setDest("ordinfo.txt", DTASCIIVar)
    dt.setSource("orders.db")

    ;Specify the single quote (') to surround the fields.
    ;The delimited fields will be text fields only.
    dt.setDestDelimiter("'")
```

setDestSeparator method

```
dt.setDestDelimitedFields(DTDelimJustText)

;Specify the tab character to separate the fields.
dt.setDestSeparator("\t")

;Set the first row of the ORDINFO.TXT to be the field names
dt.setDestFieldNamesFromFirst(True)

;Set the character set of the destination file ORDINFO.TXT to be the ANSI
;character set.
dt.setDestCharSet(DTAnsi)

;run Export
dt.transferData()
endMethod
```

setDestSeparator method

DataTransfer

Sets the separator character for delimited ASCII text.

Syntax

```
setDestSeparator ( const separatorChar String )
```

Description

setDestSeparator sets the separator to the character specified by *separatorChar*. The default separator is the comma character. **setDestSeparator** only applies when the destination is a delimited ASCII text file.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification, and then call the transferData method.

Export to text

```
var
    dt DataTransfer
endVar
msgInfo("Info", "The current source separator is " + dt.getSourceSeparator())
dt.setSource ( "ANSWER.DB" )
msgInfo("Info", "The current destination separator is " + dt.getDestSeparator())
dt.SetDest ( "NEWFILE.TXT",dtASCIIVar )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

setKeyviol method

DataTransfer

Writes violations to the Keyviol table.

Syntax

```
setKeyviol ( const GenerateKeyviol Logical )
```

Description

setKeyviol writes violations to the Keyviol table. The argument *generateKeyviol* is a logical that is set to True to write violations to the Keyviol table. *generateKeyviol* is ignored for tables without keys. **setKeyviol** applies only when the destination is a table.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Imports ASCII delimited text to Paradox (long form):

```
var
    dt DataTransfer
endVar

; Fields Quoted even if numeric
dt.setAppend(True)      ; Append to an existing Table
dt.setProblems(True)    ; Generate a Problems Table (if Any)
dt.setKeyviol(True)     ; Generate a Keyviol Table (if any)

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.db")
dt.TransferData()
```

setProblems method**DataTransfer**

Writes problems to a specified table.

Syntax

```
setProblems ( const generateProblems Logical )
```

Description

setProblems writes problems to the Problems table. This method applies only when the destination is a table.

Example

The following example specifies a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Imports ASCII Delimited Text to Paradox (long form):

```
var
    dt DataTransfer
endVar

; Fields Quoted even if numeric
dt.setAppend(True)      ; Append to an existing Table
dt.setProblems(True)    ; Generate a Problems Table (if Any)
dt.setKeyviol(True)     ; Generate a Keyviol Table (if any)

dt.setSource("NewRecords.txt", DTAsciiVar)
dt.setDest("TimeCards.DB")
dt.TransferData()
```

setSource method**DataTransfer**

Specifies the file or table that acts as the data source and its type.

Syntax

```
setSource ( const sourceName String, [ const sourceType SmallInt ] )
```

setSourceCharSet method

Description

setSource specifies the file or table to use as the data source. **SourceType** specifies the field type according to the application that generated the file. If no file type is specified, the *sourceName* file extension is used to determine the file's type. The file type constant *destType* specifies one of the **DataTransferFileType** constants.

Example

The following example specifies a **DataTransfer** data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from spreadsheet

```
var
    dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

setSourceCharSet method

DataTransfer

Sets the file character set to dtOEM or dtANSI.

Syntax

```
setSourceCharSet ( const charSetCode SmallInt )
```

Description

setSourceCharSet sets the file character set. The argument *charSetCode* specifies one of the two **DataTransferCharset** constants: dtOEM or dtANSI. **setSourceCharSet** applies only when the source is a fixed or delimited ASCII text file.

Example

The following example specifies a **DataTransfer** data type. This structure is used with the **transferData** method. This example assumes that the **DataTransfer** variable, dt, is declared within a **Var ... EndVar** statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

method **cmTransfer()** ;this example completes a **DataTransfer**

```
dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
                                         ; to CUSTOMER.TXT
dt.setSourceSeparator("/") ; specifies the forward slash "/" character
                           ; to separate each field
dt.setSourceDelimiter("'") ; specifies the single quote to surround
                           ; the fields
dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
                                             ; quote (delimiter) surrounds
                                             ; only text fields of the
                                             ; source file
dt.setSourceCharSet(DTANSI) ; specifies that the character set used
                           ; when creating the source file
                           ; was the ANSI character set

dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
                                       ; row of the source file as
                                       ; field names
dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
```

```

; any problems importing the source file
dt.transferData() ; executes the data transfer. In this case it
; imports the CUSTOMER.TXT file as NEWCUST.DB.
dt.empty() ; empties the dt variable structure to set it up for
; a new transfer.

endmethod

```

setSourceDelimitedFields method

DataTransfer

Sets the Delimited Fields setting to DtDelimAllFields or DtDelimJustText.

Syntax

```
setSourceDelimitedFields ( const delimitCode SmallInt )
```

Description

setSourceDelimitedFields sets the delimited fields value. The argument *delimitCode* specifies one of the two DataTransferDelimitCode constants: DtDelimAllFields or DtDelimJustText.

setSourceDelimitedFields only applies when the source is a delimited ASCII text file.

Example

The following example specifies a DataTransfer data type. This structure is used with the **transferData** method. This example assumes that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

```

method cmTransfer() ;this example completes a DataTransfer

    dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
; to CUSTOMER.TXT
    dt.setSourceSeparator("/") ; specifies the forward slash "/" character
; to separate each field
    dt.setSourceDelimiter("'") ; specifies the single quote to surround
; the fields
    dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
; quote (delimiter) surrounds
; only text fields of the
; source file
    dt.setSourceCharSet(DTANSI) ; specifies that the character set used
; when creating the source file
; was the ANSI character set

    dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
; row of the source file as
; field names
    dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
    dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
; any problems importing the source file
    dt.transferData() ; executes the data transfer. In this case it
; imports the CUSTOMER.TXT file as NEWCUST.DB.
    dt.empty() ; empties the dt variable structure to set it up for
; a new transfer.

endmethod

```

setSourceDelimiter method

DataTransfer

Specifies a character as the delimiter.

setSourceFieldNamesFromFirst method

Syntax

```
setSourceDelimiter ( const delimiterChar String )
```

Description

setSourceDelimiter sets the delimiter to the character specified by *delimiterChar*. The default delimiter is a comma. **setSourceDelimiter** only applies when the source is a delimited ASCII text file.

Example

The following example specifies a DataTransfer data type. This structure is used with the **transferData** method. This example assumes that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

```
method cmTransfer() ;this example completes a DataTransfer

    dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
                                           ; to CUSTOMER.TXT
    dt.setSourceSeparator("/") ; specifies the forward slash "/" character
                             ; to separate each field
    dt.setSourceDelimiter("'") ; specifies the single quote to surround
                             ; the fields
    dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
                                           ; quote (delimiter) surrounds
                                           ; only text fields of the
                                           ; source file
    dt.setSourceCharSet(DTANSI) ; specifies that the character set used
                             ; when creating the source file
                             ; was the ANSI character set

    dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
                                           ; row of the source file as
                                           ; field names
    dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
    dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
                       ; any problems importing the source file
    dt.transferData() ; executes the data transfer. In this case it
                    ; imports the CUSTOMER.TXT file as NEWCUST.DB.
    dt.empty() ; empties the dt variable structure to set it up for
              ; a new transfer.

endmethod
```

setSourceFieldNamesFromFirst method

DataTransfer

Sets field names using the data in the first row of input.

Syntax

```
setSourceFieldNamesFromFirst ( const namesFirst Logical)
```

Description

setSourceFieldNamesFromFirst sets field names using the first row of the input data. Setting *namesFirst* to True always skips the first row. However, the field names only apply to newly created tables that do not already have field names. **setSourceFieldNamesFromFirst** only applies when the source is a spreadsheet or a delimited text file.

Example

The following example specifies a DataTransfer data type. This structure is used with the **transferData** method. This example assumes that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

```
method cmTransfer() ;this example completes a DataTransfer

    dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
                                           ; to CUSTOMER.TXT
    dt.setSourceSeparator("/") ; specifies the forward slash "/" character
                               ; to separate each field
    dt.setSourceDelimiter("'") ; specifies the single quote to surround
                               ; the fields
    dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
                                                ; quote (delimiter) surrounds
                                                ; only text fields of the
                                                ; source file
    dt.setSourceCharSet(DTANSI) ; specifies that the character set used
                               ; when creating the source file
                               ; was the ANSI character set

    dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
                                           ; row of the source file as
                                           ; field names
    dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
    dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
                        ; any problems importing the source file
    dt.transferData() ; executes the data transfer. In this case it
                    ; imports the CUSTOMER.TXT file as NEWCUST.DB.
    dt.empty() ; empties the dt variable structure to set it up for
              ; a new transfer.

endmethod
```

setSourceRange method**DataTransfer**

Specifies a sub range of the spreadsheet to import.

Syntax

```
setSourceRange ( const range String )
```

Description

setSourceRange specifies a sub range of the spreadsheet to import. The value specified by **setSourceRange** can be a named range, a page name, or an explicit range in QPW or Excel format. **setSourceRange** only applies when the source is a spreadsheet.

Example

The following example uses the **setSourceRange** method to specify a range in a spreadsheet to import. You can specify named ranges and standard ranges.

```
method pushButton(var eventInfo Event)
    var
        dt      DataTransfer
    endVar
    dt.setSource("092595.wb2")

    ;Set the range to import from the spreadsheet.
    ;Either named range or specified range (ie. Page1:A1..Page3:AB10)
```

setSourceSeparator method

```
dt.setSourceRange("myRange")
dt.setSourceFieldNamesFromFirst(True)
dt.setDest("delme09.db")

;Prompt the user to verify range to import. getSourceRange returns the
;actual range notation.
view(dt.getSourceRange(),"Import Range")
dt.transferData()
endMethod
```

setSourceSeparator method

DataTransfer

Sets the separator character for delimited ASCII text.

Syntax

```
setSourceSeparator ( const separatorChar String )
```

Description

setSourceSeparator sets the separator to the character specified by *separatorChar*. The default separator is a comma. **setSourceSeparator** only applies when the source is a delimited ASCII text file.

Example

The following example specifies a DataTransfer data type. This structure is used with the **transferData** method. This example assumes that the DataTransfer variable, dt, is declared within a Var ... EndVar statement. The custom method **cmTransfer()** is within the scope of the variable (dt).

```
method cmTransfer() ;this example completes a DataTransfer

dt.setSource("CUSTOMER.TXT", DTASCIIVar) ; sets the datatransfer source
                                         ; to CUSTOMER.TXT
dt.setSourceSeparator("/") ; specifies the forward slash "/" character
                           ; to separate each field
dt.setSourceDelimiter("'") ; specifies the single quote to surround
                           ; the fields
dt.setSourceDelimitedFields(DTDelimJustText) ; specifies that the single
                                             ; quote (delimiter) surrounds
                                             ; only text fields of the
                                             ; source file
dt.setSourceCharSet(DTANSI) ; specifies that the character set used
                           ; when creating the source file
                           ; was the ANSI character set

dt.setSourceFieldNamesFromFirst(False) ; specifies to use the first
                                       ; row of the source file as
                                       ; field names
dt.setDest("NEWCUST.DB") ; sets the destination file to NEWCUST.DB
dt.setProblems(True) ; specifies to create a PROBLEMS.DB if there are
                    ; any problems importing the source file
dt.transferData() ; executes the data transfer. In this case it
                 ; imports the CUSTOMER.TXT file as NEWCUST.DB.
dt.empty() ; empties the dt variable structure to set it up for
           ; a new transfer.

endmethod
```


transferData method**DataTransfer**

Copies data from the source to the target.

Syntax

```
transferData ( )
```

Description

transferData copies data from the source to the destination. This method applies only if the source, the destination, or both the source and the destination are tables.

Example

The following examples specify a DataTransfer data type. Use this code to build an Import or Export specification and then call the **transferData** method.

Import from spreadsheet

```
var
    dt DataTransfer
endVar
dt.setSource ( "MYFILE.WKS" )
dt.setDest ( "New Data.db" )
dt.setProblems ( True )
dt.transferData ( )
```

Import from text

```
var
    dt DataTransfer
endVar
dt.SetSource ( "MYFILE.TXT" )
if dt.getSourceType ( ) = DTASCIIFixed Then
    dt.loadDestSpec ( "SpecTable" )
EndIf
dt.setDest ( "Existing Data.db" )
dt.setAppend ( True )
dt.transferData ( )
```

Export to text

```
var
    dt DataTransfer
endVar
dt.setSource ( "ANSWER.DB" )
dt.SetDest ( "NEWFILE.TXT" )
dt.setDestSeparator ( ";" )
dt.transferData ( )
```

Date type

In ObjectPAL, you can represent Date values in either month/day/year, day-month-year, or day.month.year format. Dates must be explicitly declared. For example, the following code assigns the date December 21 1997 to d.

```
var
    d Date
endVar
d = date("12/21/1997")
```

If you omit the quotes around the Date value ObjectPAL divides the values.

date method

The Date type includes methods defined for the AnyType type and the DateTime type. For more information, see AnyType and DateTime.

Date values are formatted by the **formatSetDateDefault** method (System type), or by ObjectPAL formatting statements. See the formatSetDateDefault method for more information.

Dates from the 20th century can be specified using two digits for the year, as in

```
myDay = date("11/09/59") ; November 9, 1959
```

Dates from the 2nd to the 10th centuries must include three digits of the year (e.g., 12/17/243). Dates from the 11th to the 19th centuries must have four digits (e.g.,12/17/1043). The year cannot be omitted completely. Paradox treats all dates in the B.C. era as leap years.

Note

- When specifying a two digit year:
00-49 refers to 2000
50-99 refers to 1900

The Date type includes several derived methods from the DateTime and AnyType types. The Date type also includes several methods defined for the DateTime type. For more information, see DateTime.

Methods for the Date type

AnyType	←	DateTime	←	Date
blank		day		date
dataType		daysInMonth		dateVal
isAssigned		dow		today
isBlank		dowOrd		
isFixedType		doy		
view		isLeapYear		
		month		
		moy		
		year		

date method

Date

Returns a Date value.

Syntax

1. date (const value AnyType) Date
2. date () Date
3. date (month SmallInt, day SmallInt, year SmallInt) Date

Description

date casts **value** as a date. If the date specified by **value** is invalid, the procedure fails. If you do not define **value**, **date** returns the current system date as a Date value.

If you use Syntax 3, the month ranges from 1 to 12. The day range depends on the month and can range from 1 to 28 or 31. The year can range from -9999 to 9999 all four digits must be used (i.e., 1997). An error is returned if a value does not lie within the required range. For example, specifying 40 for the day value generates a Bad Day Specification error message.

Note

- When using dates in calculations, note that in Paradox, day 0 falls on 1/1/0001.

Example

The following example casts a String value as a date, uses the Date value in a calculation, and 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 and add 7 days
                ; and add 7 days

d.view()       ; show value of d in a dialog box (11/18/99)
                ; dialog box title displays "Date"

endMethod
```

dateVal procedure**Date**

Returns a specified value as a date.

Syntax

```
dateVal ( const value AnyType ) Date
```

Description

dateVal returns a specified value as a date.

Example

In the following example, the **pushButton** method for a button uses **dateVal** to convert a String value into a Date value. The result is displayed 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
```

today procedure**Date**

Returns the current date.

Syntax

```
today ( ) Date
```

today procedure

Description

today returns the current date, as displayed on your system clock/calendar.

Example

The following 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
```

DateTime type

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 explicitly declared. For example, in the following statements, the time assigned to the **DateTime** variable **dt** is 10 minutes and 40 seconds past eleven o'clock and the date is December 21, 1997. **DateTime** values must be enclosed in quotation marks.

```
var dt DateTime endVar
dt = DateTime("11:10:40 am 12/21/97")
```

You can use the following characters as separators in your **DateTime** specifications: blank, tab, space, comma, hyphen, slash, period, colon, and semicolon. **DateTime** values are formatted by the **formatSetDateTimeDefault** procedure (System type) or by ObjectPAL formatting statements.

You must specify a **DateTime** value completely, including all of the fields. Specify a value of zero for empty fields.

For more information, see the methods and procedures for the **Date** type and the **Time** type.

The following table lists the methods of the **DateTime** type, including several derived methods from the **AnyType** type.

Methods for the DateTime type

AnyType	←	DateTime
blank		dateTime
dataType		day
isAssigned		daysInMonth
isBlank		dow
isFixedType		dowOrd
view		doy
		hour
		isLeapYear
		milliSec
		minute
		month
		moy
		second
		year

dateTime method**DateTime**

Returns a DateTime value.

Syntax

1. `dateTime (const value AnyType) DateTime`
2. `dateTime () DateTime`

Description

dateTime casts *value* as a DateTime data type. If *value* is not supplied, **dateTime** returns the system date and time as a DateTime value.

Example

The following statements assign date and time values to the DateTime variable dt. The time is 10 minutes and 40 seconds past eleven o'clock and the date is December 21, 1997. This code assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy. DateTime values must be enclosed in quotation marks.

```
var dt DateTime endVar
dt = dateTime("11:10:40 am 12/21/97")
```

You can use the following characters as separators in your DateTime specifications: blank, tab, space, comma, hyphen, slash, period, colon, and semicolon. DateTime values are formatted by **formatSetDateTimeDefault** (System type) or by ObjectPAL formatting statements.

You must specify a DateTime value completely, including all of the fields. Specify a value of zero for empty fields.

day method**DateTime**

Extracts the day of the month from a date.

Syntax

```
day ( ) SmallInt
```

Description

day extracts the day of the month from a DateTime value and returns a value between 1 and 31. If the DateTime value is invalid, the method fails.

Example

The following example uses a button's **pushButton** method to display 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
```

daysInMonth method**DateTime**

Returns the number of days in a month.

Syntax

```
daysInMonth ( ) SmallInt
```

Description

daysInMonth returns the number of days in the month specified in a DateTime value. If the DateTime value is invalid, the method fails.

Example

The following example uses a button's **pushButton** method to display the number of days in February 1992 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.

```
; 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
```

dow method**DateTime**

Returns the day of the week.

Syntax

```
dow ( ) String
```

Description

dow returns the first three letters of the day in a specified DateTime value. If the DateTime value is not valid, the method fails.

Example

The following example displays the day of week from a specified DateTime value. 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
```

dowOrd method**DateTime**

Returns the number representing a specified day's position in the week.

Syntax

```
dowOrd ( ) SmallInt
```

Description

dowOrd returns an integer from one to seven representing a specified day's position in the week. Sunday is day one, Monday is day two, and so on. If the **DateTime** value given is invalid, the method fails.

Example

The following example displays the day of the week by name rather than by abbreviation or 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
```

doy method**DateTime**

Returns the number representing a specified day's position in the year.

Syntax

```
doy ( ) SmallInt
```

Description

doy returns an integer from 1 to 366 representing a specified day's position in the year. January 1 is day one, February 1 is day 32, and so on. If the **DateTime** value given is invalid, the method fails.

Example

The following example uses a button's **pushButton** method to display a day's position in a specified year. This example assumes the current date and time format is in the form hh:mm:ss am/pm mm/dd/yy.

hour method

```
; 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
```

hour method

DateTime

Extracts the hour from a specified DateTime value.

Syntax

```
hour ( ) SmallInt
```

Description

hour returns an integer representing the hour of the day in the 24-hour format. If the DateTime value given is not valid, the method fails.

Example

The following example extracts the hour from a specified DateTime and displays it in a dialog box. If the DateTime value is specified in the 12-hour format, hour returns its 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
```

isLeapYear method

DateTime

Reports whether a year has 366 days.

Syntax

```
isLeapYear ( ) Logical
```

Description

isLeapYear returns True if the year within a specified DateTime value has 366 days; otherwise, it returns False. If the DateTime value given is not valid, the method fails.

Note

- **isLeapYear** returns True for all B.C. era dates.

Example

The following example uses the **pushButton** method for the *testLeapYr* button to display True if the specified **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
```

milliSec method**DateTime**

Extracts the milliseconds from a **DateTime**.

Syntax

```
milliSec ( ) SmallInt
```

Description

milliSec returns an integer representing the milliseconds specified in a **DateTime** value. If the **DateTime** value given is not valid, the method fails.

Example

The following example constructs a **DateTime** value from integer calculations and displays the milliseconds 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
```

minute method**DateTime**

Extracts the minutes from a **DateTime**.

month method

Syntax

```
minute ( ) SmallInt
```

Description

minute returns an integer representing the minutes in a specified DateTime value. If the DateTime value given is not valid, the method fails.

Example

The following example uses the **pushButton** method for thisButton to display the minutes in 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
    dt DateTime
endVar

dt = DateTime("9:20:15 am 8/2/93")

msgInfo("Minutes", dt.minute())    ; displays 20

endMethod
```

month method

DateTime

Extracts as a number the month from a specified DateTime.

Syntax

```
month ( ) SmallInt
```

Description

month returns an integer representing the specified month's position in the year. January is month one, February is month two, and so on. If the DateTime value given is not valid, the method fails.

Example

The following example displays the month of the year by name rather than by abbreviation or number. This code uses **month** to retrieve the appropriate subscript of a fixed array and 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

```

moy method

DateTime

Extracts the month from a specified DateTime as a string.

Syntax

```
moy ( ) String
```

Description

moy returns the first three letters of the name of the specified month. If the DateTime value given is not valid, the method fails.

Example

The following example uses the **pushButton** method for *thisButton* to display 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

```

second method

DateTime

Extracts the seconds from a specified DateTime.

Syntax

```
second ( ) SmallInt
```

Description

second returns an integer representing the seconds in a DateTime. If the DateTime value given is not valid, the method fails.

Example

The following example constructs a DateTime value from integer calculation and displays the seconds the DateTime in a dialog box.

```

; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    dt DateTime

```

year method

```
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
```

year method

DateTime

Extracts the year from a specified DateTime.

Syntax

```
year ( ) SmallInt
```

Description

year returns an integer representing the year within a specified DateTime. If the DateTime value given is invalid, this method fails.

Example

The following example uses yearButton's **pushButton** method to display the four-digit year in a specified DateTime value. 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
```

DDE type

Dynamic data exchange (DDE) is a Windows protocol that allows Paradox to share data with other applications that adopt the DDE protocol. DDE methods grant you access to data created and stored in other applications. You can also use DDE methods to send commands and data to other applications.

Notes

- When you use DDE to access Paradox from another application, the Paradox application name is PDXWIN32.
- Paradox and ObjectPAL also support OLE, another protocol for sharing data between applications. For more information, see the OLE type.

Methods for the DDE type**DDE**

close

execute

open

setItem

close method**DDE**

Closes a DDE link.

Syntax

```
close ( )
```

Description

close ends a DDE conversation by closing the link between Paradox and the other application. **close** does not affect the status of the other application.

Example

The following example retrieves data from a Quattro Pro for Windows worksheet and then uses the **close** method to close the DDE link.

```
method run(var eventInfo Event)

var
    ddeVar    DDE      ; used as DDE link to QPW application
    Winery    AnyType  ; hold answer from a cell in application
endVar

; find the path where QPW.exe resides
strLevel = getRegistryValue( "Software\\Microsoft\\Windows\\CurrentVersion\\"+"App
Paths\\QPW.exe","",RegKeyLocalMachine )
; run QPW.exe
if not execute(strLevel ) then
    msgStop("Execute QPW", "FAIL: Could not execute QPW.exe")
    return
endif

; use sendKeys to open up an application in QPW.
; use sleep() to ensure that keys were sent properly.
sleep(3000)
sendKeys("%fo")
sleep(2000)
sendKeys("C:\\core1\\samples\\wines.wb3")
sleep(500)
sendKeys("{enter}")
sleep(3000)

; open a DDE link to the application
ddeVar.open("QPW", "C:\\core1\\samples\\wines.wb3", "$A:$C$2")

; get the value and message it to user.
Winery = ddeVar

; close the window
sendKeys("%{f4}")
```

execute method

```
msginfo("First Winery", Winery)

; close the DDE link. This should not affect the status of the window.
ddeVar.close()

endMethod
```

execute method

DDE

Uses a DDE link to send a command to another application.

Syntax

```
execute ( const command String )
```

Description

execute uses a DDE link to send the string *command* to an application. The nature of *command* varies from one application to another. For example, a string that is understood by a word processing program may not be accepted by a spreadsheet application, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Example

See the **open** example.

open method

DDE

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

open creates a DDE link to another application, and instructs the application to open a document specified in *item*.

This method returns True if the application is successfully opened; otherwise, it returns False. If the server application cannot open the application this method fails.

The nature of *item* varies from one application to another. For example, a string that is understood by a word processing program may not be accepted by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Note

- A DDE session can only be started with a running application, a fully-registered application, or an application that resides in the known system path (e.g., an application that is within the path statement in the Autoexec.bat).

Example

The following example uses a Paradox DDE Session to launch WordPerfect, minimize the application and invoke the WordPerfect Import dialog box.

This example uses **getRegistryValue** to locate the path for the WordPerfect executable, and uses **execute** to launch the application.

```

Method pushButton(var eventInfo Event)
var
    wpDDE      dde      ;declare a variable of DDE type
    strLevel   String   ;declare a variable of string type
endVar

    strLevel = getRegistryValue( "Software\\Microsoft\\Windows\\CurrentVersion\\App
Paths\\WPWin.exe","",
    RegKeyLocalMachine ) ; check registry for path to WordPerfect application
if not execute(strLevel ) THEN
    ;attempt to launch WordPerfect
    MSGINFO("Stop","Could not find WordPerfect!") ;alert user if launch failed
else
    sleep(5000) ; sleep allows WordPerfect time to open and get ready to accept DDE
commands

if not wpDDE.open("WPWin_MACROS","commands") then ;attempt to start DDE dialog with
WordPerfect
    sleep (5000);sleep some more in case WordPerfect isn't fully open
    wpDDE.open("WPWin_MACROS","commands") ; try DDE link again
    wpDDE.execute("AppMinimize()") ; minimize WordPerfect
    wpDDE.execute("importdlg ()") ; open WordPerfect's import dialog box
else
    wpDDE.execute("AppMinimize()") ; minimize WordPerfect
    wpDDE.execute("importdlg ()") ;open WordPerfect's import dialog box
endif
endif
endMethod

```

setItem method

DDE

Specifies an item in a DDE conversation.

Syntax

```
setItem ( const item String )
```

Description

setItem specifies an item in a DDE link where the application and topic are established. The argument *item* specifies a new item. The nature of *item* varies from application to application. For example, a string that is understood by a word processing program may not be accepted by a spreadsheet, and spreadsheets from different manufacturers may use different commands to perform similar activities.

Example

The following example uses **setItem** to retrieve the values of two cells in a QPW worksheet:

```

method run(var eventInfo Event)

var
    winesLink      DDE      ; DDE link to Quattro Pro
    Appellation, RegionAnyType ; values in Quattro Pro file
    strLevel       string   ; path to QPW.exe
endVar

; check registry for path to QuattroPro application
strLevel = getRegistryValue( "Software\\Microsoft\\Windows\\CurrentVersion\\"+"App
Paths\\QPW.exe","",RegKeyLocalMachine )

; execute QPW
if not execute(strLevel) then

```

setItem method

```
        msgStop("Execute", "FAIL: Could not execute QPW.exe")
        return
    endif

    ; open up wines.wb3 in QPW
    sleep(3000)
    sendKeys("%fo")
    sleep(200)
    sendKeys("c:\\corel\\samples\\wines.wb3")

    sleep(500)
    sendKeys("{enter}")
    sleep(2000)

    ; open a DDE link to the application
    if not winesLink.open("QPW", "c:\\corel\\samples\\wines.wb3") then
        msgStop("Open", "FAIL: Could not open QPW")
        return
    endif

    sleep(2000)
    winesLink.setItem("$A:$D$2")    ; item is cell A:D2
    Appellation = winesLink        ; sets Appellation = cell D2

    winesLink.setItem("$A:$E$2")    ; item is cell A:E2
    Region = winesLink              ; sets Region = cell E2

    ; close the DDE link. This should not affect the status of the window.
    winesLink.close()
    sendKeys("%{F4}")

    ;message results appear to user.
    msgInfo("Appellation:", Appellation)
    msgInfo("Region: ", Region)

endMethod
```

DynArray type

A DynArray is a flexibly structured dynamic array. Using a DynArray, you can retrieve values quickly, even when the dynamic array contains a large number of items.

This type of array is dynamic because you do not specify its size. Instead, a DynArray's dimensions automatically change as items are added or removed. A DynArray's size is limited only by system memory.

ObjectPAL also supports fixed-size and resizable arrays. For more information, see Array type.

The indexes of dynamic arrays are not integers; dynamic array indexes (also called keys) can be any valid ObjectPAL expression that evaluates to a String. Each index in a dynamic array is associated with a value.

An array is not derived from Anytype. Each element in the array is derived from the Anytype class.

The following table list the methods of the DynArray type, including several derived methods from the AnyType type.

Methods for the DynArray type

AnyType	←	DynArray
blank		contains
dataType		empty
isAssigned		getKeys
isBlank		removeItem
isFixedType		size
		view

contains method**DynArray**

Searches the indexes in a DynArray.

Syntax

```
contains ( const value AnyType ) Logical
```

Description

contains returns True if an elements index in a DynArray matches the specified value, character for character; otherwise, it returns False. **contains** is not 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.

The following 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 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

  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
```

empty method

```
        ; 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
```

The following code is attached to the form's **menuAction** method:

```
; 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
endif

else
    ;code here executes just for form itself
endif
endMethod
```

empty method

DynArray

Removes all items from a dynamic array.

Syntax

```
empty ( )
```

Description

empty removes all items from an dynamic array. The size of the DynArray becomes 0.

Example

The following example removes all items from 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"] = "Porsche" ; 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

```

getKeys method

DynArray

Loads the indexes of an existing DynArray into a resizable array.

Syntax

```
getKeys ( var keyNames Array[ ] String )
```

Description

getKeys creates the resizable array specified in *keyNames* and assigns the index in the DynArray to the values of each element. This method loads the index values from a DynArray into a resizable array. If *keyNames* exists, **getKeys** overwrites it without asking for confirmation. Index values are sorted into the new array so that the lowest index value becomes *keyNames*[1].

Example

The following example assigns several elements to the *myCar* DynArray and then uses **getKeys** to create an array that stores the *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"] = "Porsche" ; 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

```

removeItem method**DynArray**

Deletes a specified item from a DynArray.

Syntax

```
removeItem ( const value AnyType )
```

Description

removeItem deletes the item in value (specified by its index) from a DynArray. **removeItem** is not case sensitive.

Example

The following example concatenates two values in a dynamic array and uses **removeItem** to remove the obsolete element.

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

```
; thisForm::Var
var
  CustInfo DynArray[] AnyType
endVar
```

The following 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 following code, the value of the *ZipExt* element is concatenated to the value of the *Zip* element and the *ZipExt* element is removed 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
```

size method**DynArray**

Returns the number of elements in a DynArray.

Syntax

```
size ( ) LongInt
```

Description

size returns the number of elements in a DynArray.

Example

The following example uses the **pushButton** method for *thisButton* to create a dynamic array and 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
```

view method**DynArray**

Displays the contents of a DynArray in a dialog box.

Syntax

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

Description

view displays the indexes and elements of a DynArray in a modal dialog box. ObjectPAL execution suspends until the user closes this dialog box. *title* specifies the title of the dialog box. If you omit the title string, the dialog box is named DynArray. **view** sorts the DynArray on its index before displaying the dialog box.

Unlike other data types, DynArray values displayed in a view dialog box cannot be changed interactively. See view (AnyType type) for information about other data types.

Example

The following example uses the **pushButton** method for the *thisButton* button to create a dynamic array and displays its contents 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
```

User-defined error constants

```
        ; two      second
    endMethod
```

ErrorEvent type

The ErrorEvent type provides methods that allow you to retrieve and set information about ObjectPAL execution errors. The only built-in event method triggered by an ErrorEvent is error.

The following table lists the methods of the ErrorEvent type, including several derived methods from the Event type.

You can also define your own error constants, as long as you keep them within a specific range. For more information, see User-defined error constants.

Methods for the ErrorEvent type

Event	←	ErrorEvent
errorCode		reason
getTarget		setReason
isFirstTime		
isPreFilter		
isTargetSel		
setErrorCode		

User-defined error constants

ErrorEvent

You can define your own error constants within a specific range. Because the error constant range is subject to change in future versions of Paradox, ObjectPAL provides the IdRanges constants UserError and UserErrorMax. These constants represent the minimum and maximum values accepted for user-defined error constants.

To define ThisError and ThatError as constants, set values in a Const window as follows:

```
Const
    ThisError = 1
    ThatError = 2
EndConst
```

To use one of these constants, add it to UserError:

```
method error(var eventInfo ErrorEvent)
    if eventInfo.errorCode() = UserError + ThisError then
        doSomething()
    endIf
endMethod
```

By adding your own constant, a value above the minimum is guaranteed. To keep the value under the maximum, use the value of UserErrorMax. You can check the value with a **message** statement:

```
message(UserErrorMax)
```

In Paradox, the difference between UserError and UserErrorMax is 2046. This means the largest value that you can use for an error constant is UserError + 2046. The error code 0 is reserved to mean there is no error.

reason method**ErrorEvent**

Reports the cause of an error.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report the cause of an ErrorEvent. ObjectPAL provides the ErrorReasons constants for testing the value returned by **reason**.

Note

- Use `errorCode` to identify an error, and `reason` to identify the cause of an error.

Example

The following example shows code which should be attached to a form's built-in **error** method. 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
msgInfo("Error", eventInfo.errorCode())
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
```

setReason method**ErrorEvent**

Specifies a reason for generating an ErrorEvent.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies a reason for generating an ErrorEvent. This method takes an ErrorReasons constant as an argument.

Example

The following example creates an ErrorEvent, sets the reason to ErrorWarning, and 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)
```

errorCode method

```
ev.setReason(ErrorWarning) ; set the reason to ErrorWarning
thisForm.error(ev)         ; send the error to the form
endMethod
```

Event type

The Event type is the base type from which the other event types (e.g., ActionEvent) are derived. Many of the methods listed here are also used by other event types as derived methods.

The following built-in event methods are triggered by events:

- open
- close
- setFocus
- removeFocus
- newValue
- pushButton

Methods for the Event type

Event

errorCode
getTarget
isFirstTime
isPreFilter
isTargetSelf
reason
setErrorCode
setReason

errorCode method

Event

Reports the status of an error flag.

Syntax

```
errorCode ( ) SmallInt
```

Description

errorCode returns a nonzero error code if there is an error; otherwise, **errorCode** returns 0. To test for a specific error, use the ObjectPAL Errors constants (e.g., peDiskError) or a user-defined error constant. To create a list of the Error constants and the corresponding error messages, use enumRTLErrors.

Example

The following example assume that a form contains a field object, bound to the Quant field of the Orders table. When the field's value changes, this code executes the built-in code for this method and determines whether an error occurred.

```
; Quant::changeValue
method changeValue(var eventInfo ValueEvent)
doDefault
; check the event to see if it has an error
if eventInfo.errorCode() 0 then
```



```

        errorShow() ; Display the error message in a dialog box.
    endif
endMethod

```

getTarget method

Event

Creates a handle to the target of an event.

Syntax

```
getTarget ( var target UIObject )
```

Description

getTarget returns in target the handle of the UIObject that was the target of the most recent event. The target does not change as the event bubbles up the containership hierarchy.

Example

The following 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 form's **setFocus** method identifies the target of the event, determines if the target is a field, and changes the field's color to light blue. This provides a more dramatic visual cue than the normal highlight. The field's previous color is stored in the global variable called *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 form's Var window.

```

; thisForm::Var
Var
    oldFieldColor LongInt      ; to store the previous color of the field
endVar

```

The following code is attached to the form's **setFocus** method:

```

; 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

```

isFirstTime method

```
    ; 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
```

isFirstTime method

Event

Reports whether the form is handling an event for the first time before dispatching it.

Syntax

```
isFirstTime ( ) Logical
```

Description

isFirstTime 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 the form's built-in event methods.

Example

The following example uses **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.
```

```

        msgInfo("Status", "Dispatching a pushButton event to "
            + targObj.Name + ".")

    case NOT eventInfo.isTargetSelf() AND NOT eventInfo.isFirstTime() :
        ; The event has been explicitly bubbled back to the form.
        ; isPreFilter returns False.

        msgInfo("Status", "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 to the button, the button's **pushButton** method returns it to the form with the **passEvent** command. When the event returns to the form, the methods **isTargetSelf**, **isFirstTime**, and **isPreFilter** return False.

```

; testPassEvent::pushButton
method pushButton(var eventInfo Event)
passEvent    ; bubble the event up the hierarchy
endMethod

```

isPreFilter method

Event

Reports whether the form is handling an event for another object.

Syntax

```
isPreFilter ( ) Logical
```

Description

isPreFilter reports whether the form is handling an event for another object. This method returns True 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 external methods bubble back to the form, this method returns False. See About built-in methods for information about internal and external methods.

Note

- Form methods are not prefiltered. When an event occurs for the form, **isPreFilter** returns False.

Example

See the **getTarget** example.

isTargetSelf method

Event

Reports whether an object is the target of an event.

Syntax

```
isTargetSelf ( ) Logical
```

Description

isTargetSelf reports whether an object is the target of an event. Use **isTargetSelf** in the form's built-in event methods.

reason method

Example

See the **isFirstTime** example.

reason method

Event

Reports why an event occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why an event occurred. The return value depends on the event type. ObjectPAL provides the ValueReasons constants for testing the value returned by **reason**. ErrorReasons constants are defined for ErrorEvents, MenuReasons constants for MenuEvents, MoveReasons constants for MoveEvents, and StatusReasons constants for StatusEvents.

The **reason** method is valid for other event types, including ActionEvent, KeyEvent, MouseEvent, and ValueEvent, but returns a value of zero. **setReason** is also valid for ActionEvent, KeyEvent, MouseEvent, and ValueEvent, but can only be used to set user-defined Reason constants.

Example

The following example assumes 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. Assume also that the form is in Edit mode. The **newValue** method for Ship_VIA displays a message indicating why **newValue** was called. When the form opens, the Reason 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
```

setErrorCode method

Event

Sets the error code for an event.

Syntax

```
setErrorCode ( const errorId SmallInt )
```

Description

setErrorCode sets the error code for an event packet. If *errorId* is 0, it means there has been no error, and any non zero value for *errorId* indicates an error. To indicate a specific error, use an `EventErrorCodes` constant or a user-defined error constant.

Calling **setErrorCode** is not the same as calling `errorLog`, which adds error information directly to the error stack. **setErrorCode** adds the error code to the current event packet. This code may be added to the error stack, depending on how custom code and built-in code handles it.

Example

The following example creates an `ErrorEvent`, sets the reason to `ErrorWarning`, and 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
```

setReason method

Event

Specifies a reason for generating a move.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies in *reasonId* a reason for generating an event in an object's built-in **newValue** method, where *reasonId* is a `ValueReasons` constant.

Note

- `ErrorReasons` constants are defined for `ErrorEvents`, `MenuReasons` constants for `MenuEvents`, `MoveReasons` constants for `MoveEvents`, and `StatusReasons` constants for `StatusEvents`. See the entry for **setReason** in those sections for examples. **setReason** is also valid for `ActionEvent`, `KeyEvent`, `MouseEvent`, and `ValueEvent`, but can be used only to set user-defined Reason constants.

Example

The following example assumes 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 `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 example 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
```

FileSystem type

FileSystem variables provide access to and information about disk files, drives, and directories. They provide a handle, a variable you can use in ObjectPAL statements to work with a directory or a file. You can use **findFirst** to view available information and initialize the FileSystem variable.

The FileSystem type includes several derived methods from the AnyType type.

Methods for the FileSystem type

AnyType	←	FileSystem	
blank		accessRights	setWorkingDir
dataType		clearDirLock	shortName
isAssigned		copy	size
isBlank		delete	splitFullFileName
isFixedType		deleteDir	startUpDir
unAssign		drives	time
		EnumFileList	totalDiskSpace
		ExistDrive	totalDiskSpaceEx
		FindFirst	windowsDir
		FindNext	windowsSystemDir
		FreeDiskSpace	workingDir
		freeDiskSpaceEx	
		fullName	
		getDir	
		getDrive	
		getFileAccessRights	
		getValidFileExtensions	
		isDir	
		isFile	
		isFixed	
		isRemote	
		isRemovable	
		isValidDir	
		isValidFile	
		makeDir	
		name	
		privDir	
		rename	
		setDir	
		setDirLock	
		setDrive	
		setFileAccessRights	
		setPrivDir	

accessRights method**FileSystem**

Reports a file's access rights.

Syntax

```
accessRights ( ) String
```

Description

accessRights returns a string which describes the file's access rights. Access rights 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 **accessRights** returns an empty string, the file has no attributes set. You must use **findFirst** before using **accessRights**.

Example

Checks the attributes of the file MEMO14.TXT. Calls **findFirst** to ensure that the file exists and then calls **accessRights**. If the file is writable, calls Notepad so you can edit the file.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    fileName    String
    fs           FileSystem
endVar

fileName = "c:\\Corel\\Paradox\\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
```

clearDirLock procedure**FileSystem**

Unlocks a specified directory.

Syntax

```
clearDirLock ( const dirName String ) Logical
```

Description

clearDirLock removes a directory lock from the directory specified in *dirName*. This method returns True if it succeeds; otherwise, it returns False.

Example

See setDirLock example.

copy method**FileSystem**

Copies a file.

Syntax

```
copy ( const srcName String, const dstName String ) Logical
```

Description

copy returns True if successful in copying source file *srcName* to destination file *dstName*; otherwise, it returns False. If *dstName* exists, this method overwrites the file without asking for confirmation. This method copies only one file at a time and does not accept DOS wildcard characters.

Example

Searches the current directory for *sourceFile*. If *sourceFile* exists, copy creates a new file called *destFile*, which contains the original file's information.

```
; 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
```

delete method**FileSystem**

Deletes a file.

Syntax

```
delete ( const name String ) Logical
```

Description

Returns True if it deletes the specified file; otherwise, returns False. This method can delete only one file at a time and does not accept DOS wildcard characters.

Example 1

Displays a dialog box asking whether you want to delete *fileName*. If you choose Yes, 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
        msgInfo(fileName, "File not found.")
    endIf

endMethod

```

Example 2

Uses a **while** loop to delete files with the .OLD extension in the current directory.

```

; 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

```

deleteDir method

FileSystem

Deletes a directory, but only if the directory is empty (contains no files).

Syntax

```
deleteDir ( const name String ) Logical
```

Description

Returns True if successful in deleting the specified directory; otherwise, returns False. This method does not prompt for confirmation before deleting.

Example 1

Deletes the directory (folder) C:\DOS. If the C:\DOS folder contains files, deleteDir cannot delete it and an error message is displayed.

```

; 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

```

drives method

```
endIf
```

```
endMethod
```

In the following code, **enumFileList** checks whether the directory C:\SCAN\SUBSCAN is empty. If so, it creates an array containing one item (the directory name), and **deleteDir** deletes the directory:

```
; 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
```

Example 2

deleteDir deletes the directory (folder) C:\SCAN\SUBSCAN. Before the directory is deleted, **enumFileList** creates an array that contains the current directory and its parent directory.

```
; 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 has 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
```

drives method

FileSystem

Returns the letters of the drives attached to the system and known to Windows.

Syntax

```
drives ( ) String
```

Description

drives returns a string containing the letters of the drives that are attached to the system and known to Windows.

Example

Displays a dialog box listing the ID letters of the drives that are 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

```

enumFileList method

FileSystem

Lists information about files.

Syntax

1. enumFileList (const fileSpec String, var *arrayName* Array[] String)
2. enumFileList (const *fileSpec* String, const *tableName* String)

Description

enumFileList lists information about files that match the criteria specified in *fileSpec*. If *fileSpec* is **.**, the array or table includes records for the current directory (.) and the parent directory (..).

Syntax 1 writes data to the array *arrayName*, which you must declare before calling this method. The resulting array contains filenames and extensions, but does not contain paths.

Syntax 2 writes data to the table *tableName*. If the table does not exist, creates it automatically and enumerates the file list. If *tableName* does not specify a path, enumFileList creates the table in the working directory. If the table exists and is open, this method appends data to it; if the table is closed, overwrites its data.

The following table describes the structure of the table:

Field name	Type & size	Description
Name	Alpha 255	Filename (and extension)
Size	Numeric	File size in bytes
Attributes	Alpha 10	DOS file attributes
Date	Alpha 10	Date of last modification
Time	Alpha 10	Time of last modification

enumFileList lists filenames in the same order as the directory.

Example

Demonstrates both syntaxes of **enumFileList**. First, **enumFileList** searches the specified directory (folder) for forms and uses Syntax 1 to create an array of filenames, which is displayed in a pop-up menu. Then, **enumFileList** uses Syntax 2 to create a table of information about the files in a Table window.

```

; demoButton::pushButton
method pushButton(var eventInfo Event)
var
    fs  FileSystem
    formDir, theForm String

```

existDrive method

```
formNames Array[] String
tv tableView
p PopUpMenu
endVar

formDir = "C:\\Core1\\Paradox\\samples\\*.f?l"

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
```

existDrive method

FileSystem

Reports whether a drive is attached to the system.

Syntax

```
existDrive ( const driveLetter String ) Logical
```

Description

existDrive returns True if the specified drive is attached to the system; otherwise, it returns False. You can specify the drive using a letter (C) or a letter and a colon (C:).

Example

Calls **existDrive** to check whether drive P exists. If **existDrive** returns True, **setDrive** sets drive P as 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
```

findFirst method

FileSystem

Searches a file system for a filename.

Syntax

```
findFirst ( const pattern String ) Logical
```

Description

findFirst returns True if a file is found whose name matches *pattern*; 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 check whether a file or directory exists and to initialize a FileSystem variable before calling another FileSystem method or procedure. You must fully qualify **findFirst** calls to other than the current default drive or path, unless you reset the default drive and path with the **setDir** method.

Under Windows 95 and Windows 98, **findFirst** also finds the 8.3 format of the filename that exists in the file system for long filenames.

Note

- **findFirst** finds file and directory names in the order that they're listed in the directory. The first value returned by **findFirst** depends on the path and file specification.

Example

The following 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 folder for a file
; or folder named COREL\PARADOX.
if fs.findFirst("c:\\Corel\\Paradox") then
    ; this displays COREL\PARADOX (findFirst finds the folder)
    msgInfo("Pattern: c:\\Corel\\Paradox", "Name: " + fs.name())
else
    errorShow()
endif

; >INVALID PATTERN CAUSES AN ERROR!! <
if fs.findFirst("c:\\Corel\\Paradox\\") then
    message("This message never displays.")
else
    errorShow("Invalid pattern: c:\\Corel\\Paradox\\")
endif

; Search in the COREL\PARADOX folder for
; any file or folder.
if fs.findFirst("c:\\Corel\\Paradox\\*.*)") then
    ; This displays one dot (.) because the
    ; first file in a directory is a single dot (.).
    msgInfo("Pattern: c:\\Corel\\Paradox\\*.*)", "Name: " + fs.name())
else
    errorShow()
endif

endmethod
```

findNext method**FileSystem**

Searches a file system for multiple instances of a filename.

Syntax

```
findNext ( [ const fileSpec String ] ) Logical
```

Description

After **findFirst** succeeds, **findNext** searches for the next file whose name matches Pattern. **findNext** returns True if successful; otherwise, it returns False.

You can also use the optional argument *fileSpec* to specify a path and file specification. If you do, the call to **findFirst** is unnecessary.

Example 1

The following example calls **findNext** to fill a list with the names of the tables in the current directory (folder). The example assumes that a field displayed as a drop-down list has already been placed in the form. The code is attached to the built-in **open** method of the list object contained by the field object.

```
; tablesFld.listObj::open
method open(var eventInfo Event)
var
    fs FileSystem
endVar

doDefault

; This while loop fills the list in the drop-down edit
; box with *.db files in the default sample directory
while fs.findNext("c:\\Corel\\Paradox\\samples\\*.db")
    self.list.selection =
        self.list.selection + 1
    self.list.value = fs.name()
endWhile
endMethod
```

Example 2

The following example uses **findNext** with a file specification as an argument and displays a pop-up menu listing the files in the C:\COREL\PARADOX directory (folder):

```
; editText::pushButton
method pushButton(var eventInfo Event)
var
    fs      FileSystem
    p      PopUpMenu
    choice  String
endVar

; search for *.txt files in the COREL\PARADOX directory
; then add their names to a pop-up menu
while fs.findNext("c:\\Corel\\Paradox\\*.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
```

freeDiskSpace method**FileSystem**

Returns the amount of free space on a drive, measured in bytes.

Syntax

```
freeDiskSpace ( const driveLetter String ) LongInt
```

Description

freeDiskSpace returns the number of bytes available on a specified drive. You can specify the drive using a letter (C) or a letter and a colon (C:).

Note

- This method will fail if the specified drive doesn't exist.

Example 1

The following 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
```

Example 2

The following example compares the size of the file MEMO14.TXT with 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
    stDrive     String
    liFileSize,
    liFreeSpace LongInt
    dyFileInfo  DynArray[] String
endVar

if fs.findFirst(":WORK:memo14.txt") then
    liFileSize = fs.size()
    splitFullFileName(workingDir(), dyFileInfo)
    stDrive = dyFileInfo["DRIVE"]
    liFreeSpace = fs.freeDiskSpace(stDrive)
else
    msgStop("MEMO14.TXT", "File not found.")
    return
endif

if liFreeSpace > liFileSize then
    fs.copy("memo14.txt", "memo14.bak")
    message("File copied successfully.")
else
    msgStop("Copy", "Not enough disk space to copy file.")
endif

endMethod
```

freeDiskSpaceEx method**FileSystem**

Returns the amount of free space on a drive, measured in kilobytes.

Syntax

```
freeDiskSpaceEx ( const driveLetter String ) LongInt
```

Description

freeDiskSpaceEx returns the number of kilobytes available on a specified drive. You can specify the drive using a letter (C) or a letter and a colon (C:).

Note

- This method will fail if the specified drive doesn't exist.

Example

The following example displays a dialog box listing the number of kilobytes available on drive C.

```
method run(event eventinfo)
var
    L longint
    FS filesystem
endvar
L = FS.freeDiskSpace("C")
msginfo("Free Disk Space", "Drive C is " + string(L) + " Kilobytes.")
endmethod
```

fullName method/procedure**FileSystem**

Returns the full path to a file.

Syntax

1. (Method) **fullName** () String
2. (Procedure) **fullName** (const *fileName* String) String

Description

In Syntax 1, after a successful **findFirst** or **findNext**, **fullName** returns the full path of the found file. Use this method with **splitFullName** to analyze the components of a filename.

Syntax 2 operates on a filename, expanding or translating aliases and returning the expanded string. For example, if the working directory (:WORK:) is defined as C:\COREL\PARADOX\FORMS. Given the string :WORK:myForm.fsl Syntax 2 returns C:\COREL\PARADOX\FORMS\myForm.fsl.

Example

The following example calls **fullName** to get the full name of the first form listed in the current directory. The code then calls **splitFullName** to split the name into its component parts and store them in a dynamic array. Finally, the code calls view to display the dynamic array.

```
; 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:\\Corel\\Paradox\\samples\\customer.db") then
```



```

; store the full filename to a variable
fullFileName = fs.fullName()

; split filename into parts and store them in a DynArray
splitFullFileName(fullFileName, splitName)

; display the component parts
splitName.view("Split name")
endIf

endMethod

```

getDir method

FileSystem

Returns the path to which the FileSystem variable points.

Syntax

```
getDir ( ) String
```

Description

getDir returns a string that represents the path to which the FileSystem variable points. You can use **setDir** to make a FileSystem variable point to a specified directory. To get a drive letter, use **getDrive**.

Example

The following example gets the path of the directory to which the FileSystem variable points, and compares it with a path. If the directories don't match, **getDir** calls **setDir** to change the directory.

```

; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    fs  FileSystem
    st  String
endVar

    st = "c:\\Corel\\Paradox\\myforms"

    if fs.getDir() st then
        fs.setDir(st)
    endIf
endMethod

```

getDrive method

FileSystem

Returns the drive letter or alias that the FileSystem variable points to.

Syntax

```
getDrive ( ) String
```

Description

getDrive returns a string representing the drive letter or alias that the FileSystem variable points to.

Example

The following example calls **getDrive** to return the alias of the working directory. The code then sets the default drive to H and calls **getDrive** again to confirm the change.

```

; setH::pushButton
method pushButton(var eventInfo Event)
var

```

getFileAccessRights procedure

```

    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
```

getFileAccessRights procedure

FileSystem

Reports a file's access rights.

Syntax

```
getFileAccessRights ( const fileName String ) String
```

Description

getFileAccessRights returns a string that describes the access rights of a file. The 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 **getFileAccessRights** returns an empty string, the file has no attributes set.

Example

The following example displays the file attributes for C:\CONFIG.SYS.

```

; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  fileName String
endVar

fileName = "C:\\CONFIG.SYS"

msgInfo(fileName, getFileAccessRights(fileName))

endMethod
```

getValidFileExtensions procedure

FileSystem

Returns the valid file extensions for a specified object.

Syntax

```
getValidFileExtensions ( const objectType String ) String
```

Description

getValidFileExtensions returns a string containing the valid file extensions for the object specified in **objectType**, which is a Form, Library, Report, or Script.

Example

The following 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
```

isAssigned method**FileSystem**

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just FileSystem.

Example

The following 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 is attached 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
```

isDir procedure**FileSystem**

Reports whether a specified string represents the name of a directory.

Syntax

```
isDir ( const dirName String ) Logical
```

isFile procedure

Description

isDir returns True if **dirName** is a valid directory name; otherwise, it returns False.

Example

The following example calls **isDir** to ensure that the directory (folder) specified by the variable *newDir* is valid. If so, the code calls **setDir** to make *newDir* the default directory. In this example, the value of *newDir* is hard coded, but it can also be supplied by the user, read from a table, or extracted from another source.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    fs      FileSystem
    newDir  String
endVar

newDir = "C:\\Core1\\Paradox\\diveplan"
if isDir(newDir) then
    fs.setDir(newDir)
    msgInfo("Current directory", fs.getDir())
else
    msgStop(newDir, "Directory does not exist.")
endIf

endMethod
```

isFile procedure

FileSystem

Reports whether a specified string is a filename in the active file system.

Syntax

```
isFile ( const fileName String ) Logical
```

Description

isFile returns True if *fileName* is a file in the current file system; otherwise, it returns False.

Example 1

The following 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\\MyXFilex.ext")) ; displays False
sleep(1500)

endMethod
```

Example 2

The following example asks for the full path and filename of a file to delete. The code calls **isFile** to test whether the file exists, and then calls **delete** to delete 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

```

isFixed method

FileSystem

Reports whether a drive is fixed (not removable or networked).

Syntax

```
isFixed ( const driveLetter String ) Logical
```

Description

isFixed returns True if the specified drive represents a fixed drive; otherwise, it returns False. You can specify the drive 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

```

isRemote method

FileSystem

Reports whether a drive is a remote (network) drive.

Syntax

```
isRemote ( const driveLetter String ) Logical
```

Description

isRemote returns True if the specified drive represents a remote (network) drive; otherwise, it returns False. You can specify the drive using a letter (C) or a letter and a colon (C:).

Example

The following example calls **existDrive** to ensure drive H is attached and then calls **isRemote** to determine whether drive H is a network drive.

isRemovable method

```
var
  h FileSystem
endVar
if h.existDrive("h") then ; if drive H is attached
  if h.isRemote("h") then
    msgInfo("Drive H: ", "Remote Drive")
  else
    msgInfo("Drive H:", "Not a Remote Drive.")
  endIf
else
  msgStop("Drive H", "Drive is not attached.")
endIf
```

isRemovable method

FileSystem

Reports whether a drive is removable.

Syntax

```
isRemovable ( const driveLetter String ) Logical
```

Description

isRemovable returns True if the specified drive is a removable drive; otherwise, it returns False. You can specify the drive using a letter (C) or a letter and a colon (C:).

Example

The following example calls **existDrive** to ensure drive D is attached, then calls **isRemovable** to determine whether 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
```

isValidDir procedure

FileSystem

Checks whether a directory name is valid.

Syntax

```
isValidDir ( const dirName String ) Logical
```

Description

isValidDir checks whether the directory name is valid for the file system. Use **isValidDir** to see if long filenames are supported on a specific volume. This procedure returns True if the directory is valid; otherwise, it returns False.

Use the **isValidFile** method to check the validity of the entire path.

Example

See the **isValidFile** example

isValidFile procedure**FileSystem**

Checks whether a filename is valid.

Syntax

```
isValidFile ( const fileName String ) Logical
```

Description

isValidFile checks whether the filename is valid for the file system. Use **isValidFile** to see if long filenames are supported on a specific volume. This procedure returns True if the file is valid; otherwise it returns false.

Example

The following example uses the view dialog to request a new filename. **isValidFile** is used to check whether the file is valid for the volume so that it can be copied to that volume.

```
proc copyNewFile( origFileName String )
var
    newFile string
endVar

newFile.view()

if isValidFile( newFile ) then
    copy( origFileName, newFile )
else
    msgInfo( "Error", "This is not a valid filename" )
endif

endProc
```

makeDir method**FileSystem**

Creates a new directory.

Syntax

```
makeDir ( const name String ) Logical
```

Description

makeDir creates all directories and subdirectories specified in *name*. **MakeDir** returns True if successful in creating *name* (or if the directory already exists); otherwise, it returns False.

Example

The following example tries to create a new directory (folder) on drive C, and displays a dialog box to report success or failure.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    fs          FileSystem
    returnValue Logical
endVar
```

name method

```
; this creates \New and \New\Directory etc...
returnValue = fs.makeDir("C:\\New\\Directory\\Tree")

msgInfo("Status", iif(returnValue, "New directory created", "makeDir Failed"))

endMethod
```

name method

FileSystem

Returns a filename.

Syntax

name () String

Description

After a successful **findFirst** or **findNext**, **name** returns the filename that matches the pattern.

Example

The following example calls **findFirst** and **findNext** to find the tables in the current directory and then calls **name** to create a pop-up menu listing the filenames.

```
; 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 filenames 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
```

privDir procedure

FileSystem

Returns the name of the private directory.

Syntax

privDir () String

Description

privDir returns a string containing the full DOS path (including the drive letter) of the private directory.

Each user must have a private directory that stores temporary tables. The private directory can be on a network or on a local drive. Use **setPrivDir** to set the path to the private directory.

Example

The following example calls `privDir` to display the path to the private directory (:PRIV:) in the Status Bar.

```
method pushButton(var eventInfo Event)
    message("Your private directory is: ", privDir())
endMethod
```

rename method**FileSystem**

Renames a file.

Syntax

```
rename ( const oldName String, const newName String ) Logical
```

Description

rename changes the name of the file *oldName* to *newName*. If *newName* is used by another file, the method does not overwrite the existing file. The **rename** method returns True if successful; otherwise, it returns False. **rename** is independent of **findFirst** and **findLast**.

Example

The following example searches the current directory for the file specified in the *oldName* variable. If the file exists, the example calls **rename** to rename it. A dialog box reports any errors.

```
; 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
```

setDir method**FileSystem**

Sets the directory path for a FileSystem variable.

Syntax

```
setDir ( const name String ) Logical
```

Description

setDir sets the path to *name* for a FileSystem variable. Use **setDrive** to set the default drive.

Example

The following example calls **isDir** to check whether the directory *newDir* is valid. If the directory is valid, the code calls **setDir** to set *newDir* as the default directory.

setDirLock procedure

```
method pushButton(var eventInfo Event)
var
    fs      FileSystem
    newDir  String
endVar

newDir = "c:\\Corel\\Paradox\\mine\\zap"

if isDir(newDir) then
    fs.setDir(newDir)
else
    msgStop(newDir, "Not a valid directory.")
endif

message(fs.getDir()) ; displays \\Corel\\Paradox\\mine\\zap
endMethod
```

setDirLock procedure

FileSystem

Locks a specified directory.

Syntax

```
setDirLock ( const dirName String ) Logical
```

Description

setDirLock locks the directory *dirName*. The code returns True if successful; otherwise, it returns False.

A directory lock makes the directory read-only. This prevents Paradox from reading from or writing to a lock file in that directory. A directory lock is required for Paradox to access data from a CD-ROM drive, and can improve performance on network drives and local drives. A lock is not respected on a local drive if Local Share is turned off.

Example

The following example calls **setDirLock** to make a network drive read-only when the form opens, and calls **clearDirLock** to remove the lock when the form closes.

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

```
method open(var eventInfo Event)
var
    h FileSystem
endVar

if eventInfo.isPreFilter() then
    ; This code executes for each object on the form:
else
    ; This code executes only for the form:
    if h.existDrive("h") then ; if drive H is attached
        if h.isRemote("h") then
            setDirLock("h")
            message("Drive H: locked.")
        else
            msgStop("Drive H:", "Not a Remote Drive.")
            return
        endif
    else
        msgStop("Drive H:", "Drive is not attached.")
        return
    endif
endif
```

```

        endIf
    endIf
endMethod

```

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

```

method close(var eventInfo Event)

    var
        h FileSystem
    endVar

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:
        if h.existDrive("h") then ; if drive H is attached
            if h.isRemote("h") then
                clearDirLock("h")
                message("Drive H: unlocked.")
            else
                msgStop("Drive H:", "Not a Remote Drive.")
                return
            endif
        else
            msgStop("Drive H:", "Drive is not attached.")
            return
        endif
    endif

endIf

endMethod

```

setDrive method

FileSystem

Sets a specified drive as the default drive.

Syntax

```
setDrive ( const name String ) Logical
```

Description

setDrive sets the specified drive as the default. The method returns True if successful; otherwise, it returns False. You can specify the drive with a letter (C), a letter and a colon (C:), or an alias (e.g., :MAST:).

Example 1

The following example calls **view**, cast for the String type, to display a dialog box and ask for input. If you type a valid drive letter, the code calls **setDrive** to set the specified drive as the default.

```

; 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

```

Example 2

Shows how to use an alias with **setDrive**. This example assumes that the alias (:MAST:) has already been defined.

```

; setDrive::pushButton
method pushButton(var eventInfo Event)
var
    fs FileSystem
endVar

fs.setDrive(":MAST:")

endMethod

```

setFileAccessRights procedure

FileSystem

Sets a file's access rights.

Syntax

```
setFileAccessRights ( const fileName String, const rights String ) Logical
```

Description

setFileAccessRights sets the access rights of a specified file to those specified in *rights*. Rights is a string that contains one or more of the following: A, D, H, R, S, V (for archive, directory, hidden, read-only, system, and volume, respectively). If *rights* is an empty string (""), **setFileAccessRights** removes all access rights settings for the specified file. You don't have to declare a FileSystem variable (or use the **findFirst** method) before calling **setFileAccessRights**.

Example

The following example sets the file access rights 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

```

setPrivDir procedure

FileSystem

Sets or changes the private directory.

Syntax

```
setPrivDir ( const path String ) Logical
```

Description

setPrivDir sets a path to the current private directory. **setPrivDir** returns True if successful; otherwise, it returns False. The following table displays valid path values.

Value of path	Example
Directory name	ORDERS
Full path	C:\COREL\PARADOX\APPS\ORDERS\
Relative path	..\..\ORDERS
Alias	:ORDERS:

Paradox closes all of its open windows and frees all locks before setting the private directory. Therefore, **setPrivDir** does not take effect until all ObjectPAL code has finished executing. You can keep a form open by adding code to its built-in **menuAction** method to trap for the MenuChangingPriv menu command (see the example for details). If you do so, save any documents that need saving before changing the working directory. **setPrivDir** returns True if successful; otherwise, it returns False.

ObjectPAL provides the following MenuCommands constants for handling changes to the private directory:

Constant	Description
MenuFilePrivateDir	Issued when the user chooses Tools, Settings, Preferences, Database, Private Directory from the Paradox menu. Trap for this constant to prevent the user from changing the private directory.
MenuChangingPriv	Issued just before the private directory changes. Trap for this constant to keep a form open when changing the private directory.
MenuChangedPriv	Issued just after the private directory changes. Trap for this constant to find out when the private directory has changed.

ObjectPAL also provides the constant MenuFileWorkingDir, issued when the user clicks Tools, Settings, Preferences, Database, Working Directory.

Example I

The following example changes the private directory and the resulting menu commands generated by Paradox. When you click Tools, Settings, Preferences, Database, Private Directory the code calls **disableDefault** to block the default behavior. This prevents Paradox from displaying the Set Private Directory dialog box and then tests the value of a Logical variable *okToChangePriv* (declared and assigned elsewhere). If *okToChangePriv* is True, the code calls **setPrivDir** to set the private directory (:PRIV:) behind the scenes.

This example also handles the MenuChangingPriv menu command, issued by Paradox just before it changes the private directory. **setErrorCode** sets the error code to a nonzero value, which keeps this

setPrivDir procedure

form open when the private directory changes. The code responds to the MenuChangedPriv menu command, issued by Paradox just after it changes the private directory.

```
method menuAction(var eventInfo MenuEvent)

const
    kKeepFormOpen = UserMenu ; UserMenu is an ObjectPAL constant.
endConst
    ; In a real app you'd declare and assign this variable elsewhere.
    okToChangePriv = True

switch
    case eventInfo.id() = MenuFilePrivateDir :
        disableDefault ; Block the default behavior.
        if okToChangePriv then
            setPrivDir("c:\\pdx\\mine") ; Set :PRIV: to hard-coded path.
        else
            return
        endif

    case eventInfo.id() = MenuChangingPriv :
        eventInfo.setErrorCode(kKeepFormOpen)

    case eventInfo.id() = MenuChangedPriv :
        ; You may want to take some action after changing :PRIV:.
        ; This example just displays the new path.
        message(privDir())
        sleep(1000)

    otherwise : doDefault
endSwitch
endMethod
```

Example 2

The following example uses the **open** and the **menuAction** methods of a form to set the private directory before the form opens. In the form's built-in **open** method, **setPrivDir** changes the private directory to the same directory as the form. The ObjectPAL code in the **menuAction** prevents the form from closing during the change.

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

```
;frm1 :: open
method open(var eventInfo Event)
    var
        f          Form
        dynPath    DynArray[] String
    endVar

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:
        f.attach()
        splitFullFileName(f.getFileName(), dynPath)
        setPrivDir(dynPath["Drive"] + dynPath["Path"])
    else
        ; This code executes only for the form:
    endif
endMethod
```

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

```

;frm1 :: menuAction
method menuAction(var eventInfo MenuEvent)
const
    kKeepFormOpen = UserMenu    ; UserMenu is an ObjectPAL constant.
endConst
    ; Any nonzero value keeps the form open.

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:
        if eventInfo.id() = MenuChangingPriv then
            eventInfo.setErrorCode(kKeepFormOpen)
        endIf
    else
        ; This code executes only for the form:
    endIf
endMethod

```

setWorkingDir procedure

FileSystem

Sets the working directory.

Syntax

```
setWorkingDir ( const path String ) Logical
```

Description

setWorkingDir sets the path of the current working directory. The following table gives examples of valid values for path:

Value of path	Example
Directory name	ORDERS
Full path	C:\COREL\PARADOX\APPS\ORDERS\
Relative path	..\..\ORDERS
Alias	:ORDERS:

By default, Paradox closes all open windows before setting the working directory, and prompts you to save modified documents. Therefore, **setWorkingDir** does not take effect until all ObjectPAL code executes. You can keep a form open by adding code to its built-in **menuAction** method to trap for the MenuChangingWork menu command. If you do so, save any active documents before changing the working directory.

Use the following ObjectPAL MenuCommands constants to handle changes to the working folder:

Constant	Description
MenuFileWorkingDir	Issued when the user clicks Tools, Settings, Preferences, Database, Working Directory. Trap for this constant to prevent the user from changing the working directory.
MenuChangingWork	Issued before the working directory changes. Trap for this constant to keep a form open when changing the working directory.
MenuChangedWork	Issued after the working directory changes. Trap for this constant to determine whether the working directory has changed.

setWorkingDir procedure

ObjectPAL also provides the constant `MenuFilePrivateDir`, issued when the user clicks Tools, Settings, Preferences, Database, Private Directory.

Example 1

The following example uses a menu command to change the working directory, and the resulting menu commands. When you click Tools, Settings, Preferences, Database, Working Directory, the code calls **disableDefault** to block the default behavior and prevent Paradox from displaying the Set Working Directory dialog box. Next, this code tests the value of a Logical variable *okToChangeWork* (declared and assigned elsewhere). If *okToChangeWork* is True, it calls **setWorkingDir** to set the working directory (:WORK:) behind the scenes.

Also handles the `MenuChangingWork` menu command, issued by Paradox just before it changes the working directory. The call to **setErrorCode** sets the error code to a nonzero value, which keeps the form open when the working directory changes. The code in this example responds to the `MenuChangedWork` menu command, issued by Paradox just after it changes the working directory.

```
method menuAction(var eventInfo MenuEvent)
const
    kKeepFormOpen = UserMenu    ; UserMenu is an ObjectPAL constant.
endConst
                                ; Any nonzero value keeps the form open.

    ; In a real app you'd declare and assign this variable elsewhere.
    okToChangeWork = True

switch
    case eventInfo.id() = MenuFileWorkingDir :
        disableDefault                ; Block the default behavior.
        if okToChangeWork then
            setWorkingDir("c:\pdx\mine") ; Set :WORK: to hard-coded path.
        else
            return
        endif

    case eventInfo.id() = MenuChangingWork :
        eventInfo.setErrorCode(kKeepFormOpen)

    case eventInfo.id() = MenuChangedWork :
        ; You may want to take some action after changing :WORK:.
        ; This example just displays the new path.
        message(workingDir())
        sleep(1000)

    otherwise : doDefault
endSwitch
endMethod
```

Example 2

The following example uses a form's **open** and **menuAction** methods to set the working directory before the form opens. In the form's built-in **open** method, `setWorkingDir` changes the current working directory to the same directory as the form. The ObjectPAL code in the `menuAction` prevents the form from closing during the change.

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

```
;frm1 :: open
method open(var eventInfo Event)
var
    f                Form
    dynPath    DynArray[] String
endVar
```



```

if eventInfo.isPreFilter() then
    ; This code executes for each object on the form:
else
    ; This code executes only for the form:
    f.attach()
    splitFullFileName(f.getFileName(), dynPath)
    setWorkingDir(dynPath["Drive"] + dynPath["Path"])
endIf

endMethod

```

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

```

;frm1 :: menuAction
method menuAction(var eventInfo MenuEvent)
const
    kKeepFormOpen = UserMenu    ; UserMenu is an ObjectPAL constant.
endConst                ; Any nonzero value keeps the form open.

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:
        if eventInfo.id() = MenuChangingWork then
            eventInfo.setErrorCode(kKeepFormOpen)
        endIf
    else
        ; This code executes only for the form:
    endIf
endMethod

```

shortName method

FileSystem

Returns the short name of a file.

Syntax

```
shortName ( ) String
```

Description

After a successful **findFirst** or **findNext**, **shortName** returns the short name of the file whose name matches the pattern. A short name is the 8.3 filename stored in the file system.

Example

The following example calls **findFirst** and **findNext** to locate tables in the current directory and then calls **shortName** to create a pop-up menu listing the filenames.

```

; 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.shortName())   ; use filenames in pop-up
    while fs.findNext()
        p.addText(fs.shortName())
    endWhile
endMethod

```

size method

```
        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
```

size method

FileSystem

Returns the size of a file.

Syntax

size () LongInt

Description

size returns the size of a file, measured in bytes, after a successful **findFirst** or **findNext**.

Example

The following example creates a dynamic array containing the filenames 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
```

splitFullFileName procedure

FileSystem

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

splitFullFileName divides a full path (obtained using **fullName**) into its component parts.

SplitFullFileName does not return the values directly, but assigns them to variables that you declare and pass as arguments.

Syntax 1 assigns the returned values to a dynamic array that you must declare and pass as an argument. The DynArray has the following keys: DRIVE, PATH, NAME, and EXT.

Syntax 2 assigns the returned values to four String variables that you must declare and pass as arguments.

With both syntaxes, path components can include colons, periods, slashes, and backslashes. For example, if given C:\COREL\PARADOX\FORMS\ORDERS.FSL, **splitFullName** assigns values as follows:

DRIVE = C:

PATH = \COREL\PARADOX\FORMS\

NAME = ORDERS, and EXT = .FSL

The DRIVE variable (or key) stores everything up to and including the last colon in the filename. If the filename includes an alias, the alias is assigned to DRIVE.

The PATH variable (or key) stores everything following the drive, up to and including the last backslash or slash. If a directory name in the path includes an extension, it is included in the PATH variable.

The NAME variable (or key) stores everything following the path, up to but not including the period that separates a filename from its extension. If the filename does not include a name, an empty string is assigned to the NAME variable.

The EXT variable (or key) stores everything following the filename, including the last period. If the filename does not include an extension, an empty string is assigned to the EXT variable.

Note

- The extension must be registered in the HKEY_CLASSES_ROOT section of the system registry to be return a value in this field. For more information about HKEY_CLASSES_ROOT, see your Windows documentation.

Example 1

The following example calls **fullName** to return the full name of the first form listed in the current directory. Then calls **splitFullName** to split the name into its component parts and store them in a dynamic array. The call to **view** displays the dynamic array.

```
; 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:\\Corel\\Paradox\\samples\\customer.db") then

    ; store the full filename to a variable
    fullFileName = fs.fullName()

    ; split filename into parts and store them in a DynArray
    splitFullName(fullFileName, splitName)

    ; display the component parts
    splitName.view("Split name")
endif

endMethod
```

startUpDir procedure

Example 2

The following example calls **splitFullFileName** to split the full name of a form into its component parts, then displays the path and the filename (without an extension) in dialog boxes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    fs FileSystem
    driveName, pathName, fileName, extName String
endVar

    splitFullFileName("c:\\data\\sales\\stats.fsl", driveName, pathName, fileName,
extName)
    pathName.view("Path name") ; displays the path
    fileName.view("Filename") ; displays the filename (no extension)

endMethod
```

Example 3

The following example displays a dialog box and prompts you to enter a filename. **splitFullFileName** splits the filename into its component parts and then displays the parts in dialog boxes.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    stTestFileName,
    stPrompt,
    stDrive,
    stPath,
    stName,
    stExt      String
    dyFileName DynArray[] String
endVar

    stPrompt = "Enter a filename here."
    stTestFileName = stPrompt

    stTestFileName.view("Enter a filename to split:")

    if stTestFileName = stPrompt then
        ; User closed the dialog box without clicking OK,
        ; or clicked OK without typing a value.
        return
    else
        ; User typed a value and clicked OK.
        splitFullFileName(stTestFileName, dyFileName)
        dyFileName.view("DynArray")

        splitFullFileName(stTestFileName, stDrive, stPath, stName, stExt)
        stDrive.view("Drive")
        stPath.view("Path")
        stName.view("Name")
        stExt.view("Ext")
    endIf
endMethod
```

startUpDir procedure

FileSystem

Returns a string containing the path to your start-up directory (folder).

Syntax

```
startUpDir ( ) String
```

Description

startUpDir returns a string containing the path (including the drive letter) of the Paradox start-up directory (folder).

Example

The following example opens a dialog box that displays the path to the Paradox start-up directory (folder).

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
msgInfo("Start-up directory", startUpDir())
endMethod
```

time method**FileSystem**

Returns the time and date of a file's last modification after a successful findFirst or findNext.

Syntax

```
time ( ) DateTime
```

Description

time returns a DateTime value that represents the time and date of the file's last modification.

Example

The following example calls **time** to return the time and date of the most recent modification to the *Customer* table. The code then compares the modification date with today's date and reports 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
endIf
endMethod
```

totalDiskSpace method**FileSystem**

Returns the total capacity of a specified drive, measured in bytes.

Syntax

```
totalDiskSpace ( const driveLetter String ) LongInt
```

Description

totalDiskSpace returns the total number of bytes the specified drive can hold. You can specify a drive using a letter (C) or a letter and a colon (C:).

totalDiskSpaceEx method

Note

- This method will fail if the specified drive doesn't exist.

Example

The following example calls **totalDiskSpace** and **freeDiskSpace** to calculate the amount of space available. The code stores the information in a dynamic array and then calls the appropriate **view** method 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
```

totalDiskSpaceEx method

FileSystem

Returns the total capacity of a specified drive, measured in kilobytes.

Syntax

```
totalDiskSpaceEx ( const driveLetter String ) LongInt
```

Description

totalDiskSpaceEx returns the total number of kilobytes the specified drive can hold. You can specify a drive using a letter (C) or a letter and a colon (C:).

Note

- This method will fail if the specified drive doesn't exist.

Example

The following example uses **totalDiskSpaceEx** to calculate the total amount of disk space and then displays the information.

```
var
    L longint
    FS filesystem
endvar

L = FS.totalDiskSpace("C")
msginfo("Total Disk Space", "Drive C is " + string(L) + " Kilobytes.")

endmethod
```

windowsDir procedure

FileSystem

Returns the path to the WINDOWS directory (folder).

Syntax

```
windowsDir ( ) String
```

Description

windowsDir returns the path to the WINDOWS directory.

Example

The following example reads WIN.INI from drive B and copies it to the WINDOWS folder 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
```

windowsSystemDir procedure**FileSystem**

Returns the path to the Windows System directory (folder).

Syntax

```
windowsSystemDir ( ) String
```

Description

windowsSystemDir returns the path to the Windows System directory.

Example

The following example reads SPECIAL.DRV from drive B and copies it to the Windows System directory (folder) 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
```

workingDir procedure**FileSystem**

Returns the name of the working directory.

Syntax

```
workingDir ( ) String
```

Description

workingDir returns the name (including the path) of the working directory.

Example

The following example displays a message that contains the path to the working directory:

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

message("Working directory is: " + workingDir())

endMethod
```

Disk errors**FileSystem**

When a method fails because of a disk error, the error code constant is `peDiskError` and the error message is, "A disk error occurred" plus one of the following strings:

- "Invalid function number."
- "The file could not be found."
- "The directory path could not be found."
- "No file handle available."
- "Access to this file is denied. It is read only or a directory."
- "Invalid handle."
- "Memory control blocks have been damaged."
- "Insufficient memory to allocate file structures."
- "Invalid memory block address."
- "Invalid environment."
- "Invalid format."
- "Invalid file access byte."
- "Invalid data."
- "Invalid drive."
- "Cannot remove the current directory."
- "Not the same device."
- "No more files match the wildcard specification."
- "Cannot write to a write-protected disk."
- "Unknown unit."
- "The drive is not ready."
- "Command is not recognized."
- "Checksum error (Bad CRC)."
- "Invalid request structure length."
- "File seek error."

- “Unknown media type.”
- “Sector not found.”
- “Out of paper.”
- “An error occurred while trying to write to the disk.”
- “An error occurred while trying to read from the disk.”
- “General DOS error.”
- “File sharing violation.”
- “File lock violation.”
- “Invalid disk change.”
- “File control blocks unavailable.”
- “Sharing buffer overflow.”
- “Bad code page.”
- “Handle EOF.”
- “The disk is full.”
- “Device is not supported.”
- “Device is not listening.”
- “Duplicate name.”
- “Invalid network path.”
- “The network is busy.”
- “The device does not exist.”
- “Too many commands.”
- “Adapter error.”
- “Invalid network response.”
- “Network error.”
- “Adapter is incompatible.”
- “The print queue is full.”
- “Out of spool space.”
- “Print job was canceled.”
- “The network name was deleted.”
- “Your access to the network is denied.”
- “Invalid device type.”
- “Invalid network name.”
- “Too many names.”
- “Too many sessions.”
- “Sharing pause.”
- “Request not accepted.”
- “Redirection pause.”
- “The file already exists.”
- “Duplicate file control blocks.”

Disk errors

- “Cannot create the specified directory.”
- “DOS critical error.”
- “Out of structures. Cannot perform operation.”
- “Drive is already assigned.”
- “Invalid password.”
- “Invalid parameter.”
- “Network write error.”
- “Comp command is not loaded.”
- “The mode specification is invalid.”
- “Cannot write to the file because it was opened in read-only mode.”

Form type

A Form variable provides a handle for working with a Paradox form. Form type methods let you

- load a form in a Form Design window and save a design
- open and close a form
- attach to an open form
- work with tables in a data model
- work with table aliases
- 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, such as a **mouseUp** or **keyPhysical**
- get and set methods for a form

The Form type is the base type from which the other display manager types (for example, Report) are derived. Many of the methods listed in this section are also used by the Application, Report, and TableView types.

The Form type includes several derived methods from the AnyType type.

Methods for the Form type

AnyType	←	Form	
blank	action	getStyleSheet	selectCurrentTool
dataType	attach	getTitle	setCompileWithDebug
isAssigned	bringToTop	hide	setIcon
isBlank	close	hideToolbar	setMenu
isFixedType	create	isCompileWithDebug	setPosition
unAssign	delayScreenUpdates	isDesign	setProtoProperty
	deliver	isMaximized	setSelectedObjects
	design	isMinimized	setStyleSheet
	disableBreakMessage	isToolbarShowing	setTitle
	disablePreviousError	isVisible	show
	dmAddTable	keyChar	showToolbar

dmAttach	keyPhysical	wait
dmBuildQueryString	load	windowClientHandle
dmEnumLinkFields	maximize	windowHandle
dmGet	menuAction	writeText
dmGetProperty	methodDelete	
dmHasTable	methodEdit	
dmLinkToFields	methodGet	
dmLinkToIndex	methodSet	
dmPut	minimize	
dmRemoveTable	mouseDouble	
dmResync	mouseDown	
dmSetProperty	mouseEnter	
dmUnlink	mouseExit	
enumDataModel	mouseMove	
enumSource	mouseRightDouble	
enumSourceToFile	mouseRightDown	
enumTableLinks	mouseRightUp	
enumUIObjectNames	mouseUp	
enumUIObjectProperties	moveToPage	
formCaller	open	
formReturn	openAsDialog	
getFileName	postAction	
getPosition	run	
getProtoProperty	save	
getSelectedObjects	saveStyleSheet	

action method/procedure

Form

Performs an action command.

Syntax

```
action ( const actionId SmallInt ) Logical
```

Description

action performs the function represented by the constant *actionId*, where *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

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

attach method

constants. You can use user-defined action constants to signal other parts of an application. For example, 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. The Paradox default response for user-defined **action** constants is simply to pass the action to the action method.

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 ActionEvent.

Note

- When you call the **action** method as a procedure, the form dispatches it to the object represented by *self*. The event bubbles through the containership hierarchy until the event either reaches an object 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 the following example, the form sample:custform contains field objects bound to the sample:customer table. The current form contains a button named *openEditCust*; the **pushButton** method for *openEditCust* opens sample:custform, starts Edit mode, and waits for sample:custform to be closed:

```
; openEditCust::pushButton
method pushButton(var eventInfo Event)
var
    f Form
endVar
f.open("custform.fsl") ; open custform
f.action(DataBeginEdit) ; start Edit mode on custform
message("To return, close custform form.")
if f.wait() then          ; this form will be inactive until custform returns
f.close()                 ; this form must close custform
endif
endMethod
```

attach method

Form

Associates a Form variable with an open form.

Syntax

```
attach ( [ const formTitle String ] ) Logical
```

Description

attach associates a Form variable with an open form. You can use *formTitle* to specify a form's title, or you can omit *formTitle* to attach to the form where **attach** is executing. This method returns True if successful; otherwise, it returns False.

Note

- The argument *formTitle* specifies a form's title as displayed in the Title Bar (for example, Orders), not the form's filename or UIObject name. You can specify a form's title interactively by right-clicking the form's Title Bar, choosing Window Style, and entering a value in the Window Style dialog box. You can specify a title in ObjectPAL by setting a form's Title property, or by calling **setTitle**.

Example

In the following example, a form has two buttons: *openSites* and *attachToSites*. The **pushButton** method for *openSites* opens 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; therefore *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")
sitesForm.Title = "Notes" ; Set the form's title.

endMethod
```

The following code is attached to the **pushButton** method for *attachToSites*:

```
; attachToSites::pushButton
method pushButton(var eventInfo Event)
var
    sitesForm Form
endVar

; Attach to Sitenote by its title (Notes).
; Note that this won't work: sitesForm.attach("Sitenote")
if not sitesForm.attach("Notes") then
    errorShow()
    return
endif

; 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
```

bringToTop method/procedure**Form**

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 and give the appearance of layers. Use **bringToTop** to display a window on the top of the stack and 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 and then opens a Table window for the *Orders* table. The Table window, *orderTV*, opens over the

close method/procedure

Sitenote form, *siteForm*. The method pauses for a few seconds and 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
```

close method/procedure

Form

Closes a window.

Syntax

1. (Method) close ()
2. (Procedure) close ([const *returnValue* AnyType])

Description

close closes a window as if the user has chosen Close from the Control menu.

Example

The following example uses **close** to return a value to a form that called it with *wait*. Assume a form contains a button called *btn1*. A second form contains two buttons called *btnReturnOK* and *btnReturnCancel*. The first form opens the second form and waits for one of three values: OK, Cancel, or False. OK and Cancel are returned from the two buttons on the second form (see the following code) and False is returned if the user closes the second form without pressing a button. The first form processes the user's selection in a **switch** statement that calls one of three custom methods (assumed to be defined elsewhere).

The following code is attached to the button *btn1* in the first (calling) form.

```
;frm1.btn1 :: pushButton
method pushButton(var eventInfo Event)
var
    f  Form      ;Declare form variable.
    s  String    ;Declare string value.
endVar

f.open("wait2") ;Open form that will return string.
s = string(f.wait()) ;Wait for value from other form.
s.view("Returned value") ;View returned value.

;Process returned value using custom methods defined elsewhere.
switch
    case s = "OK"      : cmOK() ;User pressed the OK button.
    case s = "Cancel": cmCancel();User pressed the Cancel button.
    case s = "False" : cmNone() ;User closed form, no button pressed.
endSwitch
endmethod
```

The following code is attached to the button *btnReturnOK* in the second (called) form:

create method

```
;frm2.btnReturnOK :: pushButton
method pushButton(var eventInfo Event)
    close("OK") ;Close & return OK.
endmethod
```

The following code is attached to the button *btnReturnCancel* in the second (called) form:

```
;frm2.btnReturnCancel :: pushButton
method pushButton(var eventInfo Event)
    close("Cancel") ;Close & return Cancel.
endmethod
```

create method

Form

Creates a blank form in a Form Design window.

Syntax

```
create ( ) Logical
```

Description

create opens a blank form and leaves it in a 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. You can attach methods to the form using the Form type method **methodSet**. Use the Form type method **run** to open the form in a Form window.

Example

In the following 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. After 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 Form 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 (View Data window)
; call the mouseUp method for the form
newForm.mouseUp(100, 100, LeftButton ) ; dialog box displays "Hello"
newForm.close() ; close the form
endMethod
```

delayScreenUpdates

Form

Turns delayed screen updates on or off.

Syntax

```
delayScreenUpdates ( const yesNo Logical )
```

delayScreenUpdates

Description

delayScreenUpdates postpones or enables the redrawing of areas of the screen. You must specify Yes or No in *yesNo*. Specifying Yes delays screen updates (redraws) until the system yields or is idle. This can increase performance in operations that frequently refresh the display (e.g., when using ObjectPAL to add items to a list). Specifying No allows screen updates to occur without delay.

For some operations, you won't notice a difference when **delayScreenUpdates** is set to Yes.

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. When this code runs, 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 they all appear 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 and 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
```


deliver method**Form**

Delivers a form.

Syntax

```
deliver ( ) Logical
```

Description

deliver behaves like Format, Deliver. This method saves a copy of a form with an .FDL extension, which prevents users from editing the form in the 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.

Paradox opens saved forms before delivered forms with the same name. For example, suppose the working directory contains ORDERS.FSL (a saved form) and ORDERS.FDL (a delivered form).

The following statement opens the saved form, ORDERS.FSL.

```
ordersForm.open("ORDERS") ; Opens :WORK:ORDERS.FSL.
```

To specify a delivered form, include the .FDL extension. For example,

```
ordersForm.open("ORDERS.FDL") ; Opens the delivered form.
```

Example

In the following example, the **createDeliver** button creates a new form, saves it to the name *Newhello* and 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 because a delivered form can't be loaded in a Form Design window.

```
; createDeliver::pushButton
method pushButton(var eventInfo Event)
var
    newForm Form
endVar
newForm.create()           ; create a new blank form (a Form 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
    errorShow("Can't load a delivered form.")
endif
endMethod
```

design method**Form**

Switches a form from the Form window to the Form Design window.

Syntax

```
design ( ) Logical
```

Description

design switches a form from the Form window to the Form Design window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

Use run to switch from the Form Design window to the Form window.

disableBreakMessage procedure

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 call to **sleep**. For more information, see the **sleep** method in the System type.

Example

The following example uses a custom procedure to force a form (specified by its title) into design mode.

```
proc forceDesign(const foTemp Form) Logical
if foTemp.isDesign() then
    return True
else
    return foTemp.design()
endif
endProc
```

disableBreakMessage procedure

Form

Prevents program interruption by CTRL + Break.

Syntax

```
disableBreakMessage ( const yesNo Logical ) Logical
```

Description

disableBreakMessage lets you prevent or allow the user to interrupt a running program with CTRL + Break.

Example

In the following 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
```

disablePreviousError procedure

Form

Specifies whether you have access to the Previous Error dialog box.

Syntax

```
disablePreviousError ( const yesNo Logical ) Logical
```

Description

By default, when you move the pointer over the Status Bar, the pointer changes shape; you can then click the Status Bar to display the Previous Error dialog box (if error information is available). If *yesNo* is Yes (or True), **disablePreviousError** prevents this behavior; otherwise it restores the default behavior.

Returns True if successful; otherwise, returns False. This setting remains in effect (and affects all forms) as long as Paradox is running. The default behavior is restored the next time you start Paradox.

Example

The following example uses `disablePreviousError` in a script named `InitApp` to prevent user access to the Previous Error dialog box:

```
; InitApp::run
method run(var eventInfo Event)
    disablePreviousError(Yes)
    openMainForm() ; Call a custom method to open the main application form.
endMethod
```

dmAddTable method/procedure**Form**

Adds a table to a form's data model.

Syntax

```
dmAddTable ( const tableName String ) Logical
```

Description

dmAddTable adds the table *tableName* to a form's data model, where *tableName* is a valid table name. This method returns `True` if successful; otherwise, it returns `False`.

Example

In the following 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* and 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*.

The following code is attached to 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 and 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 and 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
```

dmAttach method/procedure**Form**

Associates a TCursor variable with a table in the form's data model.

Syntax

```
dmAttach ( tc TCursor, const tableName String ) Logical
```

Description

dmAttach associates the TCursor variable *tc* with the table *tableName* in the form's data model, where *tableName* is either a valid table name or a table alias. This method returns True if successful; otherwise it returns False.

Example

The following example demonstrates how to use **dmAttach** and **dmResync** to keep two forms synchronized. Both forms have the *Customer* table in their data models. When the user moves from the first form *frm1* to the second form *frm2*, a form variable *f* is used to attach back to the first form and **dmAttach** is used to attach to the appropriate table in its data model. **dmResync** is used to move to the same record as the first form.

```
;Frm2.pgcl :: setFocus
method setFocus(var eventInfo Event)
var
    f    Form           ;Declare a form variable.
    tc   TCursor        ;Declare a TCursor variable.
endVar

if f.attach("dmAttach2") then ;Attach to other form.
    f.dmAttach(tc, "Customer.db") ;Attach tc to a table in the
        dmResync("Customer.db", tc) ;data model of the other form.
        ;Then sync the two forms.
endif
endMethod
```

dmBuildQueryString method/procedure**Form**

Builds a query string based on the data model of a form.

Syntax

```
dmBuildQueryString ( var queryString String ) Logical
```

Description

dmBuildQueryString creates a query string *queryString* based on the data model of a form. The query built by **dmBuildQueryString** creates checked example elements for all the link fields in the data model. The form's data model must have a linked table. **dmBuildQueryString** returns True if it is successful; otherwise, it returns False.

Example 1

The following example assumes a data model has the *Customer* and *Orders* tables linked on the CustomerNo field. The code displays a query string based on that data model.

```
method pushButton(var eventInfo Event)
var
    stQBE String
endVar
dmBuildQueryString(stQBE)
stQBE.view("Query String")

{
Displays the following string:
Query
```

```

::C:\COREL\PARADOX\BLDNOTES\CUSTOMER.DB|CustomerNo    |
                                     |Check __join1!!|

::C:\COREL\PARADOX\BLDNOTES\ORDERS.DB|CustomerNo    |
                                     |Check __join1|

EndQuery
}
endMethod

```

Example 2

The following example assumes a form contains a button named *btnDMQuery*. The **pushButton** method for *btnDMQuery* uses **dmBuildQueryString** as a procedure to generate a query string in **s**. **readFromString** is called to assign the string to a Query variable and the method runs the query and opens a Table window for the *Answer* table.

```

;btnDMQuery :: pushButton
method pushButton(var eventInfo Event)
  var
    s      String
    tv     TableView
    qVar    Query
  endVar

  dmBuildQueryString(s)
  qVar.readFromString(s)
  if qVar.executeQBE() then
    tv.open(":PRIV:ANSWER.DB")
  else
    errorShow()
    return
  endIf
endMethod

```

dmEnumLinkFields method/procedure

Form

Lists the fields that link two tables.

Syntax

```
dmEnumLinkFields ( var masterTable String, var masterFields Array[ ] String, const
detailTable String, var detailFields Array[ ] String, var detailIndex String ) Logical
```

Description

dmEnumLinkFields lists the fields that link the tables named in *masterTable* and *detailTable*. You must supply a table name or table alias for *detailTable*. This method assigns values to the other variables (passed as arguments) as follows:

Variable	Assigned value
<i>masterTable</i>	The name of the master table. Blank if the table specified in <i>detailTable</i> has no master table.
<i>masterFields</i>	Names of the linking fields in the master table. Blank if the table specified in <i>detailTable</i> has no master table.

dmGet method/procedure

detailFields	Names of the linking fields in the detail table. Blank if the table specified in detailTable has no master table. If the detail table is a dBASE table and uses an expression index, the expression is returned in angled brackets. Examples: + LASTNAME MEANS AN EXPRESSION INDEX BASED ON THE FIELDS NAMED FIRSTNAME AND LASTNAME; + LASTNAME;QTY I MEANS AN EXPRESSION INDEX BASED ON THE FIELDS NAMED FIRSTNAME AND LASTNAME WITH QTY I AS A SUBSET CONDITION.
indexName	Name of the index used by the detail table. Blank if the table specified in detailTable is not using an index. If the detail table is a dBASE table, you can use dmGetProperty to get the associated tag name, if any.

The tables must already be in the specified data model. This method returns True if successful; otherwise, it returns False.

Example

In the following example, assume that a form's data model links the *Customer* and *Orders* tables on the CustomerNo field, with the *Orders* table as the detail table. The tables do not use secondary indexes.

```
method pushButton(Var eventInfo Event)
  var
    mAr, dAr Array[] String
    m, d, inx String
  endVar

  d = "orders"
  dmEnumLinkFields(m, mAr, d, dAr, inx)
  m.view("Master table name") ; Displays CUSTOMER.DB
  mAr.view("Master link fields") ; Displays Customer No
  d.view("Detail table name") ; Displays ORDERS.DB
  dAr.view("Detail link fields") ; Displays Customer No
  inx.view("Index name") ; Displays Customer No
endMethod
```

dmGet method/procedure

Form

Retrieves a field value from a table in the data model.

Syntax

```
dmGet ( const tableName String, const fieldName String, var datum AnyType ) Logical
```

Description

dmGet provides access to table data in the form's data model. **dmGet** writes to *datum*, a field value from a specified table. The table specified by *tableName* must be the name or table alias of a table 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 active 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.

dmGetProperty method/procedure

```
; getHighlight::pushButton
method pushButton(var eventInfo Event)
var
    siteHighlight AnyType
    qAnswer       String
endVar
; get the value in the Site Highlight field for the active 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
```

For information on table aliases, see Table Aliases in the Paradox online help.

dmGetProperty method/procedure

Form

Returns the value of a specified table property.

Syntax

- 1. dmGetProperty (const *tableName* String, const *propertyName* String) AnyType
- 2. dmGetProperty (const *tableName* String, const *propertyName* String, var *value* AnyType) Logical

Description

Returns the value of a property *propertyName* of the table *tableName* in the specified data model. The value of *tableName* must be a valid table name or a table alias.
The return value depends on the value of *propertyName* that you supply from the following:

This value	Returns
AutoAppend	True if Auto Append is set to True for the table; otherwise, it returns False.
Editing	True when a form is in Edit mode, or a field object is active and being edited; otherwise, it returns False.
Flyaway	True when a record has moved to its sorted position in a table; otherwise, it returns False.
FullName	The full filename (as a string, including path or alias) of the table.

dmGetProperty method/procedure

Index	The name of the index (as a string) that is currently used to view the table. For a child table, it returns the name of the index chosen in the link diagram. For a master table or unlinked table, it returns the setting of ORDER/RANGE. It returns an empty string when the primary key is used.
Inserting	True when a record is being inserted anywhere in a form; otherwise, it returns False.
LinkType	A string describing the way the table relates to its master table: None, One-to-one, or One-to-many.
Locked	True when the table bound to a design object is locked; otherwise, it returns False.
Name	The table's alias (as a string) if it exists; otherwise, returns an empty string.
Next	The name (as a string) of the next object in the same container.
One-to-many	The name (as a string) of the first detail table linked 1:M to this table.
One-to-one	The name (as a string) of the first detail table linked 1:1 to this table.
Parent	The table name (as a string) of this table's master in the data model.
Read-only	True if READONLY is set to True for the table; otherwise, it returns False.
Refresh	True when data displayed onscreen is being changed, either across a network (by an ObjectPAL statement) or by a user action; otherwise, it returns False.
StrictTranslation	True if STRICT TRANSLATION is set to True for the table; otherwise, it returns False.
TagName	The tag name (as a string) for the current dBASE index (if any); otherwise, it returns an empty string.
Touched	True when the user has made changes to data not yet committed.

Syntax 1 returns the property value directly.

Syntax 2 assigns the value to *value*, an AnyType variable that you declare and pass as an argument.

Syntax 2 returns True if the method succeeds; otherwise, it returns False.

For both syntaxes, **dmGetProperty** returns False if *tableName* is not in the data model, or if the value of *propertyName* is not one of the strings listed earlier.

The value of *tableName* must be a valid table name or a table alias .

If *propertyValue* = Name this method returns the table's alias (as a string) if it exists; otherwise, it returns an empty string.

If *propertyValue* = FullName this method returns the full filename (including path or alias) of the table.

Example

The following example sets a table's Auto Append property to False if the table isn't read-only and then checks to see if the table has a one-to-many link to another table. If it does, the read-only setting of the master table is set to the same read-only setting as the detail (subject) table.

```
method UpdateProperties()
```

```
  if dmGetProperty(subject.tableName, "ReadOnly") True then
    dmSetProperty(subject.tableName, "AutoAppend", False)
  endif
```



```

if dmGetProperty(subject.tableName, "LinkType") = "One-to-many" then
  dmSetProperty(dmGetProperty(subject.tableName, "Parent"), "ReadOnly",
    dmGetProperty(subject.tableName, "ReadOnly"))
endif

endMethod

```

For information on table aliases, see Table Aliases in the Paradox online help.

dmHasTable method/procedure

Form

Reports whether a table is part of the data model of a form.

Syntax

```
dmHasTable ( const tableName String ) Logical
```

Description

dmHasTable reports whether *tableName* is a table associated with a form, where *tableName* is a valid table name or table alias.

Example

See the example for **dmAddTable** for an illustration of how to use **dmHasTable** as a procedure.

The following 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 and 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

```

For information on table aliases, see Table Aliases in the Paradox online help.

dmLinkToFields method/procedure

Form

Links two tables in a data model based on lists of field names.

Syntax

```
dmLinkToFields ( const masterTable String, const masterFields Array[ ] String, const
detailTable String, const detailFields Array[ ] String ) Logical
```

Description

dmLinkToFields links the tables specified in *masterTable* and *detailTable* on the field names listed in *masterFields* and *detailFields* (resizeable arrays of strings). The values of *masterTable* and *detailTable* can be table names or table aliases. The tables must already be in the form's data model.

dmLinkToFields method/procedure

The linking fields cannot be any of the following types: Binary, Byte, Formatted Memo, Graphic, Logical, Memo, or Object Linking and Embedding (OLE). This method returns True if successful; otherwise, it returns False. If the detail table does not have an index that matches the fields in *detailFields*, it returns False.

Example 1

The following example creates a form, adds the Customer and Orders tables to the new specified data model, and calls **dmLinkToFields** to link the tables. It also creates some field objects and a table frame and binds them to the tables. Finally, this code runs the new form so you can see the results.

The following code specifies the names of the fields to link; you could leave this to Paradox, but default linking in Paradox may not give the results you expect.

```
method pushButton(var eventInfo Event)
    var
        masterTC, detailTC      TCursor
        newForm                  Form
        masterFieldsAr,
        detailFieldsAr,
        keyFieldsAr              Array[] String
        badKeyTypesAr           Array[7] String
        masterName,
        detailName,
        keyFieldName,
        newFormName              String
        newField,
        newTFrame                UIObject
        x, y, w, h, offset       LongInt
        i                        SmallInt
    endVar

    ; initialize variables
    masterName = "customer.db"
    detailName = "orders.db"
    newFormName = "custOrd.fsl"

    badKeyTypesAr[1] = "MEMO"          ; types not allowed as key fields
    badKeyTypesAr[2] = "FMTMEMO"
    badKeyTypesAr[3] = "BINARYBLOB"
    badKeyTypesAr[4] = "GRAPHIC"
    badKeyTypesAr[5] = "OLEOBJ"
    badKeyTypesAr[6] = "LOGICAL"
    badKeyTypesAr[7] = "BYTES"

    masterTC.open(masterName)
    masterTC.enumFieldNames(masterFieldsAr)

    detailTC.open(detailName)
    detailTC.enumFieldNames(detailFieldsAr)

    ; specify the key field(s)
    keyFieldName = "Customer No"

    ; make sure key field type is valid
    if badKeyTypesAr.contains(masterTC.fieldType(keyFieldName)) or
       badKeyTypesAr.contains(detailTC.fieldType(keyFieldName)) then
        msgStop("Invalid key field type:",
                keyFieldName + " in\n" +
                masterName + " or\n" + detailName)
    return
```

```

else
    keyFieldsAr.grow(1)
    keyFieldsAr[1] = keyFieldName
endIf

; create the form
newForm.create()
newForm.dmAddTable(masterName)
newForm.dmAddTable(detailName)

if newForm.dmLinkToFields(masterName, keyFieldsAr,
    detailName, keyFieldsAr) then

; place objects in the form

    x = 100
    y = 100
    w = 2880
    h = 360
    offset = 10

; create field objects bound to master table
    for i from 1 to masterFieldsAr.size()
        newField.create(FieldTool, x, y, w, h, newForm)
        y = y + h + offset
        newField.TableName = masterName
        newField.FieldName = masterFieldsAr[i]
        newField.Visible = Yes
    endFor

; create a table frame bound to detail table
    newTFrame.create(TableFrameTool, x, y, w, 8 * h, newForm)
    newTFrame.TableName = detailName
    newTFrame.Visible = Yes

; save the form and run it
    newForm.save(newFormName)
    newForm.run()

else
    errorShow("Link failed")
endIf

endMethod

```

Example 2

The following example shows how to use **dmLinkToFields** to link three tables 1:M:M. Like the Example 1, this code specifies which fields to link.

```

method pushButton(var eventInfo Event)
    var
        firstTable,
        secondTable,
        thirdTable      String
        firstKeyAr,
        secondKeyAr,
        thirdKeyAr      Array[] String
        newForm          Form
    endVar

```

dmLinkToIndex method/procedure

```
; initialize variables
firstTable = "customer.db"
secondTable = "orders.db"
thirdTable = "lineitem.db"

firstKeyAr.grow(1)
firstKeyAr[1] = "Customer No"
secondKeyAr.grow(1)
secondKeyAr[1] = "Customer No"
; thirdKeyAr is initialized below, after 1st link

; create the form
newForm.create()

newForm.dmAddTable(firstTable)
newForm.dmAddTable(secondTable)
newForm.dmAddTable(thirdTable)

; 1st link
if newForm.dmLinkToFields(firstTable, firstKeyAr,
                        secondTable, secondKeyAr) then

    ; initialize arrays for 2nd link
    secondKeyAr[1] = "Order No"

    thirdKeyAr.grow(1)
    thirdKeyAr[1] = "Order No"

    ; 2nd link
    if newForm.dmLinkToFields(secondTable, secondKeyAr,
                            thirdTable, thirdKeyAr) then

        {Code to create UIObjects in new form could go here.}

        newForm.save("ordentry.fsl")

    else
        errorShow("2:3 link failed.")
    endIf

else
    errorShow("1:2 link failed.")
endIf

endMethod
```

dmLinkToIndex method/procedure

Form

Links two tables in the form's data model based on a list of field names and an index name.

Syntax

```
dmLinkToIndex ( const masterTable String, const masterFields Array[ ] String, const
detailTable String, const detailIndex String ) Logical
```

Description

Links the tables specified in *masterTable* and *detailTable* to the field names listed in *masterFields* and the index specified in *detailIndex*. You can specify a Paradox table's primary index by assigning an empty string to *detailIndex*.

dmLinkToIndex method/procedure

The values of *masterTable* and *detailTable* can be table names or table aliases. The tables must already be in the form's data model. This method returns True if successful; otherwise, it returns False.

The linking fields cannot be any of the following types: Binary, Bytes, Formatted Memo, Graphic, Logical, Memo, or Object Linking and Embedding (OLE).

Example

The following example creates a form, adds the Customer and Orders tables to the new specified data model, and calls **dmLinkToIndex** to link the tables. It also creates some field objects and a table frame and binds them to the tables. Finally, this code runs the new form so you can see the results.

```
method pushButton(var eventInfo Event)
var
    masterTC, detailTC      TCursor
    newForm                 Form
    masterFieldsAr,
    detailFieldsAr,
    masterKeysAr,
    detailKeysAr            Array[] String
    masterName,
    detailName,
    detailIndexName,
    newFormName             String
    newField,
    newTFrame              UIObject
    x, y, w, h, offset      LongInt
    i                      SmallInt
endVar

; Initialize variables
detailIndexName = "Customer No"
newFormName = "idxDemo"
masterName = "customer.db"
detailName = "orders.db"

masterTC.open(masterName)
masterTC.enumFieldNames(masterFieldsAr)
masterTC.enumFieldNamesInIndex(masterKeysAr)

detailTC.open(detailName)
detailTC.enumFieldNames(detailFieldsAr)

; create the form
newForm.create()
newForm.dmAddTable(masterName)
newForm.dmAddTable(detailName)

if newForm.dmLinkToIndex(masterName, masterKeysAr,
                        detailName, detailIndexName) then

    x = 100
    y = 100
    w = 2880
    h = 360
    offset = 10

    for i from 1 to masterFieldsAr.size()
        newField.create(FieldTool, x, y, w, h, newForm)
        y = y + h + offset
        newField.TableName = masterName
```

dmPut method/procedure

```
        newField.FieldName = masterFieldsAr[i]
        newField.Visible = Yes
    endFor

    newTFrame.create(TableFrameTool, x, y, w, 8 * h, newForm)
    newTFrame.TableName = detailName
    newTFrame.Visible = Yes

    newForm.save(newFormName)
    newForm.run()

else

    errorShow("Link failed")

endIf

endMethod
```

For information on table aliases, see Table Aliases in the Paradox online help.

dmPut method/procedure

Form

Writes data to a table in the data model.

Syntax

```
dmPut ( const tableName String, const fieldName String, const datum AnyType ) Logical
```

Description

dmPut provides access to table data in the data model. **dmPut** writes *datum* to a field in a specified table. The value of *tableName* can be a table name or a *table alias*. The table specified by *tableName* must be one of the tables in the data model. *fieldName* must be a field in *tableName*. This method returns True if successful; otherwise, it returns False.

Example

See **dmGet** example.

dmRemoveTable method/procedure

Form

Removes a table from the form's data model.

Syntax

```
dmRemoveTable ( const tableName String ) Logical
```

Description

dmRemoveTable removes *tableName* from a form's data model. The value of *tableName* can be a table name or a table alias. Any objects on the form that depend on the table will be undefined when the table is removed. If any UIObjects in the form are bound to the table, **dmRemoveTable** fails. It returns True if successful; otherwise, it returns False.

Example

See **dmAddTable** example.

dmResync method/procedure

Form

Resynchronizes a table in the form's data model to a TCursor.

Syntax

```
dmResync ( const tableName String, var tc TCursor ) Logical
```

Description

dmResync synchronizes a specified table in a data model with the TCursor *tc*. The value of *tableName* can be a table name or a table alias.

When you resynchronize a table to a TCursor, the table's filter, index, and active record position will be changed to those of the TCursor. (For dBASE tables, the table will also take the Show Deleted setting of the TCursor.) This method works on forms in design mode or run mode.

Note

- **dmResync** only works when the TCursor is associated with the table in the data model. However, the table does not have to be displayed in the form.

Example

The following example shows how to use **dmResync** with the DataSource property to add items to a drop-down edit list. First, it shows how to use DataSource alone, which fills a list with values from a specified field (column) of a table. Then it shows how to use a TCursor and **dmResync** to fill a list with a specified subset of those values.

A field displayed as a drop-down edit list is a compound object: the field object (which displays the field value) contains a list object (which contains the items in the list). In a form, the list object is represented by the down-arrow (the arrow you click to display the list).

The usual place to attach list-building code is the list object's built-in **open** method, but you can attach the code to other methods or even to other objects (as shown in the second part of this example).

Assume a form contains a field object displayed as a drop-down edit list. The field object is bound to the ShipVia field of the *Orders* table. The following code is attached to the built-in **open** method of the list object (not the field object) named *shipViaList*. It fills the list with all the values in the ShippingCo field of the *Shippers* table in the working directory.

```
; shipViaList::open
; Full containership path: form.page.ShipVia.shipViaList
method open (var eventInfo Event)
  doDefault
  ; Fills list with all values in ShippingCo field of Shippers table.
  self.DataSource = "[Shippers.ShippingCo]"
endMethod
```

The following code uses **dmResync** to filter the list based on the value of another field. The premise here is that certain shipping methods are less expensive (and so more desirable) in certain parts of the country. When the user changes the value of the *State* field, this code updates the items in the list of shippers.

```
; State::changeValue
method changeValue (var eventInfo ValueEvent)
  var
    tcShippers    TCursor
    stStateCode,
    stFldName,
    stDmTbName    String
    dyCriteria    DynArray[] AnyType
  endVar

  doDefault ; Execute the built-in code to commit the field value.
  if eventInfo.errorCode() = 0 then
    return ; If there's an error, exit the method.
```

dm SetProperty method/procedure

```
endIf

stStateCode = self.Value ; Get the value of the State field.
stFldName   = "State"    ; Filter on the State field.
stDmTbName  = "Shippers"

dyCriteria[stFldName] = stStateCode

; Associate a TCursor with a table in the form's data model.
dmAttach(tcShippers, stDmTbName)

tcShippers.setGenFilter(dyCriteria) ; Set a filter on the TCursor.
; You could also set an index, etc.

; Synchronize the table in the data model with the TCursor.
; The table takes the filter from the TCursor.
dmResync(stDmTbName, tcShippers)

; Now the list displays only the shippers for the specified state.
ShipVia.shipViaList.DataSource = "[Shippers.ShippingCo]"

endMethod
```

For information on table aliases, see Table Aliases in the Paradox online help.

dm SetProperty method/procedure

Form

Sets the value of a specified table property.

Syntax

```
dm SetProperty ( const tableName String, const propertyName String, value AnyType)
Logical
```

Description

dm SetProperty lets you change the value of a property (specified in *propertyName*) associated with the table specified in *tableName* and found in the data model.

The value of *tableName* can be a table name or a table alias. The value of *propertyName* is one of the following properties:

AutoAppend	Set <i>propertyValue</i> to True to set AUTO APPEND ON for the table; otherwise, set it to False.
Name	The value of <i>propertyValue</i> specifies the table's alias as a string. The operation fails if the table alias is already in use.
ReadOnly	Set <i>propertyValue</i> to True if READONLY should be True for the table; otherwise, set it to False.
StrictTranslation	Set <i>propertyValue</i> to True if STRICT TRANSLATION should be True for the table.; otherwise, set it to False.
Touched	Set <i>propertyValue</i> to True when the user has made changes not yet committed.

Example

See **dmGetProperty** example.

dmUnlink method/procedure**Form**

Unlinks two tables in the form's data model.

Syntax

```
dmUnlink ( const masterTable String, const detailTable String ) Logical
```

Description

dmUnlink breaks the link between the tables specified in *masterTable* and *detailTable*. *masterTable* must refer to the master table in the link, and *detailTable* must refer to the detail table in the link. The values of *masterTable* and *detailTable* can be table names or table aliases.

This method fails if the tables are not in the data model; it also fails if they are in the data model but not linked.

This method returns True if successful; otherwise, it returns False.

Example

The following example uses dmUnlink to break the link between two tables:

```
method pushButton(var eventInfo Event)

var
    theForm          Form
    masterTable,
    oldDetailTable,
    newDetailTable,
    oldFormName,
    newFormName      String
    newKeysAr        Array[] String
endVar

; initialize variables
oldFormName = "custOrd"
newFormName = "newOrd"

masterTable   = "CUSTOMER"
oldDetailTable = "ORDERS"
newDetailTable = "NEW_ord"

newKeysAr.grow(1)
newKeysAr[1] = "Customer No"

; load the form and change the data model
theForm.load(oldFormName)

if theForm.dmHasTable(masterTable) and
    theForm.dmHasTable(oldDetailTable) then

    theForm.dmAddTable(newDetailTable)
    theForm.dmUnlink(masterTable, oldDetailTable)

    theForm.dmLinkToFields(masterTable, newKeysAr,
                           newDetailTable, newKeysAr)

    theForm.ORDERS.TableName = newDetailTable

    theForm.dmRemoveTable(oldDetailTable)
    theForm.save(newFormName)

else
```

enumDataModel method/procedure

```
        errorShow()  
    endIf  
  
endMethod
```

For information on table aliases, see Table Aliases in the Paradox online help.

enumDataModel method/procedure

Form

Lists the tables in the form's data model.

Syntax

```
enumDataModel ( const tableName String ) Logical
```

Description

enumDataModel creates a table that lists information about the tables in the form's data model. 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	Description
TableName	A128	Table alias, if it exists, or filename of the table (without file extension)
PropertyName	A64	A property name
PropertyValue	A255	Value of the corresponding property

Example

In the following example, a form contains a button named *enumerateDataModel*. The **pushButton** method for *enumerateDataModel* uses **enumDataModel** as a procedure to enumerate the properties of all the tables in the data model for the current form to a table called DMORDERS.DB. The method then opens a Table window for the DMOrders table.

```
;enumerateDataModel::pushButton  
method pushButton(var eventInfo Event)  
    var  
        tv    TableView  
    endVar  
  
    enumDataModel("dmOrders.db")  
    tv.open("dmOrders.db")  
endMethod
```

Property Names for enumDataModel

Property	Description
AutoAppend	Returns True if AUTO APPEND is set to True for the table; otherwise, it returns False
FullName	Returns the full filename (including path or alias) of the table

Index	Returns the name of the index (as a string) that is currently used to view the table. For a child table, it returns the name of the index chosen in the link diagram. For a master table or unlinked table, it returns the setting of ORDER/RANGE. It returns an empty string when the primary key is used.
LinkFields	Returns a comma-separated list of fields that define the link. If the detail table is a dBASE table and uses an expression index, the expression is returned in angled brackets. Examples: + LASTNAME MEANS AN EXPRESSION INDEX BASED ON THE FIELDS NAMED FIRSTNAME AND LASTNAME; + LASTNAME;QTY I MEANS AN EXPRESSION INDEX BASED ON THE FIELDS NAMED FIRSTNAME AND LASTNAME WITH QTY I AS A SUBSET CONDITION.
LinkType	Returns a string describing the way the table relates to its master: None, One-to-one, or One-to-many
Name	Returns the table's alias (as a string) if it exists; otherwise, returns an empty string
Next	Returns the name (as a string) of the next object in the same container
One-to-many	Returns the name (as a string) of the first detail table linked 1:M to this table
One-to-one	Returns the name (as a string) of the first detail table linked 1:1 to this table
Parent	Returns the table name (as a string) of this table's master in the data model. For example, in a CUSTOMER—>BOOKORD form, dmGetProperty("BOOKORD","PARENT") = "CUSTOMER.DB." If the table has no master, an empty string is returned.
Read-only	Returns True if READONLY is set to True for the table; otherwise, it returns False
StrictTranslation	Returns True if STRICT TRANSLATION is set to True for the table; otherwise, it returns False
TagName	Returns the tag name (as a string) for the current dBASE index (if any); otherwise, it returns an empty string

enumSource method

Form

Creates a table that lists the methods for each object in a form.

Syntax

```
enumSource ( const tableName String [ , const recurse Logical ] ) Logical
```

Description

enumSource creates a Paradox table that lists every object for which you have written a method, and 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.

The structure of the created table is as follows:

Field name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

enumSourceToFile

The `Object` field contains the full path name of the object.

If `recurse` is `False`, this method returns only the method definitions for the form. To include the source code of methods for all objects contained by the form, `recurse` must be `True`.

Note

- If the `recurse` parameter is not included, then it is assumed to be `True` and the source code of methods for all objects contained by the form will be returned.

Example

In the following 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 opens a Table window for the *Tempsorc* table and waits for the user to close it. The method then 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

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
```

enumSourceToFile

Form

Creates a file that lists the methods for each object in a form.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] ) Logical
```

Description

enumSourceToFile creates a text file that lists every object for which you've written a method, and 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.

If `recurse` is `False`, this method returns only the method definitions for the form. To include the source code of methods for all objects contained by the form, `recurse` must be `True`.

Note

- If the `recurse` parameter is not included, then it is assumed to be `True` and the source code of methods for all objects contained by the form will be returned.

Example

In the following example, 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

enumTableLinks method/procedure

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
```

enumTableLinks method/procedure

Form

Creates a table that lists the tables linked in a form.

Syntax

```
enumTableLinks ( const tableName String ) Logical
```

Description

enumTableLinks creates a Paradox table that lists the names of tables linked in 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.

This method creates a table that contains one record for each table in the data model. The structure of the table is:

Field name	Type	Description
Table	A255*	Table name, without alias, path, or extension (for example, ORDERS).
Parent	A255*	Name of parent table, or blank if table has no parent.
LinkType	A24*	Type of link between table and master table: None, One-to-many, or One-to-one.

Example

In the following 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")
```

enumUIObjectNames method

```
tempTable.wait()
siteForm.open("Sitenote.fsl")
siteForm.enumTableLinks("Sitenote.db") ; lists links to siteForm
siteForm.close()
tempTable.open("Sitenote.db")
tempTable.wait()
tempTable.close()
endMethod
```

enumUIObjectNames method

Form

Creates a table that lists the UIObjects contained in a form.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

Description

enumUIObjectNames creates a Paradox table that lists 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.

The structure of *tableName* is as follows:

Field Name	Type	Size
ObjectName	A	128
ObjectClass	A	32

Note

- ObjectName includes the entire path name of the object.

Example

In the following 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 the table.

```
; 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
                                                ; SITEOBSJ.S.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
```

enumUIObjectProperties method**Form**

Lists the properties of each UIObject contained in a form.

Syntax

```
enumUIObjectProperties ( const tableName String ) Logical
```

Description

enumUIObjectProperties creates a Paradox table that lists 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 the following 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.")
    enumUIObjectProperties("tempprop.db")
tempTable.open("tempprop")
message("Close the table to continue.")
tempTable.wait()
tempTable.close()
endMethod
```

formCaller procedure**Form**

Creates a handle to the calling form.

formReturn procedure

Syntax

```
formCaller ( var caller Form ) Logical
```

Description

formCaller assigns the handle of the current form's calling form to *caller*, if the form is waiting. If the current form was not opened by another form, and the form that opened the current form is not waiting for the current form, the method returns False and *caller* is unassigned.

Example

In the following 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
```

The following code is 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
formReturn()
endMethod
```

formReturn procedure

Form

Returns control to a suspended method.

Syntax

```
formReturn ( [ const returnValue AnyType ] )
```

Description

When one form opens another form and calls wait, the first form suspends ObjectPAL execution (in effect, yielding to the second form) until the second form returns control by calling **formReturn**. You can choose to return a value to the first form in *returnValue*. You can also use **formReturn** to return control (and a value) from a script.

formReturn posts a message to the Windows message queue; therefore, ObjectPAL statements that follow **formReturn** will execute before the form returns control.

If no other form is waiting for the current form, **formReturn** closes the current form. If a form is waiting for the current form, **formReturn** does not close the current form.

Example

The following 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. Note that the calling form must call **close** to close the dialog box; the call to **formReturn** does not close the dialog box.

```
; 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
    whichButton = String(dlgForm.wait())
    dlgForm.close()
    ; return value is cast as a String so that it will be correct
    ; type even if user closes dialog box from the system menu
    msgInfo("Button pressed", whichButton)
else
    msgStop("Stop", "Couldn't open the form.")
endif
endMethod
```

The following method is attached to the **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
```

The following method is attached to *cancelButton* in *dlgForm*. It returns a value of Cancel when it returns control to the method that called **wait**. The **message** statement that follows the call to **formReturn** is not required; it is included here to show that statements following a call to **formReturn** execute before control is returned to the calling form.

```
; cancelButton::pushButton
method pushButton(var eventInfo Event)
formReturn("Cancel") ; return "Cancel" to calling form
message("Cancel") ; This statement will execute.
endMethod
```

getFileName method/procedure**Form**

Returns the path, filename, and extension of the associated form.

Syntax

```
getFileName( ) String
```

Description

As a method, **getFileName** returns the path, filename, and extension of the form associated with a Form variable. As a procedure, it returns the path, filename, and extension of the current form. If the procedure is being called from a calculated field attached to a table, the table must contain data for the

getPosition method/procedure

procedure to work. Compare this method to **getTitle**, which returns the text in a Form window's Title Bar.

Example

The following example displays the filename of the current form in the Status Bar.

```
method pushButton(var eventInfo Event)
  message(getFileName())
endMethod
```

getPosition method/procedure

Form

Reports the position of a window onscreen.

Syntax

```
getPosition ( var x LongInt, var y LongInt, var w LongInt, var h LongInt )
```

Description

getPosition gets the position of a window relative to the Paradox desktop. 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).

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper-left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes, and for the Paradox desktop application, the position is given relative to the entire screen; for forms, reports, and Table windows, the position is given relative to the Paradox desktop.

Example

In the following 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
```

getProtoProperty method/procedure

Form

Reports the value of a specified property of a prototype object.

Syntax

```
getProtoProperty ( const objectType SmallInt, propertyName String ) AnyType
```

Description

getProtoProperty returns the value of the property specified in *propertyName* of the prototype object specified in *objectType*. To specify *objectType*, use one of the UIObjectTypes constants. If called as a

method, **getProtoProperty** operates on prototype objects in the style sheet of the specified form. If called as a procedure, **getProtoProperty** uses the style sheet of the current form.

Example

The following example uses **getProtoProperty** to store the current default color for the box tool. Next, it specifies a new box color and creates three new boxes, and then restores the default box color.

```
const
    kOneInch = 1440 ; One inch = 1,440 twips.
endConst
method mouseClicked(var eventInfo MouseEvent)
    var
        uiRedBox      UIObject
        thisForm       Form
        liDefaultBoxColor    LongInt
    endVar
    thisForm.attach() ; Get a handle to this form.

    ; Get current default color.
    liDefaultBoxColor = thisForm.getProtoProperty(BoxTool, "Color")

    ; Set box color and create 3 boxes using new prototype.
    thisForm.setProtoProperty(BoxTool, "Color", Red)
    uiRedBox.create(BoxTool, kOneInch, kOneInch, kOneInch, kOneInch)
    uiRedBox.Visible = Yes
    uiRedBox.create(BoxTool, 2 * kOneInch, kOneInch, kOneInch, kOneInch)
    uiRedBox.Visible = Yes
    uiRedBox.create(BoxTool, 3 * kOneInch, kOneInch, kOneInch, kOneInch)
    uiRedBox.Visible = Yes

    ; Restore the default box color.
    thisForm.setProtoProperty(BoxTool, "Color", liDefaultBoxColor)
endMethod
```

getSelectedObjects

Form

Creates an array that lists the selected objects in a form.

Syntax

```
getSelectedObjects ( var objects Array[ ] UIObject ) SmallInt
```

Description

getSelectedObjects creates an array *objects* that lists the selected objects of a form and returns the number of objects selected. This procedure is useful for creating routines that manipulate objects on forms in design mode.

Example

The following example creates a form that contains three boxes, selects two of the boxes, displays their names in a dialog box, and sets their color to blue:

```
;btnObjectsSelected :: pushButton
const
    kOneInch = 1440 ; One inch = 1,440 twips.
endConst

method pushButton(var eventInfo Event)
    var
        foTemp      Form
        arObjects    Array[] UIObject
```

getStyleSheet method/procedure

```
        arObjNames    Array[] String
        uiVar          UIObject
        si,
        siSelObj       SmallInt
        stBoxName      String

    endVar

    foTemp.create()

    ; Create 3 boxes.
    for si from 1 to 3
        uiVar.create(BoxTool, si * kOneInch, si*kOneInch,
                     kOneInch, kOneInch, foTemp)
        uiVar.Name = "Box" + String(si)
        uiVar.Visible = Yes
    endFor

    ; Select Box2 and Box3 by setting the Select property.
    for si from 2 to 3
        stBoxName = "Box" + String(si)
        uiVar.attach(foTemp.(stBoxName))
        uiVar.Select = Yes
    endFor

    ; Get the selected objects.
    siSelObj = foTemp.getSelectedObjects(arObjects)
    siSelObj.view("Number of selected objects:")

    ; Get the names of the selected objects.
    arObjNames.setSize(siSelObj)
    for si from 1 to siSelObj
        uiVar.attach(arObjects[si])
        arObjNames[si] = uiVar.Name
    endFor
    arObjNames.view("Names of selected objects:")

    ; Change the color of the selected objects.
    for si from 1 to arObjects.size()
        uiVar.attach(arObjects[si])
        uiVar.Color = Blue
    endFor

    foTemp.close()
endMethod
```

getStyleSheet method/procedure

Form

Returns the name of a form's style sheet.

Syntax

```
getStyleSheet ( ) String
```

Description

getStyleSheet returns the full path and filename of a form's style sheet (e.g., C:\COREL\PARADOX\COREL.FT).

If called as a method, **getStyleSheet** returns the filename of the style sheet of the specified form. If called as a procedure, it uses the style sheet of the current form.

getStyleSheet returns the name of the style sheet used by the specified form, which may be different from the Paradox system style sheet. To get the name of the default screen style sheet, call the **getDefaultScreenStyleSheet** procedure defined for the System type. To get the name of the default printer style sheet, call the **getDefaultPrinterStyleSheet** procedure defined for the System type.

Example

See **setStyleSheet** example.

getTitle method/procedure

Form

Returns the text in the window's Title Bar.

Syntax

```
getTitle ( ) String
```

Description

getTitle returns the text in the Title Bar of the window that contains 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
```

hide method/procedure

Form

Makes a window invisible.

Syntax

```
hide ( )
```

Description

hide makes a window invisible but doesn't close the window.

Example

In the following example, the **pushButton** method for *hideForm* opens a form, hides it and then shows it again:

```
; hideForm::pushButton
method pushButton(var eventInfo Event)
var
    siteForm Form
```

hideToolbar procedure

```
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
```

hideToolbar procedure

Form

Makes the standard Toolbar invisible.

Syntax

```
hideToolbar ( )
```

Description

hideToolbar removes the standard Toolbar from the desktop. You must call **showToolbar** to restore the Toolbar.

Example

In the following example, the **pushButton** method for the *toggleToolbar* button checks whether the Toolbar is showing. If the Toolbar is visible, this method hides it; if the Toolbar isn't visible, this method shows it:

```
; toggleToolbar::pushButton
method pushButton(var eventInfo Event)
if isToolbarShowing() then ; if Toolbar is off
    hideToolbar()           ; hide it
else                        ; otherwise
    showToolbar()           ; show it
endif
endMethod
```

isAssigned method

Form

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just Form.

Example

The following 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 is attached 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
```

isCompileWithDebug method

Form

Reports the status of the Compile With Debug setting.

Syntax

```
isCompileWithDebug ( ) Logical
```

Description

isCompileWithDebug reports the status of the Compile With Debug setting that can be set interactively during form design. **isCompileWithDebug** returns True if Compile With Debug is set in the form; otherwise, it returns False.

Example

In the following example, the central form of a management system has two buttons: **getCompileStatus** and **setCompileStatus**. The **pushButton** method of each button opens the Windows 95 or Windows 98 Explorer dialog to allow a user to select the file that will be examined/manipulated. Each method analyzes the **fileName** selected to determine the file Type and to open the file under the appropriate object type.

The following code is attached to the **pushButton** method for **getCompileStatus**:

```
; getCompileStatus::pushButton
method pushButton(var eventInfo Event)
var
  theForm Form          ; Object variable for forms
  theLibrary Library    ; Object variable for libraries
  theScript Script      ; Object variable for scripts
  fbi FileBrowserInfo    ; File Browser information structure
  selectedFile String    ; FileName selected by user
  fileType String        ; File type of file selected by user
  status Logical         ; Debug status of the selected file
endVar

;Set allowable file types: Forms, Libraries, and Scripts
fbi.AllowableTypes = fbForm + fbLibrary + fbScript
if fileBrowser(selectedFile, fbi) then
  ; The user selected a file
  fileType = upper(substr(selectedFile, selectedFile.size() - 2, 3))
  switch
    case fileType = "FSL" :
      ; Load the Form
```

isDesign method/procedure

```
        theform.load(fbi.Drive + fbi.Path + selectedFile)
        ; Determine its status
        status = theForm.isCompileWithDebug()
        ; Close the Form
        theForm.close()

    case fileType = "LSL" :
        ; Load the Library
        theLibrary.load(fbi.Drive + fbi.Path + selectedFile)
        ; Determine its status
        status = theLibrary.isCompileWithDebug()
        ; Close the Library
        theLibrary.close()

    case fileType = "SSL" :
        ; Load the Script
        theScript.load(fbi.Drive + fbi.Path + selectedFile)
        ; Determine its status
        status = theScript.isCompileWithDebug()
        ; Close the Script
        theScript.close()
    endSwitch
    ; Inform the user
    msgInfo(selectedFile + " compiled with Debug information?", status)
else
    ; The user didn't select a file
    msgInfo("No file selected", "Please try again.")
endIf
endMethod
```

isDesign method/procedure

Form

Reports whether a form is displayed in a Form Design window.

Syntax

```
isDesign ( ) Logical
```

Description

isDesign returns True if a form is displayed in a Form Design window; otherwise, it returns False.

Example

In the following example, **enumFormNames** is used to populate an array ar with the names of the open forms. A for loop then steps through the array and saves the form if it is in design mode.

```
;btnSaveForms :: pushButton
method pushButton(var eventInfo Event)
    var
        ar                Array[] AnyType
        siCounter         SmallInt
        f                  Form
    endVar

    enumFormNames(ar)

    for siCounter from 1 to ar.size()
        f.attach(ar[siCounter])
        if f.getFileName() = "" then
            msgStop("Warning", "At least one form is a new form.")
        else
            if f.isDesign() then
```


isMaximized method/procedure

```
        f.save()
      endif
    endif
  endFor
endMethod
```

isMaximized method/procedure

Form

Reports whether a window is displayed at its maximum size.

Syntax

```
isMaximized ( ) Logical
```

Description

isMaximized returns True if a form is displayed full screen; otherwise, it returns False.

Example

In the following example, the **pushButton** method for the *cycleSize* button (on the current form) opens or attaches to the *Sitenote* form with the variable *siteForm*. If *siteForm* is maximized, this method minimizes it. If *siteForm* is minimized, this method restores it to its previous size with the **show** method. If *siteForm* is neither maximized nor minimized, this method maximizes it:

```
; 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
```

isMinimized method/procedure

Form

Reports whether a window is displayed as an icon.

Syntax

```
isMinimized ( ) Logical
```

isToolBarShowing procedure

Description

isMinimized returns True if a form is displayed as an icon; otherwise, it returns False.

Example

See **isMaximized** example.

isToolBarShowing procedure

Form

Reports whether the standard Toolbar is visible.

Syntax

```
isToolBarShowing ( ) Logical
```

Description

isToolBarShowing returns True if the standard Toolbar is visible; otherwise, it returns False.

Example

See **hideToolBar** example.

isVisible method/procedure

Form

Reports whether any part of a window is displayed.

Syntax

```
isVisible ( ) Logical
```

Description

isVisible 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 top 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
```

keyChar method

Form

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

keyChar 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 in *state* (using

KeyboardStates constants). For Syntax 2, you can specify a string of one or more characters and, optionally, use the KeyboardStates constants to specify a keyboard state.

Example

In the following example, a form named *Otherfrm* is already open and 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*, calls the **keyChar** method for *otherForm*, and passes it a string.

The following 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
```

The following 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
```

keyPhysical method**Form**

Sends an event to a form's **keyPhysical** method.

Syntax

```
keyPhysical ( const aChar SmallInt, const vChar SmallInt, const state SmallInt )
Logical
```

Description

keyPhysical 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 in *state* (using `KeyboardStates` constants).

Example

In the following 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*, and passes it the ANSI code

of the character or keystroke, the virtual ANSI code of the character or keypress, and the keyboard state.

The following code is attached to 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 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

```

load method

Form

Opens a form in the Form Design window.

Syntax

```
load ( const formName String, [const windowStyle LongInt [ , const x LongInt, const y
LongInt, const w LongInt, const h LongInt ] ] ) Logical
```

Description

load opens *formName* in the Form Design window. You have the option to specify in *windowStyle* a WindowStyles constant (or combination of constants). You also have the option to specify (in twips) the window's size and position: arguments *x* and *y* specify the position of the upper-left corner; arguments *w* and *h* specify the width and height, respectively. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

Compare this method to **open**, which opens a form in the Form window. To switch from the Form Design window to the Form window, use **run**. To switch from the Form window to the Form Design window, use **design**.

In either the Form Design window or the Form window, 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 in the Form window, the newly created objects will not automatically be saved when the form is closed.

Notes

- It is possible to load a report as a form. Declare a form as a variable and load a report using it. (For example: `f.load("report.rsl")`)
- 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 call to **sleep**. For more information, see the **sleep** procedure in the System type.

Example

In the following example, the **pushButton** method for a button named *drawABox* loads the *Sitenote* form in a Form 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. In the Form window, 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)
var
    myForm Form

```

maximize method/procedure

```
newObj UIObject
endVar
; open Sitenote in a Form 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
```

maximize method/procedure

Form

Maximizes a window.

Syntax

```
maximize ( )
```

Description

maximize displays a window at its full size. Calling this method is equivalent to clicking Maximize on the Control menu.

Example

In the following example, the **pushButton** method for the *goSites* button opens the *Sitenote* form (assumed to be in the current database), minimizes the current form and 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) and then wait
siteForm.open("Sitenote")
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
```

The following 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
```

menuAction method/procedure**Form**

Sends an event to a form's **menuAction** method.

Syntax

```
menuAction ( const action SmallInt ) Logical
```

Description

menuAction constructs a MenuEvent and calls a specified form's **menuAction** method. *action* is either one of the MenuCommand constants or a user-defined menu constant.

Note

- You can't use **menuAction** to send a MenuCommand constant that is equivalent to a File, New menu choice or a File, Open menu choice. To simulate these choices, call the appropriate ObjectPAL method (e.g., **create** {Form type} or **open** {TableView type}).

Example

In the following 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
```

methodDelete method**Form**

Deletes a form-level method or event from a form.

Syntax

```
methodDelete ( const methodName String ) Logical
```

Description

methodDelete deletes a built-in or custom method or event 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 event 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).

Example

In the following example, two forms are on the desktop in a Form 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 Form 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

```

methodEdit method

Form

Opens a form-level method or event in an Editor window.

Syntax

```
methodEdit (const methodName String) Logical
```

Description

methodEdit opens the method or event specified by **methodName** in an Editor window. If you try to open a method or event that doesn't exist, **methodEdit** will create it for you. **methodEdit** fails if you try to open a method or event that is running.

Example

The following example opens the form's **testMethod** method in an Editor window:

```

method pushButton(var eventInfo Event)
var
  MyForm form
endvar

MyForm.load("vendors.fsl")
MyForm.methodEdit("testMethod")
endMethod

```

methodGet method

Form

Gets a form-level method or event.

Syntax

```
methodGet (const methodName String ) String
```

Description

methodGet gets the text of the built-in or custom form-level method or event 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).

Example

See **methodDelete** example.

methodSet method

Form

Sets the definition of a method or event that is attached to a form.

minimize method/procedure

Syntax

```
methodSet (const methodName String, const methodText String ) Logical
```

Description

methodSet writes the text in *methodText* to the built-in or custom form-level method *methodName* and overwrites any existing method or event 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).

Note

- The method specified by *methodName* does not need to previously exist in the form.

Example

See **methodDelete** example.

minimize method/procedure

Form

Minimizes a window.

Syntax

```
minimize ( )
```

Description

minimize displays a window as an icon. Calling this method is equivalent to choosing Minimize from the Control menu.

Example

See the **maximize** example.

mouseDouble method

Form

Sends an event to a form's **mouseDouble** method.

Syntax

```
mouseDouble ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDouble 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 using KeyBoardStates constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseDouble* on the current form attaches to *Othermse* as *otherForm* and 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
```

mouseDown method

```
        otherForm.mouseDouble(1000, 1000, LeftButton)
    else
        msgStop("Quitting", "Could not find target form.")
    endif
endMethod
```

The following 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
```

mouseDown method

Form

Sends an event to a form's **mouseDown** method.

Syntax

```
mouseDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDown constructs an event 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 using `KeyBoardStates` constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseDown* on the current form attaches to *Othermse* as *otherForm* and 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
```

The following 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

```

mouseEnter method

Form

Sends an event to a form's **mouseEnter** method.

Syntax

```
mouseEnter ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseEnter 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 using KeyBoardStates constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseEnter* on the current form attaches to *Othermse* as *otherForm* and 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

```

The following 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

```

mouseExit method

```
        lastMethod = "mouseEnter"  
        ; get the target and write name to lastTarget field  
        eventInfo.getTarget(targObj)  
        lastTarget = targObj.Name  
    endif  
endMethod
```

mouseExit method

Form

Sends an event to a form's **mouseExit** method.

Syntax

```
mouseExit ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseExit 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 using **KeyboardStates** constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseExit* on the current form attaches to *Othermse* as *otherForm* and 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
```

The following 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 = "mouseExit"  
        ; get the target and write name to lastTarget field  
        eventInfo.getTarget(targObj)  
        lastTarget = targObj.Name  
    endif  
endMethod
```

mouseMove method**Form**

Sends an event to a form's **mouseMove** method.

Syntax

```
mouseMove ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseMove constructs an event 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 using `KeyBoardStates` constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseMove* on the current form attaches to *Othermse* as *otherForm* and 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
```

The following 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
```

mouseRightDouble method**Form**

Sends an event to a form's **mouseRightDouble** method.

Syntax

```
mouseRightDouble (const x LongInt, const y LongInt, const state SmallInt ) Logical
```

mouseRightDown method

Description

mouseRightDouble 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 using `KeyboardStates` constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *send MouseRightDouble* on the current form attaches to *Othermse* as *otherForm* and then calls the **mouseRightDouble** method for *otherForm*.

```
; mouseRightDouble::pushButton
method pushButton(var eventInfo MouseEvent)
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
```

The following 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
```

mouseRightDown method

Form

Sends an event to a form's **mouseRightDown** method.

Syntax

```
mouseRightDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseRightDown 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 using `KeyboardStates` constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseRightDown* on the current form attaches to *Othermse* as *otherForm* and 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
```

The following 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
```

mouseRightUp method**Form**

Sends an event to a form's **mouseRightUp** method.

Syntax

```
mouseRightUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseRightUp 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 using *KeyBoardStates* constants.

Example

In the following 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* and then calls the **mouseRightUp** method for *otherForm*.

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

mouseUp method

```
        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
```

The following 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
```

mouseUp method

Form

Sends an event to a form's **mouseUp** method.

Syntax

```
mouseUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseUp 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 using KeyboardStates constants.

Example

In the following example, the form *Othermse* is open in the Form window. The **pushButton** method for a button named *sendMouseUp* on the current form attaches to *Othermse* as *otherForm* and 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
```



```
endif
endMethod
```

The following 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
```

moveTo method

Form

Moves to a form.

Syntax

```
moveTo ( [const objectName String] ) Logical
```

Description

moveTo moves the focus to a form. Optionally, it moves to the object specified in *objectName*.

Example

In the following example, a form named *Sitenote* is already open in the Form window. 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
```

moveToPage method/procedure**Form**

Displays a specified page of a form.

Syntax

```
moveToPage ( const pageNumber SmallInt ) Logical
```

Description

moveToPage 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 using its object ID, use the **moveTo** method from the UIObject type.

Note

- To access the Page Number for a form, check the PositionalOrder property of the form in the ObjectExplorer. This property can be used in ObjectPAL as well. For example, moveToPage(page#.PositionalOrder).

Example

In the following 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) moves to the second page of the current form, 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
```

open method**Form**

Opens a window.

Syntax

1. open (const *formName* String [, const *windowStyle* LongInt]) Logical
2. open (const *formName* String, const *windowStyle* LongInt, const *x* SmallInt, const *y* SmallInt, const *w* SmallInt, const *h* SmallInt) Logical
3. open (const *openInfo* FormOpenInfo) Logical

Description

open displays the form specified in *formName*. The form is opened in a Form 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 using WindowStyles constants. You can specify more than one window style element by adding the constants together. The following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.open("sales", WinStyleDefault + WinStyleVScroll + WinStyleHScroll)
```

Compare this method with **load**, which opens a form in a Form Design window.

Syntax 3 lets you specify form settings from *openInfo*, a record of type FormOpenInfo. The predefined FormOpenInfo record has the following structure:

<i>x</i> , <i>y</i> , <i>w</i> , <i>h</i>	LongInt	;position and size of the form
<i>name</i>	String	;name of form to open
<i>masterTable</i>	String	;new master table name
<i>queryString</i>	String	;query to run (actual query string)
<i>SQLString</i>	String	;SQL query to run (actual query string)
<i>windowStyle</i>	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. If the query string is an SQL query, replace *queryString* with *SQLString*. Paradox executes the query and opens the form; the result of the query is the master table.

Paradox opens saved forms before delivered forms with the same name. For example, suppose the working directory contains ORDERS.FSL (a saved form) and ORDERS.FDL (a delivered form). The following statement opens the saved form, ORDERS.FSL.

```
ordersForm.open("ORDERS") ; Opens :WORK:ORDERS.FSL.
```

To specify a delivered form, include the .FDL extension.

```
ordersForm.open("ORDERS.FDL") ; Opens the delivered form.
```

In addition to being a table name for a QBE file, the MasterTable field may be the name of a SQL file that produces an Answer table.

FormOpenInfo now has a new field, *SQLString*, which can be used to specify an SQL statement to execute. *SQLString* is of type String.

To rebind a report to a newly created SQL statement, save the SQL statement to a file and specify the filename in ReportPrintInfo.MasterTable or ReportOpenInfo.MasterTable.

Notes

- It is possible to open a report as a form. Declare a form variable and open a report using it. (For example: `f.open("report.rsl")`)
- 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**. For more information, see the **sleep** procedure in the System type.

Example 1

In the following 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)
```

openAsDialog method

```
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
```

Example 2

The following 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
```

openAsDialog method

Form

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 SmallInt, const y SmallInt, const w SmallInt, const h SmallInt) Logical`
3. `openAsDialog (const openInfo FormOpenInfo) Logical`

Description

openAsDialog opens the form *formName* and displays it on top of any other open windows. The form is in the Form window. *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 using `WindowStyles` constants. You can specify more than one window style element by adding the constants. The following code opens a form and specifies both vertical and horizontal scroll bars:

```
theForm.openAsDialog("sales", WinStyleDefault + 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      ; master table name
queryString      String      ; run this query
```

Example

In the following example, the **keyPhysical** method for a field named *fieldOne* tests all key events. When the user presses F1, the form **HELPPFORM** 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")
    helpForm.wait()
    helpForm.close()
    disableDefault                                ; don't call Help system
endIf
endMethod
```

postAction method

Form

Posts an action to an action queue for delayed execution.

Syntax

```
postAction ( const actionId SmallInt )
```

Description

postAction works like **action**, except that the action is not executed immediately. Instead, the action specified by *actionID* is posted to an action queue at the time of the method call; Paradox waits until a yield occurs (e.g., by the current method completing execution or by a call to **sleep**).

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

Example

In the following 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 when Paradox yields:

```
; 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
```

run method

```
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
```

run method

Form

Switches a form from the Form Design window to the Form window.

Syntax

```
run ( ) Logical
```

Description

run switches a form from the Form Design window to the Form window. This method works only with saved forms (.FSL); it does not work with delivered forms (.FDL).

To switch from the Form window to the Form Design window, use **design**.

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 call to **sleep**. For more information, see the **sleep** method in the System type.

Example

The following example opens the *Sitenote* form in a Form Design window, deletes the **pushButton** method from the form and 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()
endif
; won't be permanent
endMethod
```

save method

Form

Saves a form to disk.

Syntax

```
save ( [ const newFormName String ] ) Logical
```

Description

save writes a form to disk in the user's working directory. This method works only when the form is in a Form Design window.

The newFormName argument specifies the name for the form. If the form already has a name, Paradox saves the form using that name. If you omit newFormName and the form doesn't have a name already, this method returns an error.

Example

See the **create** example.

saveStyleSheet method

Form

Saves a style sheet.

Syntax

```
saveStyleSheet ( const fileName String, const overWrite Logical ) Logical
```

Description

saveStyleSheet saves a style sheet to the file specified in *fileName*. If *fileName* does not specify a full path for the style sheet file, this method saves it to the working directory.

The value of *overWrite* specifies what to do if the file already exists. If *overWrite* is True and the file exists, Paradox overwrites the file without asking for confirmation. If *overWrite* is False and the file exists, the file is not saved. This method returns True if it saves the file; otherwise, it returns False.

saveStyleSheet saves the form's current style sheet, including any changes made interactively, or by using ObjectPAL. If called as a method, **saveStyleSheet** operates on the specified form. If called as a procedure, saveStyleSheet operates on the current form. It returns True if successful; otherwise, it returns False.

Example

The following example sets the frame style of field objects and text objects and then saves the form's style sheet to a file named IN3DFRAM.FT. If the file exists, it is overwritten.

```
const
    kOverWrite = Yes
endConst

method mouseClicked(var eventInfo MouseEvent)
    var
        thisForm Form
    endVar

    thisForm.attach()
    thisForm.setProtoProperty(FieldTool, "Frame.Style", Inside3DFrame)
    thisForm.setProtoProperty(TextTool, "Frame.Style", Inside3DFrame)
    thisForm.saveStyleSheet("in3dfram.ft", kOverWrite)
endmethod
```

selectCurrentTool method

Form

Specifies a Toolbar tool to use.

Syntax

```
selectCurrentTool ( const objType SmallInt ) Logical
```

setCompileWithDebug method

Description

selectCurrentTool specifies which Toolbar tool to use, where *objType* is one of the UIObjectTypes constants. When used with a form in the Form Design window, this method makes the specified tool active and sets the mouse shape accordingly.

Example

The following example creates a form and sets the current tool to the Field tool.

```
method pushButton(var eventInfo Event)
  var
    foTest    Form
  endVar

  foTest.create()
  foTest.selectCurrentTool(FieldTool)
  msgInfo("Next step:",
    "Click and drag to draw a field object.")
endMethod
```

setCompileWithDebug method

Form

Sets Compile With Debug.

Syntax

```
setCompileWithDebug ( const yesNo Logical ) Logical
```

Description

setCompileWithDebug sets the Compile With Debug flag to true or false. This is the same as setting Compile With Debug interactively in a Form Design window. **setCompileWithDebug** returns True if successful; otherwise, it returns False.

Example

In the following example, the central form of a management system has two buttons: **getCompileStatus** and **setCompileStatus**. The **pushButton** method of each button opens the Windows 95 or Windows 98 Explorer dialog box to allow a user to select the file that will be examined or manipulated. Each method analyzes the **fileName** selected to determine the **fileType** and opens the file under the appropriate object type.

The following code is attached to the **pushButton** method for **setCompileStatus**:

```
; setCompileStatus::pushButton
method pushButton(var eventInfo Event)
var
  theForm      Form          ; Object variable for forms
  theLibrary   Library       ; Object variable for libraries
  theScript    Script        ; Object variable for scripts
  fbi          FileBrowserInfo ; File Browser information structure
  selectedFile String        ; FileName selected by user
  fileType     String         ; File type of file
                                ; selected by user
  status       Logical        ; Debug status of the selected file
  toggle       String         ; User choice for
endVar

;Set allowable file types: Forms, Libraries, and Scripts
fbi.AllowableTypes = fbForm + fbLibrary + fbScript
if fileBrowser(selectedFile, fbi) then
  ; The user selected a file
```



```

fileType = upper(substr(selectedFile, selectedFile.size() - 2, 3))
switch
case fileType = "FSL" :
    ; Load the Form
    theForm.load(fbi.Drive + fbi.Path + selectedFile)
    ; Determine its status
    status = theForm.isCompileWithDebug()
    toggle = msgYesNoCancel ("Select a choice", selectedFile
        + iif(status, " is ", " is not ") +
        "compiled with Debug information – toggle?")
    switch
    case toggle = "Yes" :
        ; Toggle status
        theForm.setCompileWithDebug(NOT(status))
        ; Save the change
        theForm.save()
        msgInfo("User Notification",
            "Toggle of Debug State Completed.")
    case toggle = "No" or toggle = "Cancel" :
        msgInfo("User Notification",
            "Toggle of Debug State Canceled.")
    endSwitch
    ; Close the Form
    theForm.close()

case fileType = "LSL" :
    ; Load the Library
    theLibrary.load(fbi.Drive + fbi.Path + selectedFile)
    ; Determine its status
    status = theLibrary.isCompileWithDebug()
    toggle = msgYesNoCancel ("Select a choice", selectedFile
        + iif(status, " is ", " is not ")
        + "compiled with Debug information – toggle?")
    switch
    case toggle = "Yes" :
        ; Toggle status
        theLibrary.setCompileWithDebug(NOT(status))
        ; Save the change
        theLibrary.save()
        msgInfo("User Notification",
            "Toggle of Debug State Completed.")
    case toggle = "No" or toggle = "Cancel" :
        msgInfo("User Notification",
            "Toggle of Debug State Canceled.")
    endSwitch
    ; Close the Library
    theLibrary.close()

case fileType = "SSL" :
    ; Load the Script
    theScript.load(fbi.Drive + fbi.Path + selectedFile)
    ; Determine its status
    status = theScript.isCompileWithDebug()
    toggle = msgYesNoCancel ("Select a choice", selectedFile
        + iif(status, " is ", " is not ")
        + "compiled with Debug information – toggle?")
    switch
    case toggle = "Yes" :
        ; Toggle status
        theScript.setCompileWithDebug(NOT(status))
        ; Save the change

```

setIcon method/procedure

```
        theScript.save()
        msgInfo("User Notification",
            "Toggle of Debug State Completed.")
        case toggle = "No" or toggle = "Cancel" :
            msgInfo("User Notification",
                "Toggle of Debug State Canceled.")
        endSwitch
        ; Close the Script
        theScript.close()
    endSwitch
    ; Inform the user
    msgInfo(selectedFile
        + " compiled with Debug information?",
        status)
else
    ; The user didn't select a file
    msgInfo("No file selected", "Please try again")
endIf
endMethod
```

setIcon method/procedure

Form

Specifies the icon to be used with a form, report, or desktop.

Syntax

```
setIcon ( const fileName String ) Logical
```

Description

setIcon specifies the icon to be used with a form, report, or desktop. The file specified with *fileName* must be a valid icon file and the file's name must have an extension of .ICO. **setIcon** returns True if successful; otherwise it returns False.

After you set the icon for a form, all the forms on the desktop will change to the new icon and any form that is opened will be set to the new icon.

Example

The following example sets the file, DOCFILE.ICO as the icon.

```
method init ( var eventInfo Event )
    setIcon ( "i:\\resource\\docfile.ico" )
endMethod
```

setMenu method

Form

Associates a menu with a form.

Syntax

```
setMenu ( const menuVar Menu )
```

Description

setMenu associates the menu specified in *menuVar* with a form. This method performs the same function as the Menu type show, and adds the following features:

- when the form gets focus, Paradox displays the associated menu
- actions that result from choices from that menu are sent to that form

Example

The following example is a script. It opens a form, builds a simple menu and then uses **setMenu** to assign the menu to the form:

```
method run(var eventInfo Event)
  var
    foOrders    Form
    muOrderForm Menu
    puFormFile  PopUpMenu
  endVar

  ; Build a menu for the form.
  foOrders.open("orders")

  ; Setting the StandardMenu property to False
  ; (either in ObjectPAL code or interactively)
  ; can reduce flicker when changing menus.
  foOrders.StandardMenu = False

  puFormFile.addText("&New Form", MenuEnabled, MenuFormNew)
  puFormFile.addText("&Open Form", MenuEnabled, MenuFormOpen)
  puFormFile.addText("&Exit", MenuEnabled, MenuFileExit)

  muOrderForm.addPopUp("&File", puFormFile)

  foOrders.setMenu(muOrderForm)

endMethod
```

setPosition method/procedure**Form**

Positions a window on screen.

Syntax

```
setPosition ( const x LongInt, const y LongInt, const w LongInt, const h LongInt )
```

Description

setPosition positions a window on screen. 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).

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper-left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox desktop application, the position is given relative to the entire screen; for forms, reports, and Table windows, the position is given relative to the Paradox desktop.

Note

- You may want to use the keyword **self** with **setPosition**. When positioning a form, however, you may not use **self.setPosition(x,y,w,h)**. **self.setPosition(x,y,w,h)** calls the UIObject version of **setPosition** which will not set a form's position. Instead, you must explicitly define a form variable and use it.

Example

See the **getPosition** example.

setProtoProperty method/procedure**Form**

Sets the value of a specified property of a prototype object.

Syntax

```
setProtoProperty ( const objectType SmallInt, propertyName String, value AnyType )
Logical
```

Description

setProtoProperty sets the property specified in *propertyName* of the prototype object specified in *objectType* to the value specified in *value*. To specify *objectType*, use one of the UIObjectType constants. If called as a method, **setProtoProperty** operates on prototype objects in the style sheet of the specified form. If called as a procedure, **setProtoProperty** uses the style sheet of the current form.

Changes to the style sheet are not saved automatically. You must either save the style sheet interactively or call **saveStyleSheet**.

Example

See the **saveStyleSheet** example.

setSelectedObjects method**Form**

Selects specified objects in a form.

Syntax

```
setSelectedObjects ( var objects Array[ ] UIObject, const yesNo Logical )
```

Description

setSelectedObjects selects specified objects in a form in a Form Design window as if you had selected the objects interactively. The array *objects* is an array of available UIObjects (not the object names). Use **attach** to assign a UIObject to an array.

The argument *yesNo* specifies whether to show selection handles. If *yesNo* is True, the selected objects have handles; otherwise, they do not.

Example

The following example creates a form, creates two boxes in it, and calls **setSelectedObjects** to select the boxes. You must use **attach** to assign a UIObject to an array.

```
method pushButton(var eventInfo Event)
  var
    foTemp      Form
    uiTemp      UIObject
    arObjects   Array[2] UIObject
  endVar

  const
    kOneInch = 1440 ; One inch = 1,440 twips.
    kShowHandles = Yes
  endConst

  foTemp.create()

  uiTemp.create(BoxTool, 300, 300, kOneInch, kOneInch, foTemp)
  uiTemp.Visible = Yes
  arObjects[1].attach(uiTemp)

  uiTemp.create(BoxTool, 300, 2200, kOneInch, kOneInch, foTemp)
```

setStyleSheet method/procedure

```
    uiTemp.Visible = Yes
    arObjects[2].attach(uiTemp)

    foTemp.setSelectedObjects(arObjects, kShowHandles)
endMethod
```

setStyleSheet method/procedure

Form

Specifies a form's style sheet.

Syntax

```
setStyleSheet ( const fileName String )
```

Description

setStyleSheet makes a form use the style sheet specified in *fileName*. If *fileName* does not specify a full path to the style sheet, this method searches for it in the working directory. If called as a method, **setStyleSheet** operates on the specified form. If called as a procedure, it operates on the current form.

Any UIObjects created in the form while the style sheet is active will have the properties and methods of the corresponding prototype objects in the style sheet. **setStyleSheet** does not change the properties or methods of UIObjects that already exist. This method affects only the specified form; it does not affect the screen or printer style sheets. Use the System procedures **setDefaultScreenStyleSheet** and **setDefaultPrinterStyleSheet** to set the properties of the screen and printer style sheets.

Example

The following example opens a form and then calls **getStyleSheet** to see which style sheet the form is using. If the style sheet is not COREL.FT, the code calls **setStyleSheet** to set it and then calls **getStyleSheet** again to make sure it was set successfully. **setStyleSheet** requires double backslashes in the path, but **getStyleSheet** returns single backslashes.

```
method pushButton(var eventInfo Event)
    var
        f Form
    endVar
    f.open("orders")
    ; Get and set the style sheet for this form.
    if f.getStyleSheet() "c:\\Corel\\Paradox\\Corel.ft" then
        f.setStyleSheet("c:\\Corel\\Paradox\\Corel.ft")
        if f.getStyleSheet() "c:\\Corel\\Paradox\\Corel.ft" then
            msgStop("Problem", "Could not set the style sheet.")
        endIf
    endIf
endMethod
```

setTitle method/procedure

Form

Sets the text in the Title Bar of the window.

Syntax

```
setTitle ( const text String )
```

Description

setTitle changes the text of the window's Title Bar to the text specified in *text*. The maximum length of text is 78 characters. If you change a form's title, remember to use the new title when you want to attach anything to that form. For more information, see the description of **attach**.

show method/procedure

Example

See the **openAsDialog** example.

show method/procedure

Form

Displays a minimized window at its previous size; makes a hidden form visible.

Syntax

```
show ( )
```

Description

show makes a hidden form visible. **show** also restores a minimized window to the size it was before it was minimized. This method is similar to the Restore command on the Control menu.

show doesn't make a form the top window; use **bringToTop** to make a form the top layer and give it focus.

Example

See the **hide** example.

showToolbar procedure

Form

Makes the standard Toolbar visible.

Syntax

```
showToolbar ( )
```

Description

showToolbar displays the standard Toolbar.

Example

See the **hideToolbar** example.

wait method

Form

Suspends execution of a method.

Syntax

```
wait ( ) AnyType
```

Description

wait suspends execution of the current method until the form you're waiting for returns (see **formReturn**). This method is used to 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. After 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 to allow the code on the calling form to examine the it (e.g., to see settings on a dialog box).

Note

- A form cannot wait on itself.

Example

See the **formReturn** example.

windowClientHandle method/procedure**Form**

Returns the handle of a window.

Syntax

```
windowClientHandle ( ) LongInt
```

Description

A window handle is a unique integer identifier that is assigned to a window by Windows.

windowClientHandle returns an integer value that represents 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).

Example

In the following example, assume that a dynamic link library (DLL) called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle. Because *doSomething* is not an ObjectPAL method, information about the method must be declared in the Uses window. The following code defines the prototype information for *doSomething* and appears in the Uses window at the form level:

```
;Form1::Uses

Uses MYTEST

    doSomething(wHandle CLONG)

EndUses
```

The following code appears in the pushButton method on the form:

```
; someButton::pushButton
method pushButton(var eventInfo Event)
doSomething(windowClientHandle()) ; call doSomething and supply the
                                   ; handle of the client portion
                                   ; of the current form

endMethod
```

windowHandle method/procedure**Form**

Returns the handle of a window.

Syntax

```
windowHandle ( ) LongInt
```

Description

A window handle is a unique integer identifier that is assigned to a window by Windows.

windowHandle returns an integer value that represents the window handle of a form. When called as a procedure, windowHandle 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).

Example

In the following example, assume that a (DLLidh_pglos_DLL) called MYTEST.DLL exists and that it contains a function called *doSomething*. The *doSomething* function takes one argument, a window handle. Because *doSomething* is not an ObjectPAL method, information about the method must be

writeText method

declared in the Uses window. The following code defines the prototype information for *doSomething* and appears in the Uses window at the form level:

```
;Form1::Uses  
  
Uses MYTEST  
  
    doSomething(wHandle CLONG)  
EndUses
```

The following code appears in the **pushButton** method on the form:

```
; someButton::pushButton  
method pushButton(var eventInfo Event)  
    doSomething(windowHandle()) ; call doSomething and supply the  
                                ; window handle of the current form  
endMethod
```

writeText method

Form

Writes text contents of a form to file.

Syntax

```
writeText (const filename      String ) Logical
```

Description

writeText writes all text displayed on a form to the disk file specified by *filename*. This method attempts to keep the relative position of all text constants within the text file, but does not write the character or point size attributes. For forms with multiple pages, all text is written to the file, with the latter pages appended to the bottom of the file. This method writes only text. It does not write chart, graphic or OLE field information to the file.

Example

The following example writes the contents of the form BIOLIFE.FSL to the text file test.txt . This example assumes that a form named BIOLIFE.FSL already exists in the working directory.

```
method pushButton(var eventInfo Event)  
  
    var  
    f form  
    endvar  
  
    if not f.open("BIOLIFE") then ; attempts to open Biolife.fsl. If not successful,  
    alerts t                                ; the user and returns to the form  
  
    msginfo("stop","could not open Biolife form")  
    return  
endif  
    f.attach("BIOLIFE")                ; attaches form variable to biolife.fsl  
    f.writetext(":WORK:test.txt")      ; writes the displayed contents of biolife.fsl to  
    the file test.txt  
    f.close()                          ; closes biolife.fsl  
endMethod
```

Graphic type

A Graphic variable provides a handle that is used to manipulate 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), Joint Photographic Experts Group (JPG or JPEG), Paintbrush (PCX), and tagged information file format (TIF).

You can use Graphic type methods **readFromClipboard**, **writeToClipboard**, **readFromFile**, and **writeToFile** to transfer bitmaps between forms (and reports), tables, the Clipboard, and disk files.

The Graphic type includes several derived methods from the AnyType type.

Methods for the Graphic type

AnyType	←	Graphic
blank		readFromClipboard
dataType		readFromFile
isAssigned		writeToClipboard
isBlank		writeToFile
isFixedType		

readFromClipboard

Graphic type

Reads a graphic from the Clipboard.

Syntax

readFromClipboard () Logical

Description

readFromClipboard 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 the following 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 and 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
```

readFromFile

readFromFile

Graphic type

Reads a graphic from a file.

Syntax

```
readFromFile ( const fileName String ) Logical
```

Description

readFromFile 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)
- Joint Photographic Experts Group (JPEG)
- Encapsulated Postscript (EPS)
- Graphic Interchange Format (GIF)
- Paintbrush (PCX)
- 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 folder 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 folder,
; and write it to the bitmapField graphic
if chessBmp.readFromFile("c:\\windows\\chess.bmp") then
    bitmapField = chessBmp
endIf
endMethod
```

writeToClipboard method

Graphic

Writes a bitmap to the Clipboard.

Syntax

```
writeToClipboard ( ) Logical
```

Description

writeToClipboard writes a bitmap to the Clipboard. **writeToClipboard** returns True if successful; otherwise, it returns False. Formats copied to the 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* and 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

```

writeToFile method

Graphic type

Writes a bitmap to a file.

Syntax

```
writeToFile ( const fileName String ) Logical
```

Description

writeToFile writes a bitmap to a disk file specified in *fileName*. If *fileName* does not specify a path, this method writes to the working directory (:WORK:). **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* and then writes *chessBmp* to a file named CHESS1.BMP in the working directory.

```

; 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

```

KeyEvent type

The keyevent type provides methods for getting and setting information about keystroke events, including

- Characters sent to the program: `char`, `charAnsiCode`, `vChar`, `vCharAnsiCode`, `setChar`, `setVChar`
- Status of Alt, Ctrl, and Shift: `isAltKeyDown`, `setAltKeyDown`, `isControlKeyDown`, `setControlKeyDown`, `isShiftKeyDown`, `setShiftKeyDown`.

Keyboard events and event methods

When you press a key in Paradox, one of the following happens:

- Windows intercepts the keystroke and does something.

writeToFile method

- Windows passes the keystroke to Paradox, which does something.
- Windows passes the keystroke to Paradox, which passes it to the active object.

When Windows intercepts a keystroke, ObjectPAL does not see it. In the other two cases, you can use the event methods `keyPhysical`, `action`, and `keyChar`, to respond to and simulate keyboard events. These event methods are closely related.

When a key is pressed, Windows gets the keystroke from the keyboard driver and stores it in the system message queue. If the keystroke is meaningful to Windows (for example, Ctrl+F4, which closes the active window), Windows performs the associated action. Otherwise, Windows sends the keystroke to Paradox. Paradox generates a `KeyEvent` packet and sends it to the form's `keyPhysical` event method. If the keystroke corresponds to a Paradox action (for example, F9, which toggles Edit mode on and off), Paradox calls the form's action event method with the appropriate constant (such as `DataToggleEdit`).

If the keystroke is not intercepted by Windows or translated to an action by Paradox, the form's `keyPhysical` event method passes the event packet to its `keyChar` event method, which by default passes it to the `keyChar` event method of the active object. The active object handles the keystroke or bubbles it to its container, and so on, up through the containership hierarchy to the form. If the keystroke is a menu shortcut (for example, Alt+F, which displays the File menu), the form passes it back to Windows for processing.

Paradox uses virtual key codes to map keyboard keys to integer values. For more information, see [Keys and virtual key codes](#).

The following built-in event methods are triggered by the `KeyEvent`s **`keyChar`** and **`keyPhysical`**.

The `KeyEvent` type includes several derived methods from the `Event` type.

Methods for the `KeyEvent` type

Event	←	KeyEvent
<code>errorCode</code>		<code>char</code>
<code>getTarget</code>		<code>charAnsiCode</code>
<code>isFirstTime</code>		<code>isAltKeyDown</code>
<code>isPreFilter</code>		<code>isControlKeyDown</code>
<code>isTargetSelf</code>		<code>isFromUI</code>
<code>reason</code>		<code>isShiftKeyDown</code>
<code>setErrorCode</code>		<code>setAltKeyDown</code>
<code>setReason</code>		<code>setChar</code>
		<code>setControlKeyDown</code>
		<code>setShiftKeyDown</code>
		<code>setVChar</code>
		<code>setVCharCode</code>
		<code>vChar</code>
		<code>vCharCode</code>

Keys and virtual key codes

Paradox uses virtual key codes to map keyboard keys to integer values. For example, the virtual key code for Tab is 9, and the virtual key code for the letter A is 65. ObjectPAL provides keyboard constants for virtual key codes, so you don't have to remember numeric values. For example, the `Keyboard`

constant for Tab is `VK_TAB` (VK stands for virtual key code). ObjectPAL does not provide keyboard constants for alphanumeric characters because they're easy to remember and type directly.

Although the keyboard constants are defined to represent integer values, you can use them as quoted strings with certain methods. For example, the following code (attached to an unbound field object) displays `VK_TAB` as a string when you press Tab.

```
method keyPhysical(var eventInfo KeyEvent)
  x = eventInfo.vChar()
  x.view()
endMethod
```

In general, you can work with virtual key codes as integers or strings, depending on your needs, personal preference, or the syntax of the method you're calling. In some cases, you may need to convert an integer key code to a string, or a string to an integer key code. The following table lists the procedures ObjectPAL provides for this purpose (they're defined for the String type).

Method	Description
<code>chrToKeyName</code>	Returns the key name of the character contained in a string
<code>VRCodeToKeyName</code>	Returns the key name corresponding to a specified ANSI code
<code>KeyNameToChr</code>	Returns a string of length 1 containing the ANSI code for the key name
<code>KeyNameToVKCode</code>	Returns a <code>SmallInt</code> representing the ANSI code for the given key name

When the user presses Ctrl+Break the following expressions are equivalent.

```
ChrToKeyName(Chr(VK_CANCEL))      and   "VK_CANCEL"
ChrToKeyName(eventInfo.vChar())   and   VK_CANCEL

VKCodeToKeyName(VK_CANCEL)        and   "VK_CANCEL"
VKCodeToKeyName(eventInfo.vCharCode()) and   "VK_CANCEL"

KeyNameToChr("VK_CANCEL")         and   Chr(VK_CANCEL)
KeyNameToChr("VK_CANCEL")         and   eventInfo.vChar()

KeyNameToVKCode("VK_CANCEL")      and   VK_CANCEL
KeyNameToVKCode("VK_CANCEL")      and   eventInfo.vCharCode()
```

char method

KeyEvent

Returns the character associated with a keystroke.

Syntax

```
char ( ) String
```

Description

char returns the character associated with a keystroke. For example, if you type a, **char** returns a. If you press SHIFT + A, **char** returns A. If a keystroke results in an unprintable character, **char** returns an empty string ("").

char is the easiest way to check for an alphanumeric keystroke when case matters. If case doesn't matter, use **vChar** to test against the string value of a virtual key code. For example, 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 either a lowercase a or an uppercase A).

charAnsiCode

Example

The following example displays the character typed into a field object as a message at the bottom of the screen. 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
```

charAnsiCode

KeyEvent

Returns the ANSI value associated with a keystroke.

Syntax

```
charAnsiCode ( ) SmallInt
```

Description

charAnsiCode returns an integer that represents the ANSI value associated with a keystroke. 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.

Example

The following 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
```

isAltKeyDown method

KeyEvent

Reports whether ALT was held down during a KeyEvent.

Syntax

```
isAltKeyDown ( ) Logical
```

Description

isAltKeyDown 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

```

isControlKeyDown method

KeyEvent

Reports whether *CTRL* was held down during a KeyEvent.

Syntax

```
isControlKeyDown ( ) Logical
```

Description

isControlKeyDown returns True if CTRL was held down at the time a KeyEvent occurred; otherwise, it returns False.

Example

See the **setControlKeyDown** example.

isFromUI method

KeyEvent

Reports whether an event was generated by the user interacting with Paradox.

Syntax

```
isFromUI ( ) Logical
```

Description

isFromUI reports whether a KeyEvent was generated either by the user interacting with Paradox or internally (e.g., by an ObjectPAL statement). This method returns True only for the first KeyEvent generated by a keystroke; for subsequent events and actions, it returns False.

Example

The following example shows how to put one of two messages on the Status Bar depending on whether a character is put in a field by a user or by ObjectPAL. This method returns True for user actions (including **sendKeys**, which mimics user input). It returns False with all other ObjectPAL methods, including **keyPhysical**.

The following code is attached to the **pushButton** method of a button named *btnAutoFill*. This method sends the character a to the field *fldPassword*:

```

; btnAutofill :: pushButton
method pushButton(var eventInfo Event)
    fldPassword.keyPhysical(97, 97, Shift) ; send an "a"
endMethod

```

The following code is attached to the **keyPhysical** method of a field named *fldPassword*. This method sends one of two messages depending on whether the user typed in a character or used the *btnAutofill* button:

```

; fldPassword :: keyPhysical
method keyPhysical(var eventInfo KeyEvent)

```

isShiftKeyDown method

```
if eventInfo.isFromUI() then
    message("Try using the autofill button.")
else
    message("Automatically typing value.")
endif
endMethod
```

isShiftKeyDown method

KeyEvent

Reports whether SHIFT was held down during a KeyEvent.

Syntax

```
isShiftKeyDown ( ) Logical
```

Description

isShiftKeyDown returns True if *SHIFT* was held down at the time a KeyEvent occurred; otherwise, it returns False.

Example

See the **setShiftDown** example.

setAltKeyDown method

KeyEvent

Simulates pressing and holding *ALT* during a KeyEvent.

Syntax

```
setAltKeyDown ( const yesNo Logical )
```

Description

setAltKeyDown 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 and 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

```

setChar method

KeyEvent

Specifies an ANSI character for a KeyEvent.

Syntax

```
setChar ( const char String )
```

Description

setChar sets a KeyEvent to have an ANSI character based on the value of *char*, where *char* evaluates to single character string (e.g., a).

Example

The following example attaches code 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 keystrokes normally
endMethod

```

setControlKeyDown method

KeyEvent

Simulates pressing and holding CTRL during a KeyEvent.

Syntax

```
setControlKeyDown ( const yesNo Logical )
```

Description

setControlKeyDown adds information about the state of *CTRL* to eventInfo for 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

```

setShiftKeyDown method

```
    ; code here executes just for form itself
endif
endMethod
```

To simulate *CTRL* + *C*, the code for this method creates a *KeyEvent* variable and 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
```

setShiftKeyDown method

KeyEvent

Simulates pressing and holding *SHIFT* during a *KeyEvent*.

Syntax

```
setShiftKeyDown ( const yesNo Logical )
```

Description

setShiftDown 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, 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
```

setVChar method**KeyEvent**

Specifies a Windows virtual character for a KeyEvent.

Syntax

```
setVChar ( const char String )
```

Description

setVChar specifies in *char* a one-character string for a KeyEvent. Use **setVChar** with an uppercase letter or a Keyboard constant to specify a code string for a single letter, but use the constant as a quoted string instead of an integer value. In the following example, the code statement specifies a tab character:

```
eventInfo.setVChar("VK_TAB")
```

The virtual character code string for any letter is the uppercase letter. For example, the virtual character code string for the letter k is K (uppercase only).

Example

See the **setAltKeyDown** example or the **chrToKeyName** example.

setVCharCode**KeyEvent**

Specifies a Windows virtual character for a KeyEvent.

Syntax

```
setVCharCode ( const VK_Constant SmallInt )
```

Description

setVCharCode uses a Keyboard constant in *VK_Constant* to specify a Windows virtual character for a KeyEvent.

Example

The following example attaches code 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
```

vChar**KeyEvent**

Returns a Windows virtual character.

Syntax

```
vChar ( ) String
```

vChar

Description

vChar returns a Windows virtual key name as a string. Use Keyboard constants to find out which Windows virtual character was returned, but use the constants as quoted strings instead of integer values. In the following example, the statements are equivalent (they both beep when you press Return). The first statement uses **vCharCode** and the constant VK_RETURN to test for an integer value, the second statement uses **vChar** and VK_RETURN to test for a string value.

```
if vCharCode = VK_RETURN then beep() endif
if vChar = "VK_RETURN" then beep() endif
```

Example

In the following example, assume a form contains a box named *boxOne*. When the user presses a movement key, this code moves *boxOne* in increments of 100 twips. If *SHIFT* is held down in combination with a movement key, *boxOne* moves 1000 twips. Because **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 keystroke
    boxOne.position = Point(x,y)
    message("Value of vChar() was " + kp)

else
    ;code here executes just for form itself
endif
endMethod
```

vCharCode**KeyEvent**

Returns the integer value of a Windows virtual character.

Syntax

```
vCharCode ( ) SmallInt
```

Description

vCharCode returns the integer value of a Windows virtual character. Use Keyboard constants to find out which Windows virtual character the integer value represents.

Example

For the following example, assume a form has a field named *thisField*. When the user types a value in *thisField* and presses Return, 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
  qVar 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
  qVar = Query

      c:\Core1\Paradox\samples\biolife.db|Common Name |Species Name|
      |check ~cName|check|

endQuery

; run query, write contents to myFish table
qVar.executeQBE("myFish.db")
tv.open("myFish")          ; view myFish view
endif
endMethod
```

Library type

A library is a Paradox object that stores custom methods, custom procedures, variables, constants, and user-defined data types. Libraries are used to store and maintain frequently-used routines and to share custom methods and variables among several forms.

In many ways, working with a library is like working with a form. Like a form, a library has built-in event methods. You add code to a library, just as you do to a form, by using the Object Explorer and the ObjectPAL Editor. (However, you can't place design objects in the library.) As with a form, you can open Editor windows to declare custom ObjectPAL methods, procedures, variables, constants, data types, and external routines.

The Library type includes several derived methods from the Form type.

Methods for the Library type

Form	←	Library
deliver		close
isCompileWithDebug		create
isAssigned		enumSource
load		enumSourceToFile
methodDelete		execMethod
methodGet		methodEdit
methodSet		open
save		setCompileWithDebug

close method

Library

Closes a library.

Syntax

```
close ( )
```

Description

close closes a library and ends the association between a Library variable and the underlying library file.

Example

The following 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 and then calls **close** to end the association between the variable and the library. Another call to **open** associates *lib* with the library KIT.LSL to make 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
```

create method

endMethod

create method

Library

Creates a library.

Syntax

create () Logical

Description

create creates a blank library and leaves it in a design window. You can use **methodSet** (derived from the Form type) to alter or add methods in the new library.

Example

The following example uses **create** to make a new library, adds a custom method to it with **methodSet**, save the library with **save** and then close the library.

```
; btnCreateLibrary :: pushButton
method pushButton(var eventInfo Event)
  var
    lib    Library
  endVar

  ;Create library.
  lib.create()
  lib.methodSet("cmMessage", "method cmMessage()
    msgInfo(\"From new library\", \"Hello World!\") endMethod")
  lib.save("library")
  lib.close()
endmethod
```

enumSource method

Library

Writes the code from a library to a Paradox table.

Syntax

enumSource (const *tableName* String [, const *recurse* Logical])

Description

enumSource lists, in the Paradox table specified in *tableName*, all the custom code (e.g., methods, procedures, and variables) stored in a library. If the table does not exist, Paradox creates it in the working directory; if the table does exist, information is appended to the table.

The structure of the table is:

Field name	Type	Size
Object	A	128
MethodName	A	128
Source	M	64

enumSourceToFile method

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 **open** or **load** the library before calling this method.

Example

The following example declares a Library variable named *lib*, calls **open** to associate *lib* with the library TOOLS.LSL, and 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.DB—
  ; goes to :WORK: by default
  lib.enumSource("LIBSRC.DB")

else
  msgStop("TOOLS.LSL", "Could not open library.")
endif

endMethod
```

enumSourceToFile method

Library

Writes the code from a library to a text file.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] )
```

Description

enumSourceToFile lists all the custom code (e.g., methods, procedures, and variables) stored in a library to the text file specified in *fileName*. If the file does not exist, Paradox creates it. If the file does exist, Paradox overwrites it without asking for confirmation. If *fileName* contains no path or alias, the file is created in the working directory.

In the text file, comment lines are used to identify and mark the beginning and end of each method, procedure, or variable. The following example shows the code for a library's built-in **open** method:

```
;|BeginMethod|#Library1|open|
method open(var eventInfo Event)
var
  myMsgTCursor Tcursor
endVar
if not myMsgTCursor.open("Msghelp.db") then
  msgStop("Error", "Couldn't open MsgHelp.db")
  fail()
endif
```


execMethod

```
endMethod  
;|EndMethod|#Library|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** or **load** the library before calling this method.

Example

The following example declares a Library variable named *lib*, calls **open** to associate *lib* with the library *TOOLS.LSL*, and 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
```

execMethod

Library

Calls a custom method that takes no arguments.

Syntax

```
execMethod ( const methodName String )
```

Description

execMethod 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.

Example

The following example creates an array of three items, each of which 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
```

isAssigned method

```
for i from 1 to libMethods.size()
  lib.execMethod(libMethods[i])
endFor
else
  msgStop("TOOLS.LSL", "Could not open library.")
endif
```

isAssigned method

Library

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just Library.

Example

The following example uses **isAssigned** to test the value of *i* before assigning a value to it. If it has been assigned, this code increments *i* by one. The following code is attached 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
```

methodEdit method

Library

Opens a library's method in an Editor window.

Syntax

```
methodEdit (const methodName String) Logical
```

Description

methodEdit opens the method specified by *methodName* in an Editor window. If you specify a method that doesn't exist, **methodEdit** will create it for you. **methodEdit** fails if you try to open a method that is running.

Example

The following example opens the library's **testMethod** method in an editor window:

open method

```
method pushButton(var eventInfo Event)
var
    MyLib library
endvar

MyLib.load("Main.lsl")
MyLib.methodEdit("testMethod")
endMethod
```

open method

Library

Associates a Library variable with a library and makes the library code available.

Syntax

```
open ( const libraryName String [ , const libScope SmallInt ] ) Logical
```

Description

open associates a Library variable with a library and makes the library code, variables, constants, and type declarations available to the form. Variables declared in the library can be kept private to the form, or they can be shared with other forms and libraries that have opened this library, depending on the value of *libScope*. ObjectPAL defines the following two LibraryScope constants to specify the scope of variables declared in the library:

- PrivateToForm specifies that each form that opens the library has its own copy of the variables.
- GlobalToDesktop specifies that every form in the desktop (Paradox session) that opens the library shares the variables declared in the library.

To open a library and make its variables available to every form that opens the library in the current session of Paradox, use the constant GlobalToDesktop. The following example 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

The following example shows how two forms can open a library global to the desktop and share the library. The following code is attached to a form's built-in **open** method, and opens *libOne* private to the form. *libOne* cannot be shared. *libTwo* is opened global to the desktop and can be shared. *libOne* and *libTwo* are library variables that have been declared in the **var** block of the form.

```
; formOne::open
method open(var eventInfo Event)

if eventInfo.isPreFilter()
then
    ; code here executes for each object in the form
```

logical procedure

```
else
    ; code here executes just for the form itself

    libOne.open("TOOLS.LSL", PrivateToForm)    ; no sharing variables
                                              ; with other forms
    libTwo.open("KIT.LSL", GlobalToDesktop)    ; can be shared
                                              ; with other forms

endIf
endMethod
```

The following code is attached to another form’s built-in **open** method. This code calls open to open the library KIT.LSL global to the desktop. This form and the previous form can now share KIT.LSL. kitLib is a library variable declared in the **var** block of the form.

```
; formTwo::open
method open(var eventInfo Event)

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
```

Logical type

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:

- Did that statement execute successfully?
- Is that table empty?
- Is that form displayed as an icon?

The Logical type includes several derived methods from the AnyType type.

Methods for the Logical type

AnyType	←	Logical
blank		logical
dataType		
isAssigned		
isBlan		
isFixedType		
view		

logical procedure

Logical

Casts a value as type Logical.

Syntax

```
logical ( const value AnyType ) Logical
```

Description

logical casts value to the data type Logical. If value is a numeric data type, non-zero values evaluate to True and zero evaluates to False. 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.

Example

In the following 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
```

LongInt type

LongInt values are long integers; that is, they can be represented by a long series of digits. A LongInt variable occupies four bytes. ObjectPAL converts LongInt values to range from -2,147,483,648 to 2,147,483,647. The following example attempts to assign a value outside of this range to a LongInt variable causes an error:

```
var
    x, y, z LongInt
endVar

x = 2147483647 ; The upper limit value for a LongInt variable.
y = 1
z = x + y      ; This statement causes an error.
```

When ObjectPAL performs an operation on LongInt values, it expects the result to be a LongInt. That's why the addition operation in the previous example causes an error: the result is too large to be a LongInt. To work with a boundary value (in either the positive or negative direction), you must convert the value to a type that can accommodate it. In the following example, ObjectPAL converts one LongInt to a Number before doing the addition, and the statement succeeds. This example also assigns the result to a Number variable (which can handle the large value), instead of assigning it to a LongInt variable (which could not).

```
var
    x, y LongInt
    z      Number ; Declare z as a Number so it can hold the result.
endVar

x = 2147483647 ; The upper limit value for a LongInt variable.
y = 1
z = Number(x) + y ; This statement succeeds.
```

bitAnd method

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 following table displays the methods for the LongInt type, including several derived methods from the Number and AnyType types.

Methods for the LongInt type

AnyType	←	Number	←	LongInt
blank		abs		bitAND
dataType		acos		bitIsSet
isAssigned		asin		bitOR
isBlank		atan		bitXOR
isFixedType		atan2		LongInt
view		ceil		
		cos		
		cosh		
		exp		
		floor		
		fraction		
		fv		
		ln		
		log		
		max		
		min		
		mod		
		number		
		numVal		
		pmt		
		pow		
		pow10		
		pv		
		rand		
		round		
		sin		
		sinh		
		sqrt		
		tan		
		tanh		
		truncate		

bitAnd method	LongInt
Performs a bitwise AND operation on two values.	

bitIsSet method

Syntax

```
bitAND ( const value LongInt ) LongInt
```

Description

bitAND returns the result of a bitwise AND operation on value. **bitAND** operates on the binary representations of two integers and compares them one bit at a time. The truth table for **bitAND** is:

A	b	a bitAND b	a	b	a bitAND b
0	0	0	0	1	0
1	0	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
```

bitIsSet method

LongInt

Reports whether a bit is 1 or 0.

Syntax

```
bitIsSet ( const value LongInt ) Logical
```

Description

bitIsSet examines the binary representation of an integer and reports 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, 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 .

Example 1

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 and assigns the result to **bitNum**. This 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)
```

bitOR method

```
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
```

Example 2

The following 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
```

bitOR method

LongInt

Performs a bitwise OR operation on two values.

Syntax

```
bitOR ( const value LongInt ) LongInt
```

Description

bitOR returns the result of a bitwise OR operation on *value*. **bitOR** operates on the binary representations of two integers and compares them one bit at a time. Here is the truth table for **bitOR**:

bitXOR method

a	b	a bitAND b	a	b	abitORB
0	0	0	0	1	1
1	0	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 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
```

bitXOR method

LongInt

Performs a bitwise XOR operation on two values.

Syntax

```
bitXOR ( const value LongInt ) LongInt
```

Description

bitXOR performs a bitwise XOR (exclusive OR) operation on *value*. **bitXOR** operates on the binary representations of two integers and compares them one bit at a time. Here is the truth table for **bitXOR**:

a	b	a bitXOR(b)	a	b	a bitXOR(b)
0	0	0	0	1	1
1	0	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
```

LongInt procedure

LongInt

Casts a value as a LongInt.

Syntax

```
LongInt ( const value AnyType ) LongInt
```

Description

LongInt casts the data type of value to a long integer. If you convert from a more precise type (e.g., Number), precision may be lost.

Example

The following example assigns a number to x, casts x to **LongInt**, and assigns the result to l. Notice that the decimal precision of x is lost when it is cast as a **LongInt** and assigned to l.

```
; convertToInt::pushButton
method pushButton(var eventInfo Event)
var
  x Number
  y LongInt
endVar
x = 12.34                ; give x a value
x.view()                ; view x, title of dialog will be "Number"
y = LongInt(x)           ; cast x as a LongInt and assign to y
y.view()                 ; show y, note that decimal places are lost
                        ; displays 12 with "LongInt" as title of dialog
endMethod
```

Mail type

The Mail type allows you to compose electronic mail messages and transmit them using a MAPI-compliant mail system (e.g., Microsoft Mail). A Mail type variable holds a single mail message. It also holds current mail session status (set by **logon**), so that multiple mail messages can be sent (sequentially) in a single session. Declare variables of type Mail to facilitate the manipulation of mail messages, and then use the Mail methods to set (and retrieve) information about the message (such as the message subject and the recipients).

Methods for the Mail type

addAddress	getAddressCount	logon
addAttachment	getAttachment	logonDlg
addressBook	getAttachmentCount	readMessage
addressBookTo	getMessage	send
empty	getMessageType	sendDlg
emptyAddresses	getSender	setMessage
emptyAttachments	getSubject	setMessageType
enumInbox	logoff	setSubject
getAddress	logoffDlg	

addAddress method

Mail

Adds an addressee to a message.

addAttachment method

Syntax

1. `addAddress (const address String)`
2. `addAddress (const address String, const addressType SmallInt)`

Description

addAddress adds an addressee to the message. Syntax 1 defaults to a To type addressee, Syntax 2 allows you to specify one of the MailAddressTypes Constants: MailAddrTo, MailAddrCC, or MailAddrBC. Addressees are not checked for validity until the message is sent.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It assumes the user is logged on.

```
var
  m MAIL
endVar
method pushButton ( var eventInfo Event )
  m.addAddress("JDOE")
  m.addAddress("SSMITH", MailAddrCC)
  m.setSubject("Final sales numbers")
  m.setMessage("The final sales numbers are attached")
  m.addAttachment("SALES.TXT")
  m.send() ; Send the message
endMethod
```

addAttachment method

Mail

Adds an attachment to the message.

Syntax

1. `addAttachment (const fileName String)`
2. `addAttachment (const fileName String, const moniker String)`
3. `addAttachment (const fileName String, const moniker String, const displayPos LongInt)`

Description

addAttachment adds an attachment to the message. Syntax 1 sends the specified *fileName*. Syntax 2 sends the specified *fileName*, but displays the name specified in *moniker*. Some mail systems (for example, Microsoft Mail) allow the attachment icon to be displayed in the message text; in this case, you can use Syntax 3 to specify the position in the text that the file should appear. (With Microsoft mail, if you specify one, the first character of the message to be displaced by the icon for the specified attachment).

Some mail systems place limits on the number, size, and/or type of attachments you can use (a few mail systems still don't support binary attachments). No attempt is made to verify the existence of the files until the message is sent. Aliases can be used to specify attachment names.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It assumes the user is logged on.

```
var
  m MAIL
endVar
method pushButton ( var eventInfo Event )
  m.addAddress("JDOE")
  m.addAddress("SSMITH", MailAddrCC)
```

addressBook method

```
m.setSubject("Final sales numbers")
m.setMessage("The final sales numbers are attached")
m.addAttachment("SALES.TXT")

m.send() ; Send the message
endMethod
```

addressBook method

Mail

Displays the address book.

Syntax

1. addressBook ()
2. addressBook (const *numberOfLists* SmallInt)

Description

addressBook displays the address book and allows the user to modify the list of addressees. Syntax 1 allows all types of addressees (To, CC, BC) to be updated. Syntax 2 allows you to limit the number of address lists to be updated: *numberOfLists* = 1 shows only the To addressees, *numberOfLists* = 2 shows the To and CC addressees, *numberOfLists* = 3 shows the To, CC, and BC addressees.

If an existing mail session is not active, the user may be prompted with a logon dialog box. Use the **logon** method to create a mail session.

Example

The following example updates a distribution list kept in a table:

```
method pushButton ( var eventInfo Event )
var m MAIL tc TCURSOR idx LONGINT address STRING addrtype SMALLINT endVar
tc.open("distribution list.db")
scan tc: ; read the address list
m.addAddress( tc."Addressee" )
endscan
m.addressBook( 1 ) ; Display the list for editing
tc.edit( )
tc.empty( ) ; clear the old list
for idx from 1 to m.getAddressCount( ) ; write out the new list
tc.insertRecord( )
m.getAddress( idx, address, addrtype )
tc."Addressee" = address
tc.unlockRecord( )
endfor
tc.close( )
endMethod
```

addressBookTo method

Mail

Displays the To list from the address book.

Syntax

addressBookTo (const *prompt* String)

Description

addressBookTo displays the To list from the address book and allows the user to modify the list of addressees. **addressBookTo** displays only the To list, but allows you to override what the list is called (e.g., Routing).

empty method

If an existing mail session is not active, the user may be prompted with a logon dialog box. Use the **logon** method to create a mail session.

Example

The following example allows the user to update a distribution list kept in a table:

```
method pushButton ( var eventInfo Event )
    var m MAIL tc TCURSOR idx LONGINT address STRING addrtype SMALLINT endVar
    tc.open("distribution list.db")
    scan tc: ; read the address list
        m.addAddress( tc."Addressee" )
    endscan
    m.addressBookTo( "Fundraiser Mail List" )
    tc.edit( )
    tc.empty( ) ; clear the old list
    for idx from 1 to m.getAddressCount( ) ; write out the new list
        tc.insertRecord( )
        m.getAddress( idx, address, addrtype )
        tc."Addressee" = address
        tc.unlockRecord( )
    endfor
    tc.close( )
endMethod
```

empty method

Mail

Empties the contents of the mail variable.

Syntax

```
empty ( )
```

Description

empty empties the contents of the mail variable (clears the message). The session (which is set by the **logon** method), if any, is unaffected.

Example

The following example sends a message (about sales results) to John Doe, copies the message to Susan Smith and then sends a different message to Bill Brown. It assumes the user is logged on.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.send() ; Send the message

    m.empty() ; Clear out the old message

    m.addAddress("BBROWN")
    m.setSubject("Final sales numbers sent")
    m.setMessage("Bill, John and Susan have the final sales now")
    m.send() ; Send the message
endMethod
```

emptyAddresses method

Mail

Deletes all the addresses attached to a message.

Syntax

```
emptyAddresses ( )
```

Description

emptyAddresses sets the number of addresses attached to the message to zero.

Example

The following example sends a message (about sales results) to John Doe, copies the message to Susan Smith, and sends a different message to Bill Brown. It assumes the user is logged on.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.send() ; Send the message

    m.emptyAddresses() ; Clear out the old Addresses

    m.addAddress("BBROWN")
    m.setMessage("Bill, John and Susan have the final sales now")
    m.send() ; Send with subject & attachment specified earlier
endMethod
```

emptyAttachments method

Mail

Deletes all the attachments to a message.

Syntax

```
emptyAttachments ( )
```

Description

emptyAttachments sets the number of attachments to the message to zero.

Example

The following example sends a message (about sales results) to John Doe, copies the message to Susan Smith, and sends a different message to Bill Brown. It assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
var
    m MAIL
endVar
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.send() ; Send the message

    m.emptyAddresses() ; Clear out the old Addressee's
    m.emptyAttachments() ; Clear out the old Attachment
```

enumInbox method

```
m.addAddress("BBROWN")
m.setMessage("Bill, John and Susan have the final sales now")
m.send() ; Send with subject specified earlier
endMethod
```

enumInbox method

Mail

Fills an array with the list of messages in the in box.

Syntax

```
enumInbox ( var ids Array [] String, const unreadOnly Logical, [ const seedId String,
const maxCount LongInt, [ const msgType String ] ] )
```

Description

enumInbox fills the array specified by *ids* with the IDs of messages in the in box. *unreadOnly* is a True or False value that indicates whether to include only unread messages.

The optional value *seedId* lets you control the starting point from which messages are listed in the array. To retrieve the first set of messages, use a blank string (""). To retrieve subsequent sets of messages, use the last ID read from the previous set. If you specify a value for *seedId*, you must also specify a value for *maxCount*. *maxCount* lets you control the maximum number of messages retrieved in a set.

msgType lets you specify the type of message. Message types are mail system dependent - consult your mail system documentation for information on the message types it supports.

Example

The following example gets the list of message IDs for unread messages in the in box. The example reads the first set of messages and stops after listing a maximum of 100 messages. A custom method `ProcessMessage` is called to process each message until there are no messages left in the set. If additional messages remain unread and need processing, the loop repeats. When there are no more messages to process, the method ends. This example assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
var
    msg           Mail
    inboxIds      Array [] String
    seedId        String
    i             LongInt

endVar
seedId = "" ; set to retrieve first message
while True ; Process all unread messages

    msg.enumInbox( inboxIds, True, seedId, 100)
    for i from 1 to inboxIds.size()

        processMessage(inboxIds[i]) ; run a custom method to process
                                    ; each message
    endFor
    if inboxIds.size() 100 then
        quitloop
    endIf
    seedId = inboxIds[100] ; set seed to last message read
endWhile
endMethod
```

getAddress method**Mail**

Retrieves the specified addressee information.

Syntax

1. getAddress (const *index* LongInt, var *address* String, var *addressType* SmallInt)
2. getAddress (const *index* LongInt, var *address* String, var *fullAddress* String, var *addressType* SmallInt)

Description

getAddress retrieves the specified addressee information, where *index* is between 1 and **getAddressCount**, inclusive. *addressType* is one of the MailAddressTypes Constants: MailAddrTo, MailAddrCC, or MailAddrBC.

In addition to the above information, Syntax 2 retrieves the full address name of the addressee and stores this value in *fullAddress*. Full address information is available only after a MAPI-compliant mail system has made this information available to Paradox. A blank value for *fullAddress* indicates that the MAPI-compliant mail system has not yet provided this information.

Example

The following example allows the user to update a distribution list kept in a table:

```
method pushButton ( var eventInfo Event )
  var m MAIL tc TCURSOR idx LONGINT address STRING addrtype SMALLINT endVar
  tc.open("distribution list.db")
  scan tc: ; read the address list
    m.addAddress( tc."Addressee" )
  endscan
  m.addressBookTo( "Fundraiser Mail List" )
  tc.edit( )
  tc.empty( ) ; clear the old list
  for idx from 1 to m.getAddressCount( ) ; write out the new list
    tc.insertRecord( )
    m.getAddress( idx, address, addrtype )
    tc."Addressee" = address
    tc.unlockRecord( )
  endfor
  tc.close( )
endMethod
```

getAddressCount method**Mail**

Returns the number of addressees attached to the current message.

Syntax

getAddressCount () LongInt

Description

getAddressCount returns the number of addressees attached to the current message.

Example

The following example allows the user to update a distribution list kept in a table:

```
method pushButton ( var eventInfo Event )
  var m MAIL tc TCURSOR idx LONGINT address STRING addrtype SMALLINT endVar
  tc.open("distribution list.db")
  scan tc: ; read the address list
    m.addAddress( tc."Addressee" )
  endscan
```


getAttachment method

```
m.addressBookTo( "Fundraiser Mail List" )
tc.edit( )
tc.empty( ) ; clear the old list

for idx from 1 to m.getAddressCount( ) ; write out the new list
    tc.insertRecord( )
    m.getAddress( idx, address, addrtype )
    tc."Addressee" = address
    tc.unlockRecord( )
endfor
tc.close( )
endMethod
```

getAttachment method

Mail

Retrieves specific attachment information.

Syntax

```
getAttachment ( const index LongInt, var fileName String, var moniker String, var
displayPos LongInt )
```

Description

getAttachment retrieves the attachment information for the attachment specified by *index*. *index* is a number between 1 and **getAttachmentCount**, inclusive. *fileName*, *moniker*, and *displayPos* are variables whose values are filled in by this method. *fileName* represents the name of the attachment file. *moniker* is the name displayed in the MAPI mail dialog (defaults to the filename). *displayPos* is the display position of the attachment's icon in the MAPI mail dialog.

Example

The following example gets the list of attachments from a mail variable. The mail variable *m* and its attachments are presumed to have been defined and added elsewhere. The example assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
    var
        list DYNARRAY [ ] STRING
        indx LONGINT
        filename STRING
        moniker STRING

        pos LONGINT
    endVar
    for indx from 1 to m.getAttachmentCount()
        m.getAttachment(indx, filename, moniker, pos)
        list[indx]=filename
    endfor

    list.view("attachments:")
endMethod
```

getAttachmentCount method

Mail

Returns the number of attachments to the current message.

Syntax

```
getAttachmentCount ( ) LongInt
```

getMessage method

Description

getAttachmentCount returns the number of attachments to the current message.

Example

The following example displays the number of attachments. The mail variable *m* and its attachments are presumed to have been defined and added elsewhere. The message assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
    var

        cnt longint

    endVar
    m.addAttachment( "SALES.TXT" )
    cnt = m.getAttachmentCount( )
    cnt.view( "Number of attachments" )
endMethod
```

getMessage method

Mail

Returns the current text of the message.

Syntax

```
getMessage ( ) String
```

Description

getMessage returns the current text of the message.

Example

The following example displays the (previously set) message text. It assumes the user is logged on.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    var msgtext string endVar
    msgtext = m.getMessage( )
    msgtext.view( "Message text" )
endMethod
```

getMessageType method

Mail

Returns the current message type.

Syntax

```
getMessageType ( ) String
```

Description

getMessageType returns the current message type. Message types are mail system dependent - consult your mail system documentation for information on the message types it supports.

Example

The following example displays the (previously set) message type. It assumes the user is logged on.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
```

getSender method

```
var msgtype string endVar
msgtype = m.getMessageType( )
msgtype.view( "Message type" )
endMethod
```

getSender method

Mail

Returns the sender for the current message.

Syntax

1. `getSender (var address String)`
2. `getSender (var address String, var fullAddress String)`

Description

getSender returns the sender of the current mail message as *address*. In Syntax 2, **getSender** returns the full address of the sender as *fullAddress*. (Many mail systems differentiate between nickname addresses and the full email address. If the mail system that you use does not differentiate, *address* and *fullAddress* will return the same value.)

The sender's address (and full address) is available only when messages are read. The values will be blank for messages you are composing.

Example

The following example displays the sender information for the mail message `msg`. `getSender` and `getSubject` are called to get the subject title and sender name to create the reply mail. The message assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
var
    msg,
    replyMsg      Mail
    sender,
    fullAddress    String
endVar

msg.readMessage("1234")                ; 1234 is a valid message ID
msg.getSender(sender,fullAddress)       ; supplies sender info

replyMsg.addAddress(fullAddress)        ; add sender to reply
replyMsg.setSubject(msg.getSubject()+" - Reply") ; set reply message
                                              ; subject to original
                                              ; subject plus "reply"

replyMsg.sendDlg()                     ; open the send dialog to
                                              ; let user type contents
                                              ; for the reply message

endMethod
```

getSubject method

Mail

Returns the current subject of the message.

Syntax

```
getSubject ( ) String
```

Description

getSubject returns the current subject of the message.

logoff method

Example

The following example displays the (previously set) subject for the mail variable m (assigned elsewhere). It assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
    var
        subject string

    endVar
    subject = m.getsubject( )
    subject.view( "Subject" )
endMethod
```

logoff method

Mail

Attempts to logoff the mail system.

Syntax

```
logoff ( )
```

Description

logoff attempts to logoff the mail system without user intervention and to terminate the mail session created by **logon**. Any errors will trigger an exception.

Example

The following example logs on, displays the send dialog box and then logs off:

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    m.logon("mypassword", "special" )
    m.sendDlg( )
    m.logoff( )
endMethod
```

logoffDlg method

Mail

Attempts to logoff the mail system with user interaction.

Syntax

```
logoffDlg ( ) Logical
```

Description

logoffDlg attempts to logoff the mail system and to terminate the mail session created by **logon**. If supported by the mail system, the user is prompted to enter logoff information, otherwise a straight logoff is done.

logoffDlg returns True if the user logs off, and False if the user cancels. Any errors will trigger an exception.

Example

The following example logs on, displays the send dialog box, logs off and displays a logoff dialog box if appropriate:

```
var
    m MAIL
```

logon method

```
endVar
method pushButton ( var eventInfo Event )
    m.logon("mypassword", "special" )
    m.sendDlg( )
    m.logoffDlg( )
endMethod
```

logon method

Mail

Attempts to logon to the mail system.

Syntax

```
logon ( const password String, const profileName String )
```

Description

logon attempts to log on to the mail system without user intervention. Any errors will trigger an exception.

The password argument is an input parameter that specifies a credential string (maximum 256 characters). If the messaging system does not require password credentials, or if it requires that the user actively enter them, password should be blank. When the user must enter credentials, use **logonDlg**.

The argument profileName is an input parameter that specifies a named profile string (maximum of 256 characters). This is the profile to use when logging on. Some mail providers accept a null profileName as specifying the default profile. If you don't know the profileName, use **logonDlg**.

Example

The following example sends a message (about sales results) to John Doe, copies the message to Susan Smith, and sends a different message to Bill Brown. It uses logon to specify a special mail session and to group everything together.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    m.logon("mypassword", "special" )
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.send() ; Send the message

    m.empty() ; Clear out the old message

    m.addAddress("BBROWN")
    m.setSubject("Final sales numbers sent")
    m.setMessage("Bill, John and Susan have the final sales now")
    m.send() ; Send the message
    m.logoff()
endMethod
```

logonDlg method

Mail

Attempts to logon to the mail system with user interaction.

readMessage method

Syntax

1. `logonDlg ()` Logical
2. `logonDlg (const password String, const profile String)` Logical

Description

logonDlg attempts to logon to the mail system with user interaction. If necessary, the user is prompted to enter logon information. If successful, a mail session is created. The session stays active until the **logoff** method is called, or the mail variable goes out of scope.

logonDlg returns True if the user logs on, and False if the user cancels. Any errors will trigger an exception.

Example

The following example sends a message (about sales results) to John Doe, copies the message to Susan Smith, and sends a different message to Bill Brown. It uses `logonDlg` so that the user will only have to specify a mail password once.

```
var
    m MAIL
endVar
method pushButton ( var eventInfo Event )
    m.logonDlg( )
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.send() ; Send the message

    m.empty() ; Clear out the old message

    m.addAddress("BBROWN")
    m.setSubject("Final sales numbers sent")
    m.setMessage("Bill, John and Susan have the final sales now")
    m.send() ; Send the message
    m.logoff()
endMethod
```

readMessage method

Mail

Reads a mail message.

Syntax

`readMessage (var messageId AnyType, [const readOpts Anytype])`

Description

readMessage reads a mail message into a Mail variable. Use `MailReadOptions` constants to specify reading options (multiple `MailReadOptions` constants may be used at the same time by adding them together.)

Example

In the following example, the in box of the MAPI-compliant mail system is enumerated to the array *InboxIds*. Each message in *InboxIds* is read and calls a custom method that contains a message processing routine. The loop repeats as many times as there are messages to process. The example assumes the user is logged on.

send method

```
method pushButton ( var eventInfo Event )

var
    msg          Mail
    inboxIds     Array [] String
    i            LongInt
endVar

msg.enumInbox( inboxIds, True)
for i from 1 to inboxIds.size()

    msg.readMessage(inboxIds[i])
    doProcess()           ; calls a custom method for processing
endFor

endMethod
```

send method

Mail

Sends a mail message.

Syntax

```
send ( )
```

Description

send sends a mail message without user interaction. At least one addressee must have been defined. Most mail systems require that some additional information is defined (for example, the subject).

If an existing mail session is not active, the user may be prompted with a logon dialog box. Use the **logon** method to create a mail session. Some mail provider systems may require an explicit logon call before a send, others may not.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
var
    m MAIL
endVar
m.addAddress("JDOE")
m.addAddress("SSMITH", MailAddrCC)
m.setSubject("Final sales numbers")
m.setMessage("The final sales numbers are attached")
m.addAttachment("SALES.TXT")
m.send() ; Send the message
endMethod
```

sendDlg method

Mail

Sends a mail message with user interaction.

Syntax

```
sendDlg ( ) Logical
```

setMessage method

Description

sendDlg sends a mail message with user interaction. The user will be shown the message as it currently exists (using the user's default MAPI mail system provider). They can then modify it before sending it.

If an existing mail session is not active, the user may be prompted with a **logon** dialog box. Use the **logon** method to create a mail session.

sendDlg returns True if the user sends the message, and False if they cancel. Any errors will trigger an exception.

sendDlg returns True if the user cancels the logon dialog box.

Example

The following example simply displays a mail dialog box for the user to enter a message. It assumes the user is logged on.

```
method pushButton ( var eventInfo Event )
var
    m    MAIL
endVar
    m.sendDlg()
endMethod
```

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith:

```
method pushButton ( var eventInfo Event )
var
    m    MAIL
endVar
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.sendDlg() ; Display the message so the user can edit before sending
endMethod
```

setMessage method

Mail

Sets the text of the message.

Syntax

```
setMessage ( const message String )
```

Description

setMessage sets the text of the message to message. The maximum length of message is limited by the shorter of the mail system and ObjectPAL's maximum string length. This is typically at least 32,000 characters.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It assumes the user is logged on.

```
var
    m    MAIL
endVar
method pushButton ( var eventInfo Event )
```


setMessageType method

```
m.addAddress("JDOE")
m.addAddress("SSMITH", MailAddrCC)
m.setSubject("Final sales numbers")
m.setMessage("The final sales numbers are attached")
m.addAttachment("SALES.TXT")
m.sendDlg() ; Display the message so the user can edit before sending
endMethod
```

setMessageType method

Mail

Sets the type of the message.

Syntax

```
setMessageType ( const messageType String )
```

Description

setMessageType sets the type of the *message*. Some mail systems support a **messageType**. Typically, message without a specified type is assumed to be an Inter-Personal Message; whereas, typed messages can only be read by a program asking for that particular message type.

Using message types typically requires special support from your mail system. Consult your mail vendor for more information.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It uses a special message type IPM.URGENT that was previously set up on this mail system. It assumes the user is logged on.

```
var
  m MAIL
endVar
method pushButton ( var eventInfo Event )
  m.addAddress("JDOE")
  m.addAddress("SSMITH", MailAddrCC)
  m.setSubject("Final sales numbers")
  m.setMessage("The final sales numbers are attached")
  m.addAttachment("SALES.TXT")
  m.setMessageType("IPM.URGENT")
  m.sendDlg() ; Display the message so the user can edit before sending
endMethod
```

setSubject method

Mail

Sets the subject of the message.

Syntax

```
setSubject ( const subject String )
```

Description

setSubject sets the subject of the message to *subject*. The maximum length of *subject* is limited by the mail system. This is typically at least 80 characters.

Example

The following example sends a message (about sales results) to John Doe and copies the message to Susan Smith. It assumes the user is logged on.

```
var
  m MAIL
```

memo procedure

```
endVar
method pushButton ( var eventInfo Event )
    m.addAddress("JDOE")
    m.addAddress("SSMITH", MailAddrCC)
    m.setSubject("Final sales numbers")
    m.setMessage("The final sales numbers are attached")
    m.addAttachment("SALES.TXT")
    m.sendDlg() ; Display the message so the user can edit before sending
endMethod
```

Memo type

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 (and reports), tables, and disk files.

You can also use the (=) operator to assign the value of a memo field to a Memo variable or a String variable. Note that 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 includes several derived methods from the AnyType type.

Methods for the Memo type

AnyType	←	Memo
blank		memo
dataType		readFromClipboard
isAssigned		readFromFile
isBlank		readFromRTFFile
isFixedType		writeToClipboard
		writeToFile
		writeToRTFFile

memo procedure

Memo

Casts a value as a Memo.

Syntax

```
memo ( const value AnyType [ , const value AnyType ] * ) Memo
```

Description

memo casts the expression *value* to a Memo. If you specify multiple arguments, this method will cast all of them to Memos and concatenate them to one Memo.

Example

The following 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 particular record in *DocFiles*. The code then casts and concatenates the contents of the three *DocFiles* fields to a String value and 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. When the value is cast as a String,

readFromClipboard method

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
```

readFromClipboard method

Memo

Reads text from the Clipboard.

Syntax

```
readFromClipboard ( ) Logical
```

Description

readFromClipboard reads text from the Clipboard. **readFromClipboard** will attempt to read in Rich Text Format if the format is available in the Clipboard. Otherwise, text (CF_TEXT) will be read in. This method returns True if successful and False if unsuccessful.

Example

In the following example, a form has two buttons: *readFromClipboard* and *writeToClipboard*. The first button will read RTF formatted text from the Clipboard into a Memo variable that will then be stored in a table. The second button reads a memo value from a table and writes it to the Clipboard.

The following code is attached to the **pushButton** method for *btnReadFromClipboard*:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
    vrMemo Memo
    tcMemo TCursor
```

readFromFile method

```
endVar

; Open table to hold memos
tcMemo.open("mymemos.db")
tcMemo.edit()
if vrMemo.readFromClipboard() then
    ; Add a record to the table and insert the value
    tcMemo.insertRecord()
    tcMemo.MemoField = vrMemo
    tcMemo.unlockRecord()
endIf
tcMemo.close()

endMethod
```

The following code is attached to the **pushButton** method for *btnWriteToClipboard*:

```
; btnWriteToClipboard::pushButton
method pushButton(var eventInfo Event)
var
    vrMemo Memo
    tcMemo TCursor
endVar

; Open table to which contains memos
tcMemo.open("mymemos.db")
; Make sure there is data in the table
if tcMemo.nRecords() <> 0 then
    ; Copy a value to the Memo variable
    vrMemo = tcMemo.MemoField
    ; Write it out to the Clipboard
    vrMemo.writeToClipboard()
endIf
tcMemo.close()

endMethod
```

readFromFile method

Memo

Reads a memo from a file.

Syntax

```
readFromFile ( const fileName String ) Logical
```

Description

readFromFile 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 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 folder 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 folder,
```

writeToClipboard method

```
; and write it to the bitmapField graphic
if chessBmp.readFromFile("c:\\windows\\chess.bmp") then
  bitmapField = chessBmp
endif
endMethod
```

writeToClipboard method

Memo

Writes a memo to the Clipboard.

Syntax

```
writeToClipboard ( ) Logical
```

Description

writeToClipboard writes a memo to the Clipboard. The formats copied to the Clipboard are text (CF_TEXT) and Rich Text Format. writeToClipboard returns True if successful and False if unsuccessful.

Example

The following example reads the contents of a text file to a memo field in a table. This examples assumes 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, 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 PJNotes.DB 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
```

writeToFile method

Memo

Writes a memo to a file.

Syntax

```
writeToFile ( const fileName String ) Logical
```

Description

writeToFile writes a memo to a disk file specified in *fileName*. This method writes text only. It does not write the formatting of formatted memos.

writeToRTFFile method

Example

The following example writes the contents of a memo to a text file. This example assumes 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, and 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 PJNotes.DB 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
```

writeToRTFFile method

Memo

Writes a memo to an RTF file.

Syntax

```
writeToRTFFile ( const fileName String ) Logical
```

Description

writeToRTFFile writes a memo to an RTF disk file specified in *fileName*. This method writes text including the formatting of formatted memos.

Example

See the **writeToFile** example.

readFromRTFFile method

Memo

Reads a memo from an RTF file.

Syntax

```
readFromRTFFile ( const fileName String ) Logical
```

Description

readFromRTFFile reads a memo from a disk file specified in *fileName*. This method reads text including the formatting of formatted memos.

Example

See the **readFromFile** example.

addArray method

Menu

The Menu type includes several derived methods from the AnyType type.

Methods for the Menu type

AnyType	←	Menu
dataType		addArray
isAssigned		addBreak
isFixedType		addPopUp
unAssign		addStaticText
		addText
		contains
		count
		empty
		getMenuChoiceAttribute
		getMenuChoiceAttributeByld
		hasMenuChoiceAttribute
		menusetlimit
		remove
		removeMenu
		setMenuChoiceAttribute
		setMenuChoiceAttributeByld
		show

addArray method

Menu

Appends elements of an array to a menu.

Syntax

addArray (const *items* Array[] String)

Description

addArray 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

The following 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
```

addBreak method

```
    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
```

addBreak method

Menu

Starts a new row in a menu.

Syntax

```
addBreak ( )
```

Description

addBreak starts a new row in a menu. **addBreak** lets you explicitly wrap large menu constructs to two or more rows.

Example

The following 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
```

addPopUp method

Menu

Adds a pop-up menu to a Menu Bar item.

Syntax

```
addPopUp ( const menuName String, const cascadedPopup PopUpMenu )
```

Description

addPopUp 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.

addStaticText method

Note

- If you use **addPopUp** with a *menuName* of &Window, Windows automatically appends a list of open windows to that pop-up menu.

Example

In the following example the code 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.

Note

- Use SHIFT + F4 to move from the first page to the second. Use SHIFT + F3 to move from the second page to the first.

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
```

addStaticText method

Menu

Adds an unselectable text string to a menu.

Syntax

```
addStaticText ( const item String )
```

addText method

Description

addStaticText appends *item* to a menu as unselectable text.

Example

In the following 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
```

addText method

Menu

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

addText adds a selectable text string to a menu.

Syntax 1 adds the item *menuName* to a menu. Menu items are displayed from left to right across the Menu Bar.

In Syntax 2, you can use *attrib* to preset the display attribute of *menuName*. Use MenuChoiceAttributes constants to specify attributes.

In Syntax 3, you can specify an *id* number (a SmallInt) to identify the menu by number instead of by *menuName*. In the built-in event **menuAction** method, you can use the *id* number to determine which menu the user chooses. When you specify a menu *id*, you should use the built-in IdRanges constant **UserMenu** as a base constant and then add your own number to it or create a user-defined menu constant. In the following example, the code adds File to the *myMenu* menu and specifies an *id* number for that menu item:

```
myMenu.addText("File", MenuEnabled, UserMenu + 1)
```

You can use an ampersand in an item to designate an accelerator key. For example, the item &File would display 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 the following example, the return value is &File, not File.

addText method

Example 1

Examples 1 and 2 demonstrate how **addText** syntax influences the way you test for the user's menu choice.

The following example uses the first form of **addText** syntax to create a simple menu. It does not use *id* in the **addText** statements. The code attached to the built-in event **menuAction** method must evaluate the string specified in *menuName* to determine the user's menu choice. The following code 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
```

Example 2

The following 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*:

addText method

```
; 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 (i.e.: 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 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 (i.e.: 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
```

contains method

Menu

Reports whether an item is in a menu.

Syntax

```
contains ( const item AnyType ) Logical
```

Description

contains returns True if *item* is in the list of items in a menu; otherwise, it returns False.

Example

The following 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. This example uses **contains** to determine if Undo is present before it adds or removes 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 :
      if not m1.contains("&Undo") then
        ; add Undo and redisplay the menu
        m1.addText("&Undo")
        m1.show()
      endif

    ; when user posts the record (moves to another record)
    case eventInfo.id() = DataUnlockRecord :
      if m1.contains("&Undo") then
        ; remove Undo redisplay the menu
        m1.remove("&Undo")
        m1.show()
      endif
```

count method

```
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 active record  
        case choice = "&Undo" :  
            active.action(DataCancelRecord) ; restore original state  
            m1.remove("&Undo") ; remove Undo menu item  
            m1.show() ; redisplay menu without Undo  
    endswitch  
  
endif  
endMethod
```

count method

Menu

Returns the number of items in a menu.

Syntax

```
count ( ) SmallInt
```

Description

count 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 and then displays the number of items in each menu. **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
```

empty method

```
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
```

empty method

Menu

Removes all items from a menu.

Syntax

```
empty ( )
```

Description

empty removes all items from a custom menu. Use **empty** when you need to clear an existing menu before you rebuild it.

Example

The following example uses two buttons to display alternate menus. Both methods affect the same menu, which is 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
```

The following code is 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
```

The following code is 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
```

getMenuChoiceAttribute procedure

Menu

Reports the display attributes of a menu item.

Syntax

```
getMenuChoiceAttribute ( const menuChoice String ) SmallInt
```

getMenuChoiceAttributeById procedure

Description

getMenuChoiceAttribute returns an integer that represents the display attributes of the menu item specified in *menuChoice*. The integer value represents the combination of attributes that apply. Use **MenuChoiceAttributes** constants to test attributes. Use **getMenuChoiceAttribute** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item. 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 the following 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
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 disabled
msgInfo("Time enabled?", HasMenuChoiceAttribute(attrib, MenuEnabled))
; this displays True because Time is grayed
msgInfo("Time grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))

endMethod
```

getMenuChoiceAttributeById procedure

Menu

Reports the display attribute of a menu item specified by its menu ID.

Syntax

```
getMenuChoiceAttributeById ( const menuId SmallInt ) SmallInt
```

Description

getMenuChoiceAttributeById returns an integer that represents the display attributes of the menu item specified in *menuId*. The integer value represents the combination of attributes that apply. Use

getMenuChoiceAttributeById procedure

MenuChoiceAttributes constants to test attributes. Use **getMenuChoiceAttributeById** with **hasMenuChoiceAttribute** to determine whether a specific display attribute applies for a menu item.

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
```

hasMenuChoiceAttribute procedure

```
msgInfo("Undo enabled?", hasMenuChoiceAttribute(attrib, MenuEnabled))
; this displays True because Undo is grayed
msgInfo("Undo grayed?", hasMenuChoiceAttribute(attrib, MenuGrayed))
endMethod
```

hasMenuChoiceAttribute procedure

Menu

Reports whether a menu item contains a given display attribute.

Syntax

```
hasMenuChoiceAttribute ( const attrib SmallInt , const attribSet SmallInt ) Logical
```

Description

hasMenuChoiceAttribute returns True if *attribSet* contains the attribute specified in *attrib*; otherwise, it returns False. Use MenuChoiceAttributes constants to specify attributes.

Use **hasMenuChoiceAttribute** with **getMenuChoiceAttribute** or **getMenuChoiceAttributeById** to determine whether a particular display attribute for a menu item is represented in *attribSet*.

Example

The following example 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
```

menuSetLimit procedure

Menu

Used to set the number of items in a popupmenu.

Syntax

```
menuSetLimit ( const limit SmallInt )
```

Description

menuSetLimit is used to set the limit of a popupmenu to determine the number of items the menu can hold. If you don't set the limit, the maximum number of items allowed in a popupmenu is 32.

Example

The following example will create a popupmenu with 100 items.

```
method mouseRightDown(var eventInfo MouseEvent)
  var
    si1Counter    SmallInt
    si2Counter    SmallInt
    popChose      PopUpMenu
  endVar
  si2Counter = 0
  menuSetLimit(200)
  for si1Counter from 1 to 100
    popChose.addText(si1Counter)
    si2Counter = si2Counter + 1
    if si2Counter = 20 then
      popChose.addBar()
      si2Counter = 0
    endif
  endFor
  fldChose = popChose.show()
endmethod
```

isAssigned method

Menu

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just Menu.

Example

The following 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 is attached 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)
```

remove method

```
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
```

remove method

Menu

Removes an item from a menu.

Syntax

```
remove ( const item AnyType )
```

Description

remove deletes the first occurrence of *item* from a menu. This method is used to change one item in a menu without having to rebuild the entire menu.

Example

The following 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()

msgInfo("Status", "About to remove the menus. Watch closely.")

; remove entire menu, reveal built-in menus
removeMenu()
endMethod
```

removeMenu procedure

Menu

Removes a custom menu and displays the default menu.

Syntax

```
removeMenu ( )
```

setMenuChoiceAttribute procedure

Description

removeMenu replaces a menu built using ObjectPAL with the Paradox 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
```

setMenuChoiceAttribute procedure

Menu

Sets the display attribute of a menu item.

Syntax

```
setMenuChoiceAttribute ( const menuChoice String, const menuAttribute SmallInt )
```

Description

setMenuChoiceAttribute sets the display attribute of *menuChoice* to *menuAttribute*. Use MenuChoiceAttributes constants to specify attributes. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttribute** affects the built-in menu.

setMenuChoiceAttribute procedure

Note

- If a menu item's definition includes an accelerator key (for example, Print which is defined as &Print), remember to include the ampersand in the comparison string *menuChoice*.

Example

The following example changes 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 can be selected. 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
```

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
```

setMenuChoiceAttributeById procedure

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 active 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
```

setMenuChoiceAttributeById procedure

Menu

Sets the display attribute of a menu item.

Syntax

```
setMenuChoiceAttributeById ( const menuId String, const menuAttribute SmallInt )
```

Description

setMenuChoiceAttributeById sets the display attribute of *menuId* to *menuAttribute*. Use **MenuChoiceAttributes** constants to specify attributes. This procedure affects the currently displayed menu; if you have not created a custom menu, **setMenuChoiceAttributeById** affects the built-in menu.

Note

- If a menu item's definition includes an accelerator key (e.g., Print which is defined as &Print), remember to include the ampersand in the comparison string *menuChoice*.

Example

The following example changes 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 can be selected. 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 by menu name.

The following code goes in the form's Var window:

```
; thisForm::var
Var
```

setMenuChoiceAttributeById procedure

```
m1 Menu
p1 PopUpMenu
endVar
```

The following code goes in the form's Const Window:

```
; thisForm::const
Const
    InsMenu = 1 ; use constants for menu id's
    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 + UserMenu,
                                        MenuEnabled)

            ; when user posts the record (moves to another record)
        case eventInfo.id() = DataUnlockRecord :
            ; disable and dim Undo menu item
            setMenuChoiceAttributeById(UndoMenu + UserMenu,
                                        MenuGrayed + MenuDisabled)
    endswitch

else
    ;code here executes just for form itself
endif

endMethod
```


show method

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 active 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
```

show method

Menu

Displays a menu.

Syntax

```
show ( )
```

Description

show displays a menu.

The user's choice is handled using the built-in event methods **menuAction** and **menuChoice** from the MenuEvent type.

Example

In the following example, a form's **open** method constructs a simple menu and displays it with **show**. The **menuAction** method for the form handles the user's menu choice. The following code is 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
```

show method

```
        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
```

MenuEvent type

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 built-in **menuAction** method. By modifying an object's built-in **menuAction** method, you can define how the object responds.

The MenuEvent type includes several derived methods from the Event type.

You can also define your own menu constants. For more information, see User-defined menu constants.

Methods for the MenuEvent type

Event	←	MenuEvent
errorCode		data
getTarget		id
isFirstTime		isFromUI
isPreFilter		menuChoice
isTargetSelf		reason
setErrorCode		setData
		setId
		setReason

User-defined constants

You can define your own menu constants, but you must keep them within a specific range. Because this range is subject to change in future versions of Paradox, ObjectPAL provides the `IdRanges` constants `UserMenu` and `UserMenuMax` to represent the minimum and maximum values allowed.

The following example supposes that you want to define two menu constants, `ThisMenuItem` and `ThatMenuItem`. You would define values for your custom constants in a `Const` window as follows:

```
Const
  ThisMenuItem = 1
  ThatMenuItem = 2
EndConst
```

To use one of these constants, you would add it to `UserMenu`. For example,

```
method menuAction(var eventInfo MenuEvent)
  if eventInfo.id() = UserMenu + ThisMenuItem then
    doSomething()
  endIf
endMethod
```

By adding `UserMenu` to your own constant, you guarantee yourself a value above the minimum. To keep the value under the maximum, use the value of `UserMenuMax`. One way to check the value is with the following **message** statement:

```
message(UserMenuMax)
```

In this version of Paradox, the difference between `UserMenu` and `UserMenuMax` is 2047. That means the largest value you can use for a menu constant is `UserMenu + 2047`.

data method

MenuEvent

Returns information about a `MenuEvent`.

Syntax

```
data ( ) LongInt
```

Description

data 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`. For more information, see your Windows programming documentation.

id method

MenuEvent

Returns the ID of a `MenuEvent`.

Syntax

```
id ( ) SmallInt
```

Description

id returns the ID number of a `MenuEvent`. ObjectPAL provides `MenuCommands` constants (like `MenuFileOpen`) for many common menu choices. You can also use user-defined menu constants to test the value returned by **id**.

id method

Example 1

The following example attaches code to a form's built-in **menuAction** method. When the user selects Close from the System 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)
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
```

Example 2

The following 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 (i.e.: 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)
```

isFromUI method

```
    ; 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 s.

```
    ; 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
```

Example 3

The following example shows you how to use the menu action event and to test for the ID **MenuCanClose**. This will display a message before the user can close the form. To stop the closure of the form, use **setErrorCode** and provide any non zero value.

```
method MenuAction (var eventInfo MenuEvent)

if eventInfo.isPrefilter() then

else
    if eventInfo.id() = MenuCanClose then
        if msgQuestion("Exit?", "Are you sure?") = "No" then
            eventInfo.setErrorCode(1)    ; Any non-zero error code works
        endif
    endif
endif

endMethod
```

isFromUI method

MenuEvent

Reports whether an event was generated by the user interacting with Paradox.

Syntax

```
isFromUI ( ) Logical
```

isFromUI method

Description

isFromUI reports whether an event was generated by the user interacting with Paradox, or internally (e.g., by an ObjectPAL statement). This method returns True if the event was generated by the user; otherwise, it returns False.

Example 1

The following example checks for a menu action to delete a record. If the action is from the UI (that is, if the user made the menu choice), a dialog box prompts for confirmation before the record is deleted.

```
;frm :: menuAction
method menuAction(var eventInfo MenuEvent)

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:

        if eventInfo.id() = MenuRecordDelete and
            eventInfo.isFromUI() then
            if msgQuestion("Delete record?",
                "Delete this record?") "Yes" then

                disableDefault
                return
            endif
        endif
    endif

endMethod
```

Example 2

The following example shows how you can use **isFromUI** to indicate if the menu action was sent by **menuAction** or by **sendKeys**.

The following code is attached to the page's Const window. It declares constants to make it easy to remember the menu ID assignments.

```
; pageOne::Const
Const
    ; define constants for menu IDs
    ; actual values (1, 2 and 3) are arbitrary
    kTimeMenu = 1
    kDateMenu = 2
    kHelpMenu = 3
endConst
```

The following code is attached to the open method for *pageOne*. To control the menu display attributes, this code uses ObjectPAL constants such as **MenuEnabled**. To identify each menu item by number, the code uses the constants defined in the Const window for *pageOne* (**kTimeMenu**, **kDateMenu**, and **kHelpMenu**).

```
; pageOne::open
method open(var eventInfo Event)
var
    mainMenu Menu
    utilPU    PopUpMenu
endVar

; build a pop-up menu and use constants (i.e.: kTimeMenu)
```

isFromUI method

```
; defined in the Const window for thisPage
utilPU.addText("&Time", MenuEnabled, kTimeMenu + UserMenu)
utilPU.addText("&Date", MenuEnabled, kDateMenu + 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, kHelpMenu + 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*. In addition, it uses **isFromUI** to report whether the menu event was generated by **menuAction** or by **keyPhysical**.

```
; pageOne::menuAction
method menuAction(var eventInfo MenuEvent)
var
    choice          SmallInt
    youDoneIt       Logical
endVar

youDoneIt = eventInfo.isFromUI()
choice = eventInfo.id()      ; assign constant value to choice

; now use constants to determine which menu was selected
switch
case choice = kTimeMenu + UserMenu:
    msgInfo("Did a user do this", youDoneIt)
    msgInfo("Current Time", time())
case choice = kDateMenu + UserMenu:
    msgInfo("Did a user do this", youDoneIt)
    msgInfo("Today's Date", today())
case choice = kHelpMenu + 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
```

The following two buttons demonstrate the use of the code above. The following code is attached to the **pushButton** method of a button named *btnObjectPAL*. It uses **menuAction** to send a menu event:

```
;btnObjectPAL :: pushButton
method pushButton(var eventInfo Event)
    menuAction(kDateMenu + UserMenu)
endMethod
```

The following code is attached to the **pushButton** method of a button named *btnSendKeys*. It uses **sendKeys** to send the keystrokes ALT + u + t. Use this button to simulate a user selecting a menu.

```
;btnSendKeys :: pushButton
method pushButton(var eventInfo Event)
    sendKeys("%ut")
endMethod
```

menuChoice method**Menu Event**

Returns a string that contains an item chosen from a menu.

Syntax

```
menuChoice ( ) String
```

Description

menuChoice returns a string that contains 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.

If the definition of a menu item includes an accelerator key (e.g., &Print), remember to include the ampersand in the comparison string. The following example compares the return value of **menuChoice** with the string &Print:

```
if eventInfo.menuChoice() = "&Print" then
    ; print the report
endif
```

Example

The following 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 to 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.

The following 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
```

The following code is attached to 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
```


reason

The following 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
```

reason

MenuEvent

Reports the type of menu chosen.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why a MenuEvent occurred. MenuEvent reasons occur when a built-in **menuAction** method is called. ObjectPAL provides MenuReasons constants to test the value returned by **reason**.

Example

In the following example, the form's **menuAction** method examines every MenuEvent to determine the reason for the MenuEvent. The reason 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
```

setData

MenuEvent

Specifies information about a MenuEvent.

Syntax

```
setData ( const menuData LongInt )
```

Description

setData 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. For more information, see your Windows programming documentation.

setId

setId

MenuEvent

Specifies the ID of a MenuEvent.

Syntax

```
setId ( const commandId SmallInt )
```

Description

setId specifies in *commandId* an action to take as the result of a menu choice, where *commandId* is a MenuCommands constant.

If you change the ID for a MenuEvent with **setId**, you may also need to change the reason for that MenuEvent with **setReason**.

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 **id** example.

setReason

MenuEvent

Specifies a reason for generating a MenuEvent.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies in *reasonId* a reason for generating a MenuEvent, where *reasonId* is a MenuReasons constant.

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 **id** example.

MouseEvent type

A MouseEvent object answers questions about the mouse, including

- where the mouse is located
- was a mouse button clicked
- which mouse button was clicked or held down during an operation

The following built-in object variables are useful when you work with the MouseEvent objects `lastMouseClicked` and `lastMouseRightClicked`.

Many methods defined for the MouseEvent type use or return Point values. 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.

getPosition method

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 method convertPointWithRespectTo defined for the UIObject type can be used to convert values in different frameworks.

The following built-in event methods are triggered by MouseEvents: **mouseClick**, **mouseDown**, **mouseUp**, **mouseDouble**, **mouseRightUp**, **mouseRightDown**, **mouseRightDouble**, **mouseMove**, **mouseEnter**, and **mouseExit**.

The following table displays the methods for the MouseEvent type, including several derived methods from the Event type.

Methods for the MouseEvent type

Event	←	MouseEvent
errorCode		getPosition
getTarget		getObjectHit
isFirstTime		isControlKeyDown
isPreFilter		isFromUI
isTargetSelf		isInside
reason		isLeftDown
setErrorCode		isMiddleDown
setReason		isRightDown
		isShiftKeyDown
		setControlKeyDown
		setInside
		setLeftDown
		setMiddleDown
		setPosition
		setRightDown
		setShiftKeyDown
		setX
		setY
		x
		y

getPosition method

MouseEvent

Returns the mouse position as a Point.

Syntax

- getPosition (var p Point)
- getPosition (var xPos LongInt, yPos LongInt)

getObjectHit method

Description

getMousePosition returns the mouse position. This method gets the mouse position at the time the method was called. It doesn't track subsequent mouse movements.

Syntax 1 stores the value in a Point variable, *p*. When you use Syntax 1, you can use Point type methods (for example, **isLeft** and **isRight**) to get more information.

Syntax 2 stores the value in *xPosition* and *yPosition*, two LongInt variables that represent the x and y coordinates of the pointer.

Example

The following example gets the position of the last **mouseUp** event and draws a small circle at that position. The method 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, 1440, 1440)
    crObj.Color = DarkBlue
    crObj.Visible = True
endif
endMethod
```

getObjectHit method

MouseEvent

Creates a handle to the UIObject that received the event.

Syntax

```
getObjectHit ( var target UIObject ) Logical
```

Description

getObjectHit returns in *target* a handle to the UIObject that was clicked. This method is useful for the internal MouseEvents that call the built-in event methods **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**) and 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
```

isControlKeyDown

```
        eventInfo.getTarget(targObj)
        eventInfo.getObjectHit(hitObj)
        message(targObj.Name + " vs. " + hitObj.Name)
    else
        ;code here executes just for form itself
    endif
endMethod
```

isControlKeyDown

MouseEvent

Reports whether the user has held (or is holding) down CTRL during a MouseEvent.

Syntax

```
isControlKeyDown ( ) Logical
```

Description

isControlKeyDown returns True if CTRL is held down during a MouseEvent; otherwise, it returns False.

Example

The following 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
```

The following code 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
```

isFromUI method

MouseEvent

Reports whether an event was generated by the user interacting with Paradox.

Syntax

```
isFromUI ( ) Logical
```

Description

isFromUI reports whether an event was generated by the user interacting with Paradox, or internally (e.g., by an ObjectPAL statement). This method returns True if the event was generated by the user; otherwise, it returns False.

isInside method

Example

Sometimes you need to know whether a `MouseEvent` was generated by the user interacting with the form or by ObjectPAL; for example, in a computer tutorial. In the following example, **isFromUI** is used to determine whether a button's built-in **mouseEnter** method was triggered by the user or by ObjectPAL:

```
;btnOpenCust :: mouseEnter
method mouseEnter(var eventInfo MouseEvent)
  if eventInfo.isFromUI() then
    message("This button opens the customer form.")
  else
    message("After you press this button, the customer form opens.")
  endIf
endMethod
```

isInside method

MouseEvent

Reports whether the mouse is inside the border of the target object.

Syntax

```
isInside ( ) Logical
```

Description

isInside reports whether the mouse is within the border of the target object at the time of the event.

Example

In the following 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
```

isLeftDown method

MouseEvent

Reports whether the left mouse button is held down during a `MouseEvent`.

Syntax

```
isLeftDown ( ) Logical
```

Description

isLeftDown returns `True` if the left mouse button is held down during a `MouseEvent`, for example, 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 mouse button is down at the time of the move. If the left mouse button is down, the field is selected from the point of the click to the beginning of the field. If the right mouse button is down, the field is selected from the point of the click to the end of the field.

isMiddleDown method

```
; 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
```

isMiddleDown method

MouseEvent

Reports whether the middle mouse button is held down during a MouseEvent.

Syntax

```
isMiddleDown ( ) Logical
```

Description

isMiddleDown 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

The following 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 and 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
```

The following 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 and 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
```

isRightDown method

MouseEvent

Reports whether the right mouse button is held down during a MouseEvent.

Syntax

```
isRightDown ( ) Logical
```

isShiftKeyDown

Description

isRightDown returns True if the right mouse button is held down during a MouseEvent, for example, 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 mouse button is down, the field is selected from the point of the click to the beginning of the field; if the right mouse 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
```

isShiftKeyDown

MouseEvent

Reports whether SHIFT is held down during a MouseEvent.

Syntax

```
isShiftKeyDown ( ) Logical
```

Description

isShiftKeyDown 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 cursor 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
```

setControlKeyDown method

MouseEvent

Simulates pressing and holding CTRL during a MouseEvent.

Syntax

```
setControlKeyDown ( const yesNo Logical )
```

Description

setControlKeyDown 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.

setInside method

Example

The following example creates a `MouseEvent` and sets `CTRL` to `Yes`. The event is then sent to the **mouseUp** built-in event 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
    CTRLMouseEvent MouseEvent          ; declare the event
endVar

CTRLMouseEvent.setControlKeyDown(Yes) ; set the Control key
lcField.mouseUp(CTRLMouseEvent)       ; send the event
endMethod
```

The following 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
```

setInside method

MouseEvent

Sets the mouse to be inside the current object.

Syntax

```
setInside ( const TrueFalse Logical ) Logical
```

Description

setInside sets the `MouseEvent` to be inside the current object.

Example

In the following example, the **mouseUp** method for *sendAnEvent* uses **setInside** to change the *eventInfo* variable and then sends the event to *buttonOne*.

```
; sendAnEvent::mouseUp
method mouseUp(var eventInfo MouseEvent)
eventInfo.setInside(Yes)
buttonOne.mouseUp(eventInfo)
endMethod
```

setLeftDown method

MouseEvent

Simulates clicking the left mouse button.

Syntax

```
setLeftDown ( const yesNo Logical )
```

Description

setLeftDown adds information about the state of the left mouse button for a `MouseEvent`. You must specify `Yes` or `No`. `Yes` means the left mouse button was clicked; `No` means the left mouse button was not clicked.

setMiddleDown method

Example

The following example constructs a **MouseEvent** with the left mouse button set down. The **MouseEvent** is then sent to the **mouseMove** method for *Site_Notes*. The following 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
```

The following 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
```

setMiddleDown method

MouseEvent

Simulates clicking the middle mouse button.

Syntax

```
setMiddleDown ( const yesNo Logical )
```

Description

setMiddleDown 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 mouse button was not clicked.

Example

The following 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 mouse button down and 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
```

setMousePosition method

The following 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 and 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
```

setMousePosition method

MouseEvent

Sets the position of the mouse for an event.

Syntax

1. setMousePosition (const *xPosition* LongInt, const *yPosition* LongInt)
2. setMousePosition (const *p* Point)

Description

setMousePosition 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.

The following 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
```

setRightDown method

MouseEvent

Simulates clicking the right mouse button.

Syntax

setRightDown (const *yesNo* Logical)

Description

setRightDown adds information about the state of the right mouse button for a MouseEvent. You must specify Yes or No. Yes means the right mouse button was clicked; No means the right mouse button was not clicked.

setShiftKeyDown method

Example

The following example constructs a **MouseEvent** with the right mouse 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
```

The following 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
```

setShiftKeyDown method

MouseEvent

Simulates pressing and holding SHIFT.

Syntax

```
setShiftKeyDown ( const yesNo Logical )
```

Description

setShiftDown 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 was not pressed.

Example

The following example creates a **MouseEvent** and sets SHIFT to Yes. The event is then sent to the **mouseUp** built-in event 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
```

The following 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.

setX method

```
; 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
```

setX method

MouseEvent

Specifies the horizontal coordinate of the mouse-pointer position.

Syntax

```
setX ( const xPosition LongInt )
```

Description

setX sets the horizontal coordinate (in twips) of the mouse-pointer position to *xPosition*. Coordinates must be specified relative to the upper-left corner of the current object.

Example

The following 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, like 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
```

The following 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
```

setY method

```
    ui.Color = Red          ; native event
  else
    ui.Color = Green       ; mouse event passed from mouseUp
  endif
endMethod
```

setY method

MouseEvent

Specifies the vertical coordinate of the mouse-pointer position.

Syntax

```
setY ( const yPosition LongInt )
```

Description

setY sets the vertical coordinate (in twips) of the mouse-pointer position to *yPosition*. Coordinates must be specified relative to the upper-left corner of the current object.

Example

See the **setX** example.

x method

MouseEvent

Returns the horizontal coordinate of the mouse-pointer position.

Syntax

```
x ( ) LongInt
```

Description

x returns (in twips) the horizontal coordinate of the mouse-pointer position.

Example

See the **setX** example.

y method

MouseEvent

Returns the vertical coordinate of the mouse-pointer position.

Syntax

```
y ( ) LongInt
```

Description

y returns (in twips) the vertical coordinate of the mouse-pointer position.

Example

See the **setX** example.

MoveEvent type

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 following built-in event methods are triggered by MoveEvents: **arrive**, **canArrive**, **canDepart**, and **depart**.

The MoveEvent type includes several derived methods from the Event type.

getDestination method

Methods for the MoveEvent type

Event	←	MoveEvent
errorCode		getDestination
getTarget		reason
isFirstTime		setReason
isPreFilter		
isTargetSelf		
setErrorCode		

getDestination method

MoveEvent

Reports which object is the destination of a move.

Syntax

```
getDestination ( var dest UIObject )
```

Description

getDestination returns in *dest* the object that Paradox is trying to move to in a form.

Example

In the following 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
endif
```

reason method

```
        endIf
    else
        ;code here executes just for form itself
    endIf
endMethod
```

reason method

MoveEvent

Reports why a move occurred.

Syntax

```
reason ( ) SmallInt
```

Description

reason returns an integer value to report why a MoveEvent occurred. MoveEvent reasons occur when a built-in **arrive**, **depart**, **canArrive**, or **canDepart** method is called. ObjectPAL provides MoveReasons constants for testing the value returned by **reason**.

Example

In the following example, assume a form contains two field objects, *fieldOne* and *fieldTwo*, and a button named *moveToFieldOne*. A move 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
```

setReason method

MoveEvent

Specifies a reason for a Move Event.

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies a reason for generating a MoveEvent. This method takes a MoveReasons constant as an argument.

setReason method

Example

In the following 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.

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 **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
```

Number type

Number variables represent floating-point values consisting of a significand (fractional portion, for example, 3.224) multiplied by a power of 10. The significand contains up to 18 significant digits, and the power of 10 ranges from $\pm 3.4\text{E-}4930$ to $\pm 1.1\text{E}4930$. Assigning values outside of this range to a Number variable causes an error.

The following code demonstrates ObjectPAL's alternate syntax:

```
methodName ( objVar, argument [ , argument ] )
```

methodName is 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)
```

Use ObjectPAL's standard syntax for clarity and consistency and use the alternate syntax only where convenient.

Although the numeric method's display formats may vary depending on the user's Windows number format, ObjectPAL's internal representation is always the same.

setReason method

Run-time library methods and procedures defined for the Number type also work with LongInt and SmallInt variables. In all cases, the syntax is the same, and the returned value is a Number. Although sin does not appear in the list of methods for the LongInt type, the following code executes:

```
var
  abc LongInt
  xyz Number
endVar
abc = 43
xyz = abc.sin()
```

The following table displays the methods of the Number type, including several derived methods from the AnyType type:

Methods for the Number type

AnyType	←	Number
blank		abs
dataType		acos
isAssigned		asin
isBlank		atan
isFixedType		atan2
view		ceil
		cos
		cosh
		exp
		floor
		fraction
		fv
		ln
		log
		max
		min
		mod
		number
		numVal
		pmt
		pow
		pow10
		pvt
		rand
		round
		sin
		sinh
		sqrt
		tan
		tanh
		truncate

abs method**Number**

Returns the absolute value of a number.

Syntax

```
abs ( ) Number
```

Description

abs removes the sign from a number.

Example

The following example assumes that a form contains three field objects: *forecastAmt*, *actualAmt*, and *diffPercent*. The **newValue** method for *actualAmt* calculates the difference between *forecastAmt* and *actualAmt* and then determines the accuracy of the forecast. The difference between *forecastAmt* and *actualAmt* can be positive or negative. **abs** returns the absolute value of the number, which is then multiplied by 100 to determine the percentage of error. 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
```

acos method**Number**

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 (i.e., between -pi/2 and pi/2). **acos** is the inverse of cos - if $\text{acos}(x) = y$ and then $\text{cos}(y) = x$.

Example

The following example uses **pushButton** method for the *findArcCos* button to calculate and display the arc cosine of a value:

asin method

```
; findArcCos::pushButton
method pushButton(var eventInfo Event)
  var
    nuUserVal,
    nuArcCos   Number
    stPrompt   String
  endVar

  stPrompt = "Enter a number from -1 to 1"
  nuUserVal = 0

  nuUserVal.view(stPrompt)
  if (nuUserVal = -1) and (nuUserVal 1) then
    nuArcCos = nuUserVal.acos()
    nuArcCos.view("Arc cosine of " + String(nuUserVal))
  else
    msgStop("You entered: " + String(nuUserVal), stPrompt)
  endIf
endMethod
```

asin method

Number

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. **asin** is the inverse of **sin** — if **asin**(x) = y and then **sin**(y) = x .

Example

In the following example, the **pushButton** method for the *findASin* button displays the arc sine of a number.

```
; findASin::pushButton
method pushButton(var eventInfo Event)
  var
    x Number
  endvar
  x = .5
  msgInfo("arc sine of .5", x.asin()) ; displays .52
endMethod
```

atan method

Number

Returns the 2-quadrant arctangent of a number.

Syntax

`atan ()` Number

Description

Given a tangent in radians, **atan** returns the angle in radians. **atan** is called the 2-quadrant arctangent because it returns values within quadrants 1 and 4 (i.e., between $-\pi/2$ and $\pi/2$). **atan** is the inverse of **tan** — if **atan**(x) = y and then **tan**(y) = x .

Example

In the following example, the **pushButton** method for *getAtan* calculates the 2-quadrant arctangent of *x* and *y*:

```
; 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
```

atan2 method**Number**

Returns the 4-quadrant arctangent of a number.

Syntax

```
atan2 ( const x Number ) Number
```

Description

Given a sine in radians, **atan2** returns an angle in radians with cosine *x*. **atan2** is called the 4-quadrant arctangent because it returns values in all four quadrants.

Example

The following example assumes that a form contains a button named *getAtan2*. The **pushButton** method for *getAtan2* calculates the 4-quadrant arctangent of *x* and *y* and 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
```

ceil method**Number**

Rounds a numeric expression up to the nearest whole number.

Syntax

```
ceil ( ) Number
```

Description

ceil rounds a numeric expression up (toward positive infinity) to the nearest whole number.

cos method

Example

In the following example, the **pushButton** method for a button named *ceilVsRound* calculates the ceiling value of a number and then displays the rounded value of that 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
```

cos method

Number

Returns the cosine of an angle.

Syntax

cos () Number

Description

cos returns a value between -1 and 1 representing the cosine of an angle in radians.

Example

In the following 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
```

cosh method

Number

Returns the hyperbolic cosine of an angle.

Syntax

cosh () Number

Description

cosh returns the hyperbolic cosine of an angle in radians. **cosh** uses the following formula:

$\text{cosh}(\text{angle}) = (\exp(\text{angle}) + \exp(-\text{angle})) / 2$

Example

The following example uses the **pushButton** method for the *findCosineH* button to calculate and display the hyperbolic cosine of a 60 degree angle:

```
; 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
```

exp method

Number

Returns the exponential (base e) of a number.

Syntax

```
exp ( ) Number
```

Description

exp computes e to the x power, where the constant e is 2.7182845905 (the so-called natural number), and the return value is the exponent x . The inverse method is the natural log, **ln**.

Example

In the following 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
```

floor method

Number

Rounds a numeric expression down to the nearest whole number.

Syntax

```
floor ( ) Number
```

Description

floor 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
```

fraction method

Number

Returns the fractional portion of a number.

fv method

Syntax

```
fraction ( ) Number
```

Description

fraction returns the fractional portion of a number (i.e., the part to the right of the decimal).

Example

In the following 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
```

fv method

Number

Returns the future value of a series of equal payments.

Syntax

```
fv ( const interestRate Number, periods Number ) Number
```

Description

fv returns the future value of a series of equal payment periods, invested at an interest rate specified by *interestRate*. *interestRate* is expressed as a decimal number. Ensure that the rate period matches the deposit period (i.e., if the deposits are monthly, the interest rate is also monthly).

fv uses the following formula:

$$FV = \text{payment}(\text{pow}(1 + \text{rate}, \text{periods}) - 1) / \text{rate}$$

fv is also called the future or compound value of an annuity because it calculates the amount accumulated in an annuity fund when making regular, equal payments over time.

Example

The following example calculates how much a 14.5% Individual Retirement Account is worth if \$166.67 is deposited each 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!")
```


In method

```
; displays "Depositing 166.67 a month for 30 years
;          at 14.50% yields 1,027,394.23 ..."
endMethod
```

In method

Number

Returns the natural logarithm of a numeric expression.

Syntax

```
ln ( ) Number
```

Description

ln calculates the natural logarithm to the base e of a positive value. The constant e is the natural number, approximated by the value 2.7182845905. If the specified 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
```

log method

Number

Returns the base 10 logarithm of a numeric expression.

Syntax

```
log ( ) Number
```

Description

log returns the base 10 logarithm of a value or numeric expression. If the specified value is 0 or negative, **log** fails.

Use **ln** to compute natural logarithms.

Example

The following example uses the a button's **pushButton** method to calculate and display the base 10 logarithm of a value.

```
; 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
```

max procedure

```
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
```

max procedure

Number

Returns the larger of two numbers.

Syntax

```
max ( const x1 AnyType, const x2 AnyType ) AnyType
```

Description

max returns the larger of two values specified by *x1* and *x2*.

Example

The following example, calculates a medical deduction for tax purposes. The **pushButton** method for *findMedDeduct* calculates the maximum of 7.5% of *AGI* or *medExpense* and then deducts 7.5% of *AGI* from the result. Finding the maximum number first ensures that the calculation returns a positive 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
```

min procedure

Number

Returns the smaller of two numbers.

Syntax

```
min ( const x1 AnyType, const x2 AnyType ) AnyType
```

Description

min returns the smaller of two values specified by *x1* and *x2*.

Example

The following example calculates the maximum amount of tax-deductible charitable contributions when no more than 30% of the adjusted gross income can be deducted. The **pushButton** method for the *findCharityDeduct* button calculates 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

```

mod method

Number

Returns the remainder when one number is divided by another.

Syntax

```
mod ( const modulo Number ) Number
```

Description

mod returns the remainder (or modulus) when one number is divided by the value of *modulo*. If the number is greater than the value of *modulo*, **mod** returns the remainder. If the number is less than *modulo*, **mod** returns the number. If the number equals *modulo*, **mod** returns 0. The following table illustrates each scenario:

Fraction	ObjectPAL code	Return value
5/2	<pre>num = 5 num.mod(2)</pre>	1
2/5	<pre>num = 2 num.mod(5)</pre>	2
2/2	<pre>num = 2 num.mod(2)</pre>	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 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 8
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

```

number procedure

Number

Casts a value as a Number.

numVal procedure

Syntax

```
number ( const value AnyType ) Number
```

Description

number casts *value* to a Number. *value* must be in the form of a valid number that can be entered in a field. When a numeric operand is required in an expression, or when a numeric argument is required in a procedure or method, **number** is used to cast a non-numeric type to a Number. **number** behaves the same as numVal.

Example

In the following example, a variable *x* is declared as a String and then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
    x String
endVar
x = "1123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", Number(x) * 2) ; displays 2,247.08
endMethod
```

numVal procedure

Number

Casts a value as a Number.

Syntax

```
numVal ( const value AnyType ) Number
```

Description

numVal casts *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 and then assigned a string of numbers. The **pushButton** method for the *showDouble* button casts *x* to a Number before doubling it:

```
; showDouble::pushButton
method pushButton(var eventInfo Event)
var
    x String
endVar
x = "1123.54"
; cast x to a Number before multiplying by 2
msgInfo("Double " + x + " is", numVal(x) * 2) ; displays 2,247.08
endMethod
```

pmt method

Number

Returns the periodic payment required to pay off a loan.

Syntax

```
pmt ( const interestRate Number, const periods Number ) Number
```

Description

pmt returns the constant, regular payment required to pay off a loan. **pmt** uses the following formula:

$$PMT = p * i / (1 - (1 + i)^{-t})$$

(where p = principal amount, i = effective interest rate per period, and t = term of the loan or number of payment periods).

Payments are due at the end of each period.

pmt works for amortization-type loans (e.g., 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 (e.g., repayments of credit accounts or automobile loans).

The interest rate used in **pmt** is a decimal number. Ensure that the rate period matches the payment periods (i.e., if the payments are monthly, the interest rate should also be monthly). Because the interest rate for amortization loans (mortgages) is usually annual, you can divide it by 12 for monthly payments or by 4 for quarterly payments.

Use the nominal annual interest rate quoted instead of 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 a 12% interest rate:

```
; 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 " +
    String(loanAmt) + " at " + String(intRate * 12 * 100) +
    "% interest for " + String(SmallInt(numPayments)) +
    " months is " + String(monthlyPayment))    ; payment is $47.07
endMethod
```

pow method**Number**

Raises a number to a specified power.

Syntax

```
pow ( const exponent Number ) Number
```

Description

pow returns the value of a number raised to the power specified in *exponent*. If the return value is larger than 1E308 or smaller than 1E-308, **pow** returns an error.

Example

In the following example, the **pushButton** method for the *raiseTwo* button calculates and displays the result:

pow10 method

```
; 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
```

pow10 method

Number

Calculates 10 to a specified power.

Syntax

`pow10 () Number`

Description

pow10 returns the value of 10 raised to a specified power.

Example

In the following example, the **pushButton** method for the *raiseTen* button calculates 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
```

pv method

Number

Returns the current value of a series of equal payments.

Syntax

`pv (const interestRate Number, const periods Number) Number`

Description

pv calculates the current value of equal, regular payments on a loan (or withdrawals from an investment) at a rate specified in *interestRate* for a term specified in *periods*. The payments reduce the principal, but the remaining balance continues to generate and compound interest.

pv uses the following formula:

$$PV = \text{payment} * (1 - \text{rate})$$

(where *n* is the number of periods)

The interest rate used in **pv** is expressed as a decimal number. Ensure that the rate period matches the payment period (i.e., if the payments are monthly, the interest rate should also be monthly). Use **pv** to calculate the size of the 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 makes regular, equal payments to you over time. For this reason, **pv** is also called the present value of an annuity.

Example

The following example assumes that 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 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("$C", mortgage))    ; displays $149,138
endMethod
```

Imagine 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 code uses the **pushButton** method for the *findAnnuity* button to calculate 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(ShortInt(term / 12)) +
    " years, the original amount must be " +
    String(investment))    ; displays 357,544.07
endMethod
```

rand procedure

Number

Generates a random value ranging from 0 to 1.

Syntax

rand () Number

round method

Description

rand 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
```

round method

Number

Rounds a number to a specified number of decimal places.

Syntax

```
round ( const places SmallInt ) Number
```

Description

round 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. This code 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
```

sin method

Number

Returns the sine of an angle.

Syntax

```
sin ( ) Number
```


sinh method

Description

sin returns a number between -1 and 1 representing the sine of an angle in radians.

Example

The following example uses the **pushButton** method for the *findSin* button to calculate 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
```

sinh method

Number

Returns the hyperbolic sine of an angle.

Syntax

sinh () Number

Description

sinh returns the hyperbolic sine of an angle in radians. **sinh** uses the following formula:

$$\sinh(\text{angle}) = (\exp(\text{angle}) - \exp(-\text{angle})) / 2$$

Example

In the following example, the **pushButton** method for the *getHSine* button calculates 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
```

sqrt method

Number

Returns the square root of a number.

Syntax

sqrt () Number

Description

sqrt returns the square root of a positive value or numeric expression.

tan method

Example

In the following example, the **pushButton** method for the *getSqrt* button assigns the value from *fieldOne* (an unbound field object) to *x*. If *x* is positive, the code then 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
```

tan method

Number

Returns the tangent of an angle.

Syntax

tan () Number

Description

tan returns the tangent of an angle in radians. **tan** diverges at $-\pi/2$, $\pi/2$, and every $\pm \pi$ radians from those values.

Example

In the following 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
```

tanh method

Number

Returns the hyperbolic tangent of an angle.

Syntax

tanh () Number

Description

tanh returns the hyperbolic tangent of an angle in radians. **tanh** uses the following formula:

$\tanh (angle) = \sinh (angle) / \cosh (angle)$

truncate method

Example

In the following 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
```

truncate method

Number

Shortens a number to a specified number of decimal places.

Syntax

```
truncate ( const places SmallInt ) Number
```

Description

truncate returns a number truncated toward 0 to the number of decimal places specified in *places*.

truncate does not round the value.

Example

In the following example, the **pushButton** method for the *chopAValue* button assigns the value from *fieldOne* (an unbound field object) to *x*. the code then 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
```

OLE type

Object Linking and Embedding (OLE) is a protocol that allows you to access another application without leaving Paradox.

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 functionality of the paint program available to your users (assuming, of course, that your users have the paint program installed on their systems).

Note

- ObjectPAL and Paradox also support Dynamic Data Exchange (DDE) another protocol that allows you to share data among applications.

canLinkFromClipboard method

The following terms are used when discussing OLE operations:

<i>OLE server</i>	An application that uses the OLE mechanism to provide access to its documents. Paradox is an OLE server.
<i>OLE container</i>	An application that uses the OLE mechanism to access documents created by an OLE server. Paradox is an OLE container.
<i>OLE object</i>	A document created using an OLE server. A document that contains the data you want to use in your Paradox application.
<i>OLE variable</i>	An ObjectPAL variable declared as an OLE type. An OLE variable provides a handle for manipulating an OLE object. You can use OLE variables in ObjectPAL code to manipulate OLE objects.
<i>Asynchronous</i>	Code in each application executes independently (i.e., one application does not wait for the other). When you use a method that launches an OLE server for user input, declare the OLE variable in a Var window or in a method window above the method keyword. This ensures that the OLE variable is and in scope, even if the method finishes before the server application is closed.

The following table lists the methods for the OLE type, including several derived methods from the AnyType type.

Methods for the OLE type

AnyType	←	OLE
blank		canLinkFromClipboard
dataType		canReadFromClipboard
isAssigned		edit
isBlank		enumServerClassNames
isFixedType		enumVerbs
unAssign		getServerName
insertObject		
isLinked		
linkFromClipboard		
readFromClipboard		
updateLinkNow		
writeToClipboard		

canLinkFromClipboard method

OLE

Reports whether an OLE object can be linked from the Clipboard to an OLE variable.

Syntax

canLinkFromClipboard () Logical

Description

canLinkFromClipboard returns True if an OLE object can be linked from the Clipboard to an OLE variable; otherwise, it returns False. After an OLE object is linked from the Clipboard, changes made to the OLE object, while in Paradox, affect the underlying file.

canLinkFromClipboard is useful in a routine that determines whether a **linkFromClipboard** operation is possible. A menu item is dimmed and inactive when **canLinkFromClipboard** returns False.

Example

The following example attempts to link an OLE object from the Clipboard to a field in a specified record in a table. If the OLE object can't be linked, this code prompts the user to embed or read the OLE object instead.

```
; btnLinkOrRead::pushButton
method mouseClick(var eventInfo MouseEvent)
    var
        stReadOLE      String
        oleObj          OLE
        tcEmployee      TCursor
    endVar

    ; Move to specified record in table.
    tcEmployee.open("employee")
    tcEmployee.locate("EmpName", "Frank Corel")

    ; Link if you can, otherwise read (embed).
    switch
        case oleObj.canLinkFromClipboard() :
            oleObj.linkFromClipboard()

        case oleObj.canReadFromClipboard() :
            stReadOLE = msgQuestion("Can't link OLE object.",
                                    "Do you want to embed it instead?")
            if stReadOLE = "Yes" then
                oleObj.readFromClipboard()
            else
                message("No update.")
                return
            endif

        otherwise :
            msgInfo("Can't link or embed the OLE object.",
                    "The Clipboard may be empty.")
            return
    endSwitch

    ; Update the table.
    tcEmployee.edit()
    tcEmployee.VoiceSample = oleObj
    tcEmployee.endEdit()
    message("Update complete")
endMethod
```

canReadFromClipboard method

OLE

Reports whether an OLE object can be embedded from the Clipboard to an OLE variable.

Syntax

```
canReadFromClipboard ( ) Logical
```

edit method

Description

canReadFromClipboard returns True if an OLE object can be embedded or read 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.

canReadFromClipboard is useful in a routine that determines whether a readFromClipboard operation is possible. A menu item is dimmed and inactive when **canReadFromClipboard** returns False.

Example

See the **canLinkFromClipboard** example.

edit method

OLE

Launches the OLE server and allows the user to edit the object or perform another action.

Syntax

```
edit ( const oleText String, const verb SmallInt ) Logical
```

Description

edit 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 displays *oleText* in the Title Bar. *edit* passes *verb* to the application server to specify an operation.

verb is an integer that corresponds to one of the OLE server's action constants. The meaning of *verb* varies from application to application — a *verb* that is appropriate for one application may not be appropriate for another. Use **enumVerbs** to determine which verbs the server supports and then select a verb for the call to **edit**.

If you want to launch an OLE server without using **enumVerbs**, use 0 for *verb* - this value represents the primary verb, and should be supported by all OLE servers.

Example

The following example assumes that the *Pics* table stores Paintbrush graphics in an OLE field. The table has two fields: PicName (A8) and PicData (O). When you click *editButton*, this code locates a record in the table and uses **edit** to invoke Paintbrush (enabling the user to edit the graphic in the OLE field). When you click *updateButton*, the code updates the *Pics* table.

Code is attached to the page's Var window, to the *editButton*'s **pushButton** method, and to the *updateButton*'s **pushButton** method. Variables are declared in the page's Var window for two reasons: to make them available to both buttons; it ensures the OLE variable is available, even if **edit** finishes executing before Paintbrush is closed.

The page's Var window contains the following code:

```
var
  olePic  OLE
  picTC   TCursor
endVar
```

The *editButton*'s **pushButton** method contains the following code:

```
method pushButton(var eventInfo Event)
  if picTC.open ("pics.db") then
    if picTC.locate("PicName", "blueLine") then

      ; The PicData field stores OLE objects
      ; created using Paintbrush.
```

```

        olePic = picTC.PicData

        ; Launch Paintbrush so user can edit the bitmap.
        olePic.edit("PDOXWIN", 0)
    else
        msgStop("Stop", "Couldn't find blueLine.")
    endIf
else
    msgStop("Stop", "Couldn't open table.")
endIf
endMethod

```

The *updateButton*'s **pushButton** method contains the following code:

```

method pushButton(var eventInfo Event)
    picTC.edit()
    picTC.PicData = olePic
    picTC.endEdit()
    picTC.close()
endMethod

```

enumServerClassNames method

OLE

Lists the registered OLE servers.

Syntax

```
enumServerClassNames ( var serverClasses DynArray[ ] String ) Logical
```

Description

enumServerClassNames lists the OLE servers registered on the user's system. The information is assigned to *serverClasses*, a dynamic array (DynArray) that you must declare and pass as an argument. This method returns True if it succeeds; otherwise, it returns False.

The DynArray's indexes are the end-user server names (e.g., Paradox Table), and the corresponding items are the internal OLE names.

Use **enumServerClassNames** to pass a server name to **insertObject**.

Example

See the **insertObject** example.

enumVerbs method

OLE

Lists the actions supported by an OLE server.

Syntax

```
enumVerbs ( var verbs DynArray[ ] SmallInt ) Logical
```

Description

enumVerbs creates a dynamic array (DynArray) listing the action commands or *verbs* supported by the OLE server associated with an OLE variable.

When you associate an OLE variable with an OLE object, Paradox recognizes the server application which generated the object. OLE methods like **enumVerbs** and **getServerName** allow you to ask questions.

enumVerbs requests the server for a list of supported verbs and then loads them into a DynArray. Each DynArray index corresponds to the name of a specific action (i.e., DynArray items correspond to

enumVerbs method

the action constant used by the server). Because each verb's meaning varies from application to application, you must know which verb to pass to the server to instruct it to do what you want.

Windows Paintbrush is an OLE server that has only one action command (Edit, with a value of 0). The following code a Paintbrush graphic from the Clipboard and generates a dynamic array using **enumVerbs**. This code then displays the DynArray's contents 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
```

This code assumes the Clipboard contains an OLE object (a graphic image) 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 (&Edit). The ampersand prefix is especially useful when you want to display action names in a menu. Paradox recognizes the ampersand as a special character and displays &Edit as Edit. E is designated as an accelerator key.

For more information, see Menu methods.

Example

The following example assumes the *Sounds* table contains an alpha field named SoundName and an OLE field named SoundData. Data displayed in the OLE field is copied from the Windows Sound Recorder to the Clipboard. The following code uses *enumVerbs* to create a pop-up menu that lists the verbs (actions) for Sound Recorder when you click a button named *btnEditSounds*. Because Sound Recorder supports two actions (Edit and Play), this example allows the user to edit or play the sound contained in the OLE field.

The following code is attached to the button's Var window and declares the OLE variable. Declaring the OLE variable in the Var window, ensures that the variable is available, even if the method finishes before the server application is closed.

```
; btnEditSounds::Var
Var
  oleVar OLE
endVar
```

The following code is attached to the button's built-in **pushButton** method. It builds and displays a pop-up menu and launches the server application.

```
; btnEditSounds::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 field value to OLE var
    oleVar.enumVerbs(verbs)      ; Get list of Sound Recorder actions.
```



```

    forEach tagName in verbs ; Create a pop-up menu of verbs.
    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 Edit is selected
if not mChoice.isBlank() then
    oleVar.edit("PdoxWin", verbs[mChoice])
endIf

else
    errorShow("Can't find " + soundName + ".")
endIf
else
    errorShow("Can't open " + tblName + ".")
endIf

endMethod

```

getServerName method

OLE

Reports the name of the OLE server for an OLE object.

Syntax

```
getServerName ( ) String
```

Description

getServerName reports the name of the OLE server for an OLE object. *getServerName* is especially useful when you want to provide the user with the OLE server name.

Example

The following example assumes that the *Media* table has an alpha field named *MediaName*, an alpha field named *ServerName*, and an OLE field named *MediaData*. This code scans through *Media*'s records placing 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

```

insertObject method

OLE

Inserts a linked or embedded OLE object into an OLE variable.

Syntax

1. insertObject () Logical
2. insertObject (const *fileName* String , const *link* Logical) Logical
3. insertObject (const *className* String) Logical

Description

insertObject assigns a linked or embedded OLE object to an OLE variable. This method returns True if it succeeds; otherwise, it returns False.

Syntax 1 invokes the Insert Object dialog box. The user must supply any necessary information and close the dialog box. For example, the user can choose Create New to insert a new OLE object or Create From File to insert an existing OLE object from a file.

Syntax 2 inserts an object from the file specified in *fileName* without launching the server application for user input. The argument *link* specifies whether to link to the file. If *link* is True, changes made to the object in Paradox are reflected in the underlying file. If *link* is False, changes made in Paradox do not affect the file.

Syntax 3 launches the server application for user input and inserts an object from the class specified in *className*. *className* is the name of a registered OLE server class. Use **enumServerClassNames** to view a list of OLE server class names.

Note

- When creating a new file, the server application may prompt the user for file creation information.

Example 1

In the following example, a form contains buttons named *btnInsertOLE* and *btnEditOLE* and a field object named *mugShot*. *mugShot* is bound to an OLE field named MugShot in a table in the form's data model. The variables *oleVar* and *loInserted* are declared in the page's Var window to make them available to both buttons, and to ensure that the OLE variable is available if a method finishes before the server application is closed.

The following code is attached to the page's Var window. It declares the OLE variable named *oleVar* and a Logical flag variable named *loInserted* that tracks whether an OLE object was inserted into the OLE variable.

```
; thePage::Var
Var
    oleVar          OLE
    loInserted      Logical
endVar
```

The following code is attached to the **pushButton** method of *btnInsertOLE*. It displays the Insert Object dialog box, allowing the user to insert an OLE object:

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

    if not oleVar.insertObject() then ; Invoke Insert Object dialog box.
        errorShow()
        loInserted = No
        return
    else
```

```

        loInserted = Yes
    endIf

```

```

endMethod

```

The following code is attached to the **pushButton** method of *btnEditOLE*:

```

; btnEditOLE :: pushButton
method pushButton(var eventInfo Event)

    if not loInserted.isAssigned() then
        loInserted = No
    endIf

    if loInserted = Yes then
        edit()
        mugShot.Value = oleVar
        loInserted = No ; Reset the flag.
        endEdit()
    else
        msgInfo("No OLE object to insert.",
            "Click the Insert button.")
    endIf

endMethod

```

Example 2

In the following example, a form contains a button named *btnInsertOLE* and a field object named *fldOLE*. *fldOLE* is bound to an OLE field in a table in the form's data model. The **pushButton** method uses an OLE variable *oleVar* and **insertObject** to read a wave file into the OLE variable named *oleVar*. The code then assigns the file to the field *fldOLE*. This example does not launch the server application for user input.

```

;btnInsertOLEFile :: pushButton
const
    ; Changes made in Paradox will not
    ; affect the underlying file.
    kNoLink = False
endConst

var
    oleVar OLE
endVar

method pushButton(var eventInfo Event)
    var
        stFileName,
        stPrompt String
    endVar

    stPrompt = "Type the filename here."
    stFileName = stPrompt
    stFileName.view("Enter a filename.")
    if stFileName = stPrompt then
        return ; User didn't type a filename and click OK.
    endIf

    if oleVar.insertObject(stFileName, kNoLink) then
        edit()
        fldOLE.Value = oleVar
        endEdit()
    endIf
endMethod

```

insertObject method

```
    else
        errorShow("Could not insert OLE object: " + stFileName)
    endIf
endMethod
```

Example 3

Imagine that you are using Paradox to maintain and publish a database for a school and each record represents a course syllabus. Since different instructors prefer different word processors, you can store syllabus data in an OLE field and let the instructors edit it any application that is an OLE server.

The following example assumes a form contains a table frame bound to the *Courses* table and that each record in the table frame contains a field object named *Syllabus*. The following code is attached to a button named *btnAddSyllabus* that allows the user to add a new syllabus to the table. This code displays a list of the OLE server applications installed in the user's system in a pop-up menu. When the user chooses an application name from the pop-up menu, the call to **insertObject** inserts an object of the specified type.

```
; btnAddSyllabus :: pushButton
var
    oleVar    OLE
endVar

method pushButton(var eventInfo Event)
    var
        puServers    PopUpMenu
        stOLEServer,
        stUserServer  String
        dyOLEServers  DynArray[] String
    endVar

    ; Specify a title for the pop-up menu.
    puServers.addStaticText("Choose one:")
    puServers.addSeparator()

    ; enumServerClassNames returns a DynArray where the keys are
    ; the external names and the corresponding items are the
    ; names used internally by OLE.

    oleVar.enumServerClassNames(dyOLEServers)

    forEach stOLEServer in dyOLEServers
        puServers.addText(stOLEServer)
    endForEach

    stUserServer = puServers.show()
    if stUserServer "" then

        ; insertObject uses the internal name to specify an OLE server.
        if oleVar.insertObject(dyOLEServers[stUserServer]) then
            action(DataBeginEdit)
            Courses.Syllabus.Value = oleVar
            action(DataEndEdit)
        else
            errorShow("Could not insert " + stOLEServer)
        endIf
    else
        return ; User didn't choose a server.
    endIf
endMethod
```

isLinked method**OLE**

Reports whether an OLE object is a linked object.

Syntax

```
isLinked ( ) Logical
```

Description

isLinked returns True if an OLE object is a linked object and False if it is an embedded object. When used with **updateLinkNow**, you can use this method to update the linked OLE fields in a table.

Example

See the **updateLinkNow** example.

linkFromClipboard method**OLE**

Pastes a link between an OLE object from the Clipboard and an OLE variable.

Syntax

```
linkFromClipboard ( ) Logical
```

Description

linkFromClipboard returns True if an OLE object is successfully pasted from the Clipboard and linked to an OLE variable; otherwise, it returns False.

After an OLE object is linked from the Clipboard, changes made while in Paradox affect the underlying file. Compare this method to **readFromClipboard**, where changes made in Paradox do not affect the underlying file.

Example

See the **canReadFromClipboard** example.

readFromClipboard method**OLE**

Pastes an OLE object from the Clipboard into an OLE variable.

Syntax

```
readFromClipboard ( ) Logical
```

Description

readFromClipboard returns True if an OLE object is successfully pasted from the Clipboard into an OLE variable; otherwise, it returns False.

After an OLE object is pasted from the Clipboard, changes made while in Paradox do not affect the underlying file. Compare this method to **linkFromClipboard**, where changes made in Paradox affect the underlying file.

Example

See the **canReadFromClipboard** example.

updateLinkNow method**OLE**

Updates a linked OLE object.

writeToClipboard method

Syntax

```
updateLinkNow ( ) Logical
```

Description

updateLinkNow updates a linked OLE object and returns True if successful. It returns False if the OLE object is an embedded object. You can use this method with `isLinked` to update the linked OLE fields in a table.

Example

The following example scans the *Employee* table and updates any linked values in the OLE field named *VoiceSample*:

```
;btnUpdateLinks::pushButton
method pushButton(var eventInfo Event)
  var
    oleObj      OLE
    tcEmployee   TCursor
  endVar

  tcEmployee.open("employee")
  tcEmployee.edit()

  scan tcEmployee :
    oleObj = tcEmployee.VoiceSample ; VoiceSample is an OLE field.
    if oleObj.isLinked() then
      oleObj.updateLinkNow() ; Update the OLE variable.
      tcEmployee.VoiceSample = oleObj ; Assign the new value to the field in the
underlying table.
    endIf
  endScan

  tcEmployee.endEdit()
endMethod
```

writeToClipboard method

OLE

Copies an OLE variable to the Clipboard.

Syntax

```
writeToClipboard ( ) Logical
```

Description

writeToClipboard copies an original OLE object to the Clipboard. This method erases the Clipboard before copying the OLE object.

This method returns True if an OLE object is successfully copied 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. This code then writes the variable to the Clipboard, where it can be used by Paradox or 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

```

OleAuto type

OLE Automation allows you to manipulate an application's objects from outside that its application. OLE Automation uses OLE's component object model, but can be implemented independently from the rest of OLE. You can use OleAuto methods to create and manipulate objects from an application that exposes objects to OLE.

Methods for the OleAuto type

OleAuto

attach	enumServerInfo
close	first
enumAutomationServers	invoke
enumConstants	next
enumConstantValues	open
enumControls	openObjectTypeInfo
enumEvents	openTypeInfo
enumMethods	registerControl
enumObjects	unregisterControl
enumProperties	version

attach method

OleAuto

Attaches an OLE Automation variable to a UIObject.

Syntax

```
attach ( const object UIObject ) Logical
```

Description

attach attaches an OLE Automation variable to the UIObject specified by *object*. **attach** succeeds if the UIObject denotes an ActiveX control. When **attach** succeeds, the objects methods and properties are accessible from the OLE Automation variable.

Example

The following example attaches to an OLE custom control called MyCtrl that is embedded on the form:

close method

```
method pushButton ( var eventInfo Event )
var
    oa oleauto
endvar
    oa.attach(MyCtrl)
endMethod
```

close method

OleAuto

Closes the OLE Automation variable.

Syntax

```
close ( ) Logical
```

Description

close releases the reference from an OLE Automation variable to an automation server; however, some servers remain open when all references are removed. **close** is especially useful for global variables, because it is called automatically when an OLE Automation variable goes out of scope.

Example

The following example closes the OLE Automation server application:

```
var
    pdx oleauto
endvar
method pushButton ( var eventInfo Event )
    pdx.close()
endMethod
```

enumAutomationServers procedure

OleAuto

Reads the registry on the current machine and lists the available OLE Automation servers.

Syntax

```
enumAutomationServers ( var servers DynArray[ ] String ) Logical
```

Description

enumAutomationServers lists the OLE Automation servers and OLE custom controls in the registry.

The information is assigned to *servers*, a dynamic array that you must declare and pass as an argument. The indexes of the DynArray are the end user OLE Automation server names. The corresponding index values are the internal OLE names (e.g., Paradox.Application).

enumAutomationServers returns True if successful; otherwise, it returns False.

Use **enumAutomationServers** to retrieve the internal server name to pass to **open** and **openTypeInfo**.

Example

The following example demonstrates how **enumAutomationServers** compiles a list of OLE Automation servers:

```
method pushButton ( var eventInfo Event )
var
    da DynArray[ ] String
endVar
enumAutomationServers(da)
da.view()
endMethod
```


enumConstants method**OleAuto**

Enumerates the constants defined by an OLE Automation server.

Syntax

```
enumConstants ( var types DynArray[ ] String ) Logical
```

Description

enumConstants enumerates the constant type names in a type library of an OLE Automation server. The information is assigned to the dynamic array (DynArray) *types*. The indexes hold the OLE type name and the corresponding items are the equivalent ObjectPAL type. You can use the constant type name as input for the **enumConstantValues** to retrieve the constant values of this type. These constants are only available through this method.

Example

The following example enumerates the constants from Excel:

```
method pushButton ( var eventInfo Event )
  var
    oa oleauto
    da DynArray[] String
  endvar
  oa.open("Excel.application.5")
  oa.enumConstants(da)
  da.view("Excel constant types")
endmethod
```

enumConstantValues method**OleAuto**

Enumerates the constants that are accessible from an OLE Automation server.

Syntax

```
enumConstantValues ( const constantType String, var values DynArray[ ] AnyType )
Logical
```

Description

enumConstantValues enumerates the constants in a type library of an OLE Automation object. *constantType* is the type returned by **enumConstants**.

The enumerated information is assigned to the dynamic array (DynArray) *values*. The indexes are the OLE constant names and the corresponding items are the constant's values.

Example

The following example enumerates the values of constants available in Excel:

```
method pushButton ( var eventInfo Event )
  var
    oa oleauto
    da DynArray[] AnyType
  endvar
  oa.open("Excel.Application.5")
  oa.enumConstantValues("Constants",da)
  da.view()
endmethod
```

enumControls procedure

OleAuto

enumControls enumerates the registered OLE custom controls.

Syntax

```
enumControls ( var controls DynArray[ ] String ) Logical
```

Description

enumControls enumerates the OLE custom controls listed in the registry. The information is assigned to the dynamic array (DynArray) *controls*. The DynArray indexes are the end user OLE Automation control names (e.g., "My Own Control"), and the corresponding values are the internal OLE names (e.g., MyCtrl.Ctrl1).

Use **enumControls** to retrieve internal ActiveX names, as input for the **open** and **openTypeInfo** methods. You can also use **enumControls** for the progid property for the OLE object.

Example

The following example builds and displays the Controls dynamic array (DynArray):

```
method pushbutton ( var eventInfo Event )
var
  da DynArray[] String
endvar
  enumControls(da)
  da.view()
endmethod
```

The following example creates a form using an ActiveX control object. MyCtrl.Ctrl1 is an internal ActiveX control name listed by **enumControls** in the previous example.

```
method pushButton ( var eventInfo Event )
var
  f form
  o uiobject
endvar
  f.create()
  o.create(OLETool, 200, 300, 1000, 500, f)
  o.ProgId = "MyCtrl.Ctrl1"
endMethod
```

enumEvents method

OleAuto

Enumerates the events that are accessible from an OLE Automation server.

Syntax

```
enumEvents ( var events DynArray[ ] String ) Logical
```

Description

enumEvents enumerates a controls events. The information is assigned to the dynamic array (DynArray) *events*. The DynArray is empty if the OLE Automation variable is bound to an object that is not an OLE Automation control.

Example

The following example opens the type library of MyCtrl.Ctrl1, and builds and displays the dynamic array (DynArray) of the enumerated events:

```
method pushButton ( var eventInfo Event )
var
  oa oleauto
```

```

    dy DynArray[] String
endvar
    oa.openTypeInfo("MyCtrl.Ctrl1")
    oa.enumEvents(dy)
    dy.view()
endMethod

```

enumMethods method

OleAuto

Enumerates the methods that are accessible from an OLE Automation server.

Syntax

```
enumMethods ( var methods DynArray[ ] String ) Logical
```

Description

enumMethods enumerates the methods that can be accessed from an OLE Automation server. The information is assigned to the dynamic array (DynArray) *methods*. The index of the DynArray is the method name, and its value is the ObjectPAL prototype. Some of these methods might not be accessible by ObjectPAL because their types are not supported, in which case the prototype displays an asterisk character (*).

You can specify argument types with commentary information. For example, MoveCursorToPos(x LongInt {OLE_XPOS_PIXELS}, y LongInt {OLE_YPOS_PIXELS}), where OLE_XPOS_PIXELS is the OLE type of the argument. The OLE type name often indicates the nature of the argument.

Example

The following example builds and displays the dynamic array (DynArray) of the enumerated methods:

```

method viewMethods(var oa oleauto)
var
    dy DynArray[] String
endvar
    oa.enumMethods(dy)
    dy.view()
endMethod

```

enumObjects method

OleAuto

Enumerates the events accessible from an OLE Automation server.

Syntax

```
enumObjects ( var objects DynArray[ ] String ) Logical
```

Description

enumObjects lists the names of objects in a type library of a server. The object names are sub-objects in that particular OLE server. The sub-objects are often retrieved through methods and properties of the Application server object retrieved with the **open** method. This method lists the object names, which can be passed into **openObjectTypeInfo**, from which the methods and properties of the sub-object can be enumerated.

Example

The following example builds and displays the dynamic array (DynArray) of the enumerated objects.

```

method viewObjects ( oa oleauto )
var
    dy DynArray[] String
endvar

```

enumProperties method

```
    oa.enumObjects(dy)
    dy.view()
endmethod
```

enumProperties method

OleAuto

Enumerates the properties accessible from an OLE Automation server.

Syntax

```
enumProperties ( var properties DynArray[ ] String ) Logical
```

Description

enumProperties enumerates the properties that can be accessed from an OLE Automation server. The information is assigned to the dynamic array (DynArray) *properties*. The index of the DynArray is the property name, and the corresponding item is the ObjectPAL type. Some properties aren't accessible by ObjectPAL because their types are not supported. Unsupported ObjectPAL types display an asterisk (*).

Property types might be specified with commentary information. For example, ForeColor LongInt {OLE_COLOR}, BackColor LongInt {OLE_COLOR}, where OLE_COLOR is the OLE type of the argument. The OLE type name often indicates the nature of the argument.

Example

The following example builds and displays a dynamic array (DynArray) of the enumerated properties:

```
method viewProperties(oa oleauto)
var
    dy DynArray[ ] String
endvar
    oa.enumProperties(dy)
    dy.view()
endMethod
```

enumServerInfo procedure

OleAuto

Enumerates information about the OLE Automation server.

Syntax

```
enumServerInfo ( const serverName String, var info DynArray[ ] AnyType ) Logical
```

Description

enumServerInfo enumerates information about the server from the registry. The *serverName* is one of the internal OLE server names returned from either **enumAutomationServers** or **enumControls**.

The following table displays the information enumerated by **enumServerInfo**:

Key	Type	Comment
CLSID	String	The ClassID used internally by OLE. If CLSID exists the server is an ActiveX control.
ProgID	String	The internal OLE server name (e.g., Paradox.Application).
TypeLib	String	The ClassID of the type library. If TypeLib exists, openTypeInfo can be used with this server.
ToolboxBitmap32	Graphic	Toolbar bitmap for the control
Version	String	The internal version of this server

first method

Because the *info* dynamic array (DynArray) only holds information retrieved from the registry, **numServerInfo**'s results depend the server's registry.

Example

The following example builds and displays a dynamic array (DynArray) of the server information:

```
method pushButton ( var eventInfo Event )
var
    da DynArray[] anytype
endvar
enumServerInfo("MyCtrl.Ctrl1", da)
da.view()
endMethod
```

first method

OleAuto

Returns the first object in a collection.

Syntax

```
first ( var AnyType )
```

Description

first returns the first object in a collection when an OLE Automation variable denotes a sub-object in a server that is itself a collection of other sub-objects. The items in a collection are primarily OleAuto type — a reference to another OLE automation object. If the collection is empty, **first** returns a blank value. You can determine if an object is a collection object using **isBlank**. If the object is a collection, **isBlank** succeeds; otherwise, it fails. Some servers do not support **isBlank**.

A collection object behaves like any other OleAuto object. It always has a Count property and an Item method, and most of the time they have an Add and a Remove method. Specific implementations can have other methods and properties available.

Example

The following example uses **first** and **next** to return individual elements of an OleAuto collection object *collectnObj*. *collectnObj* is passed as a parameter to **first**. This example assumes that *collectnObj* is attached to an OleAuto collection object and is declared at the form level.

```
method GetContents ( var collectnObj oleAuto )
Var
    anElement LongInt
    i SmallInt
endVar

collectnObj = OA.first(anElement)
processObject(anElement) ; call a custom method which
                           ; processes the current element
                           ; and pass the element reference

for i = 1 to collectnObj^count() ; call the OleAuto object's count
    collectnObj = OA.next(anElement) ; method to determine loop max
    processObject(anElement) ; get the next item in the
                           ; collection object
endFor
endMethod
```

invoke procedure

OleAuto

Invokes a method or property in an OLE Automation server.

Syntax

```
invoke ( const methodName String [, var arg]* ) AnyType
```

Description

invoke allows you to access methods and properties in an OLE Automation server. The *methodName* argument specifies the OLE Automation server's internal method and the optional *arg* arguments are the parameters of the method specified by *methodName*.

invoke is especially useful when the OLE Automation server has a method or property name that conflicts with an ObjectPAL keyword.

Example

The following example demonstrates three ways to call the **msgbox** method of the passed automation server:

```
method callMsgBox (oa oleauto)
var
    ret LongInt
endvar
    ret = oa.msgbox("Hello", 5)
    ret = oa^msgbox("Hello", 5)

    ret = oa.invoke("msgbox", "Hello", 5)
endMethod
```

next method

OleAuto

Returns the next object in a collection.

Syntax

```
next ( var AnyType )
```

Description

next returns the next object in a collection when an OLE Automation variable denotes a sub-object in a server that is itself a collection of other sub-objects. When there are no more items in the collection, the result will be a blank value. The items in a collection are primarily OleAuto type - a reference to another OLE automation object. If the collection is empty, **next** returns a blank value. You can determine if an object is a collection object using **isBlank**. If the object is a collection, **isBlank** succeeds; otherwise, it fails. Some servers do not support **isBlank**.

A collection object behaves like any other OleAuto object. It always has a Count property and an Item method, and most of the time they have an Add and a Remove method. Specific implementations can have other methods and properties available.

Example

See the **first** example.

open method

OleAuto

Opens a server.

openObjectTypeInfo method

Syntax

```
open ( const serverName String ) Logical
```

Description

open opens the server specified by *serverName*. If the specified server is an automation server, open succeeds; otherwise, it fails.

Example

The following example opens Paradox as an OLE Automation server;

```
var
    pdx oleauto
endvar
method pushbutton ( var eventInfo Event )
    pdx.open("Paradox.Application")
endMethod
```

openObjectTypeInfo method

OleAuto

Enumerates the events that are accessible from an OLE Automation server.

Syntax

```
openObjectTypeInfo ( const server OleAuto, const objectName String ) Logical
```

Description

openObjectTypeInfo connects to the type library of the specified sub-object in a server. Unlike **openTypeInfo**, **openObjectTypeInfo** allows you to use **enumMethods** and **enumProperties** to retrieve the methods and properties of the sub-object specified in *objectName*. The object names can be enumerated by **enumObjects**.

Example

The following example connects to the type library of the sub-object, chart, in Excel. The code then builds and displays a dynamic array (DynArray) of the chart's properties.

```
method pushButton ( var eventInfo Event )
var
    oa oleauto
    excel oleauto
    chart oleauto
    da DynArray[] String
endvar
    excel.openTypeInfo("Excel.application.5")
    chart.openObjectTypeInfo(excel, "chart")
    chart.enumProperties(da)
    da.view()
endMethod
```

openTypeInfo method

OleAuto

Opens the type library of an OLE Automation server.

Syntax

```
openTypeInfo ( var serverName String ) Logical
```

registerControl procedure

Description

openTypeInfo connects to the type library of the server specified by *serverName*. Once connected, you can call the type enumeration methods to retrieve information about the server. The **openTypeInfo** method creates an instance of the server and gives you access to the server methods and properties. If a server doesn't provide a type library, this method will return False.

This method is designed for type browsing only.

Example

The following example connects to the Paradox type library and then builds and displays the dynamic array (DynArray) of Paradox's properties:

```
method pushButton ( var eventInfo Event )
var
    oa oleauto
    dy DynArray[] String
endvar
    oa.openTypeInfo("Paradox.application")
    oa.enumProperties(dy)
    dy.view()
endMethod
```

registerControl procedure

OleAuto

Registers an OLE Automation control.

Syntax

```
registerControl ( const fileName String ) String
```

Description

registerControl auto-registers the OLE Automation control specified in *fileName*.

Example

The following example registers the MyCntl.cntl1 control. The control's registered name is the complete pathname of the file containing the control.

```
method pushButton ( var eventInfo Event )
registerControl("C:\\OCXLIB\\MYCNTL1.OCX")
endMethod
```

unregisterControl procedure

OleAuto

Unregisters an ActiveX control.

Syntax

```
unregisterControl ( const fileName String ) Logical
```

Description

unregisterControl unregisters an ActiveX control. The argument *fileName* specifies the name of the ActiveX control you want to unregister. This procedure returns True if the file is a valid ActiveX control; otherwise, it returns False. The ActiveX control must support the ability to unregister itself.

Example

See the **registerControl** example.

version method**OleAuto**

Returns the version number of the current OLE2 server.

Syntax

```
version ( ) String
```

Description

version returns a string containing the version number of the currently attached OLE2 server (e.g., "2.0").

Example

The following example opens the Paradox OLE Automation server and retrieves its version number.

```
method pushButton ( var eventInfo Event )  
var  
    oa oleauto  
    v string  
endvar  
    oa.open("Paradox.Application")  
    v = oa.version()  
endMethod
```

Point type

A Point variable contains information about a point on the screen. ObjectPAL considers the screen to be a two-dimensional grid, with the origin at the upper-left corner of the design object's container, the positive x values extending to the right, and the 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 a logical inch, and 1/20 of a printer's point. ObjectPAL converts Point values to range from -2,147,483,648 to 2,147,483,647.

Methods defined for the Point type get and set information about screen coordinates and relative point positions. For example, a design object's size and position properties are specified in points.

ObjectPAL calculates point values relative to the container of the specified design object. This means that if a box contains a button, ObjectPAL calculates the button's position relative to the box. Similarly, 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. You can use **convertPointWithRespectTo** defined for the UIObject type to convert values in different frameworks.

You can use Point operators (+, -, =, <, >, <=, and >=) to add, subtract, and compare Point variables. As the following example illustrates, these operators affect the x coordinates of each point first and then the y coordinates.

```
var
    p1, p2, p3 Point
endVar

p1 = Point(10, 30)
p2 = Point(10, 30)
p3 = Point(10, 33)

message(p1 + p2) ; Displays (20, 60), because 10 + 10 = 20, and 30 + 30 = 60.
message(p1 = p2) ; Displays True. Both x and y coordinates are equal.
message(p1 = p3) ; Displays False. Both coordinates must be equal.
message(p3 > p1) ; Displays False. Both coordinates must be greater.
message(p3 >= p1) ; Displays True. Both coordinates are either greater or equal.
```

The following table displays the methods for the Point type, including the derived methods from the AnyType type.

Methods for the Point type

AnyType	←	Point
blank		distance
dataType		isAbove
isAssigned		isBelow
isBlank		isLeft
isFixedType		isRight
view		point
		setX
		setXY
		setY
		x
		y

distance method

Point

Returns the distance between two points, measured in twips.

Syntax

```
distance ( const pt Point ) Number
```

Description

distance returns the number of twips between a specified point and *pt*.

Example

The following example assumes a form contains 2 boxes: *redBox* and *brownBox*. The **pushButton** method for a button named *getDistance* determines 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
```

isAbove method

Point

Reports whether a point is positioned above another point.

Syntax

```
isAbove ( const pt Point ) Logical
```

Description

isAbove returns True if the y coordinate of a point is less than the y coordinate of *pt*; otherwise, it returns False.

Example

The following example uses the **pushButton** method for *convergeBoxes* to move *boxOne* closer to *boxTwo*, until the two boxes converge. Assume that *boxOne* is originally positioned above and left of *boxTwo*. Each time the button is clicked, *boxOne* moves down until it is on the same vertical plane and then moves 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())
```

isBelow method

```
    endIf  
endIf  
endMethod
```

isBelow method

Point

Reports whether a point is positioned below another point.

Syntax

```
isBelow ( const pt Point ) Logical
```

Description

isBelow returns True if the y coordinate of a point is greater than the y coordinate of *pt*; otherwise, it returns False.

Example

The following example uses the **pushButton** method for *convergeBoxes* to move *boxTwo* closer to *boxOne*, until the two boxes converge. Assume that *boxTwo* is originally positioned below and to the right of *boxOne*. Each time the button is clicked, *boxTwo* moves up until it is on the same vertical plane and then moves 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  
endIf  
endMethod
```

isLeft method

Point

Reports whether a point is positioned to the left of another point.

Syntax

```
isLeft ( const pt Point ) Logical
```

Description

isLeft returns True if the x coordinate of a point is less than the x coordinate of *pt*; otherwise, it returns False.

Example

See the **isAbove** example.

isRight method

Point

Reports whether a point is positioned to the right of another point.

Syntax

```
isRight ( const pt Point ) Logical
```

Description

isRight returns True if the x coordinate of a point is greater than the x coordinate of *pt*; otherwise, it returns False.

Example

See the **isBelow** example.

point procedure**Point**

Casts an expression as a Point.

Syntax

1. `point (const x LongInt, const y LongInt) Point`
2. `point (const newPoint Point) Point`

Description

point casts an expression as a Point.

Example

The following example varies the position of a box called *rateBox*. The values of an unbound field object named *rateField* range from 0 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
```

setX method

Point

Sets the x coordinate of a point.

Syntax

```
setX ( const newXValue LongInt )
```

Description

setX sets the x coordinate of a point to *newXValue*. If *newXValue* is not a LongInt, it is converted to a LongInt. This conversion may result in a loss of precision.

Example

In the following example, a form contains an ellipse named *circleOne* and a button named *moveRight*. The **pushButton** method for *moveRight* uses **setX** to change the horizontal coordinate of a point and 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
```

setXY method

Point

Sets the x and y coordinates of a point.

Syntax

```
setXY ( const newXValue LongInt, const newYValue LongInt )
```

Description

setXY 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. This conversion may result in a loss of precision.

Example

In the following 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 and 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
```

setY method**Point**

Sets the y coordinate of a point.

Syntax

```
setY ( const newYValue LongInt )
```

Description

setY 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 the following 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 and 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
```

x method**Point**

Returns the x coordinate of a point.

Syntax

```
x ( ) LongInt
```

Description

x returns the x coordinate of a point.

Example

See the **setX** example.

y method**Point**

Returns the y coordinate of a point.

Syntax

```
y ( ) LongInt
```

Description

y returns the y coordinate of a point.

Example

See the **setY** example.

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.

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 a selected item
- inspect the items in a pop-up menu
- provide keyboard access

The following table displays the methods for the PopUpMenu type, including several derived methods from the Menu type.

Methods for the PopUpMenu type		
Menu	←	PopUpMenu
contains		addArray
count		addBar
empty		addBreak
isAssigned		addPopUp
remove		addSeparator
removeMenu		addStaticText
		addText
		show
		switchMenu

addArray method

PopUpMenu

Appends elements of an array to a pop-up menu.

Syntax

```
addArray ( const items Array[ ] String )
```

Description

addArray adds *elements* from an array to a pop-up menu.

Example

In the following example, 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
  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"

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

```

addBar method

PopUpMenu

Adds a vertical bar to a pop-up menu.

Syntax

```
addBar ( )
```

Description

addBar adds a vertical bar to a pop-up menu. The **addBar** method creates a new column in the pop-up menu and inserts a vertical bar immediately before the new column. **addBar** is the vertical equivalent of **addSeparator**.

Example

The following example displays a pop-up menu with two columns of choices. The first two choices are displayed in the left column, and all 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

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

```

addBreak method**PopUpMenu**

Starts a new column in a pop-up menu.

Syntax

```
addBreak ( )
```

Description

addBreak starts a new column in a pop-up menu. The first item added after the call to **addBreak** is displayed at the top of the column to the right of the previous column, and subsequent items follow 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 example creates a pop-up menu with nine choices displayed 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
```

addPopUp method**PopUpMenu**

Adds a pop-up menu to the existing pop-up menu structure.

Syntax

```
addPopUp ( const menuName String, const cascadedPopup PopUpMenu )
```

Description

addPopUp adds *menuName* and *cascadedPopup* to the current pop-up menu structure, creating a cascading menu. *menuName* is displayed as an item in the original pop-up menu, and the first item in *cascadedPopup* appears next to it. Subsequent items in *cascadedPopup* are displayed in a column below the first item.

Example 1

The following 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 pop-up 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
```

Example 2

The following example uses **addPopUp** to create a cascading pop-up menu. This menu structure is not attached to a Menu Bar item, and the built-in **menuAction** method is not used. The code immediately following the call to show executes based on the user's selection.

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
```

addSeparator method

```
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
```

addSeparator method

PopUpMenu

Adds a horizontal bar to a pop-up menu.

Syntax

```
addSeparator ( )
```

Description

addSeparator adds a horizontal bar to separate item groups in a pop-up menu. **addSeparator** is used to group similar commands within a menu.

Example

The following example uses **addSeparator** to group pop-up menu commands. This 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

```

addStaticText method

PopUpMenu

Adds a static (unselectable) text string to a pop-up menu.

Syntax

```
addStaticText ( const item String )
```

Description

addStaticText adds a static (unselectable) text string to a pop-up menu. Static text is used as the title (first item) in a pop-up menu.

Example

In the following example, when the user right-clicks the field, a list of available payment types is displayed 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
  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"

                                ; display first item as static text
pl.addStaticText("Payment Method")
pl.addSeparator()              ; add a horizontal separator
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

```

addText method**PopUpMenu**

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

addText adds a selectable text string to a pop-up menu. The pop-up menu can be displayed alone, or as part of a menu in the Menu Bar.

Syntax 1 uses *menuName* to specify the string to add to the pop-up menu.

In Syntax 2, you can use *attrib* to preset the display attribute of *menuName*. ObjectPAL's MenuChoiceAttributes constants (e.g., MenuDisabled) for display attributes.

Syntax 3 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 use in the built-in **menuAction** method, you use the *id* number to determine which menu the user chooses.

You can also use Syntax 3 to create a menu that provides the same functions as a built-in Paradox menu. Use a MenuCommands constant to assign a value to the *id* argument. When the user chooses that item from a menu, Paradox performs the default action. For example, the following line adds Next to the *puRecord* PopUpMenu and uses the MenuCommands constant MenuRecordNext to assign an ID value.

```
puRecord.addText("Next", MenuEnabled, MenuRecordNext)
```

You must display, enable, or disable menu items to ensure that the Paradox operation that the user triggers is valid (e.g., you can only lock records in edit mode).

You can specify custom menu IDs, by adding a number or a user-defined menu constant to UserMenu. For example, the following code adds "File" to the *myPopup* PopUpMenu and specifies an *id* number for that menu item:

```
myPopup.addText("File", MenuEnabled, UserMenu + 1)
```

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 case, the returned value is &File, not File.

You can also use \t to insert a Tab between an item and its accelerator. For example, the item &Edit Data\tF9 displays Edit Data left-aligned and F9 right-aligned. In this case the string value returned is &Edit Data\tF9.

Example 1

The following example displays a variation of the **addText** syntax.

For this example, assume a form has an unbound field named *payField*. When the user right-clicks the field, a list of available payment methods is displayed in a pop-up menu. The user can choose use 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 prepares the pop-up menu for future display.

```
; 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 uses **show** to display the menu and 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
```

Example 2

The following example displays a variation of the **addText** syntax.

This example uses 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 is added to Const window for *thisPage*.

```
; thisPage::const
Const
    kMenuRed    = 1 ; define constant values for menu ids
    kMenuBlue   = 2
    kMenuWhite  = 3
    kMenuTime   = 4
    kMenuDate   = 5
    kMenuAbout  = 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 (e.g., MenuEnabled). To identify each menu item by number, the code uses the constants defined in the Const window for *thisPage* (*menuRed*, *menuBlue*, etc.).

```
; thisPage::open
method open(var eventInfo Event)
var
    mainMenu Menu
    subMenu1, clrMenu, puRecord PopUpMenu
endVar

; add text to pop-up menus and use user-defined constants
clrMenu.addText("&Red", MenuEnabled, kMenuRed + UserMenu)
clrMenu.addText("&Blue", MenuEnabled, kMenuBlue + UserMenu)
clrMenu.addText("&White", MenuEnabled, kMenuWhite + UserMenu)
```

isAssigned method

```
subMenu1.addText("&Time", MenuEnabled, kMenuTime + UserMenu)
subMenu1.addText("&Date", MenuEnabled, kMenuDate + UserMenu)
subMenu1.addSeparator()
subMenu1.addPopUp("&Page colors", clrMenu)
subMenu1.addSeparator()
subMenu1.addText("&About", MenuEnabled, kMenuAbout + UserMenu)
; Build a pop-up menu to attach to the Record menu.
; Use ObjectPAL MenuCommands constants to assign item IDs.
puRecord.addText("&First", MenuEnabled, MenuRecordFirst)
puRecord.addText("&Prev", MenuEnabled, MenuRecordPrevious)
puRecord.addText("&Next", MenuEnabled, MenuRecordNext)
puRecord.addText("&Last", MenuEnabled, MenuRecordLast)
; attach pop-up menus to mainMenu and display the Menu Bar
mainMenu.addPopUp("&Utilities", subMenu1)
mainMenu.addPopUp("&Record", puRecord)
mainMenu.show()
endMethod
```

The following code is attached to the **menuAction** method for *thisPage*. This example evaluates menu selections by ID number:

```
; thisPage::menuAction
method menuAction(var eventInfo MenuEvent)
var
  menuId SmallInt
endVar

menuId = eventInfo.id() ; store menu id number in menuId

switch
  case menuId = kMenuRed + UserMenu : self.color = Red
  case menuId = kMenuBlue + UserMenu : self.color = Blue
  case menuId = kMenuWhite + UserMenu : self.color = White
  case menuId = kMenuTime + UserMenu : msgInfo("Time", time())
  case menuId = kMenuDate + UserMenu : msgInfo("Date", date())
  case menuId = kMenuAbout + UserMenu : eventInfo.setId(MenuHelpAbout)

  ; No extra code is needed to handle choices from the Record menu,
  ; because item IDs were assigned using MenuCommands constants.
  ; Paradox handles them automatically.
endSwitch

endMethod
```

isAssigned method

PopUpMenu

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just PopUpMenu.

Example

The following 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 is attached 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
```

show method

PopUpMenu

Displays a pop-up menu and returns the selected item.

Syntax

```
show ( [ const xTwips SmallInt, const yTwips SmallInt ] ) String
```

Description

show displays a pop-up menu and returns the selected item. If the user presses ESC without making a selection, the returned value is a zero-length string. The optional arguments *xTwips* and *yTwips* specify the coordinates of the upper left corner of the pop-up menu. If not specified, these arguments 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 is displayed 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 is added to 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 prepares the menu for future display.

```
; payField::open
method open(var eventInfo Event)

payPopUp.addText("Visa")
payPopUp.addText("MasterCard")
payPopUp.addText("Check")
payPopUp.addText("Cash")

endMethod
```

switchMenu procedure

The following code is attached to *payField*'s built-in **mouseRightUp** method. When the user right-clicks the field, this method uses **show** to display the menu and 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 into unbound field
endif
endMethod
```

switchMenu procedure

PopUpMenu

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

switchMenu 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 an action if the user closes the menu without making a choice (e.g., by pressing ESC).

Example

The following example uses **switchMenu** to create, display, and process a choice from a pop-up menu. A string describing the selection is displayed in 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
```

Query type

An ObjectPAL Query variable is a query by example (QBE). You can use ObjectPAL to create and execute queries from methods in the same way that you use Paradox interactively. You can execute a query from a query file, a query statement, or a string. Some queries require Paradox to create temporary tables in your private directory.

Methods for the Query type

appendRow	getRowID	readFromFile
appendTable	getRowNo	readFromString
checkField	getRowOp	removeCriteria
checkRow	getTableID	removeRow
clearCheck	getTableNo	removeTable
createAuxTables	hasCriteri	asetAnswerFieldOrder
createQBEStrng	insertRow	setAnswerName
enumFieldStruct	insertTable	setAnswerSortOrder
executeQBE	isAssigned	setCriteria
getAnswerFieldOrder	isCreateAuxTables	setLanguageDriver
getAnswerName	isEmpty	setQueryRestartOptions
getAnswerSortOrder	isExecuteQBELocal	setRowOp
getCheck	isQueryValid	wantInMemoryTCursor
getCriteria	query	writeQBE
getQueryRestartOptions		

appendRow method

Query

Appends a row to a query table image.

Syntax

```
appendRow ( const tableID SmallInt ) SmallInt
appendRow ( const tableName String ) SmallInt
```

Description

appendRow adds a new row to the specified table image in a query by example (QBE). The table is specified numerically by *tableID* or by *tableName*. **appendRow** returns the numeric *rowID* of the new row which is used to manipulate the row's contents. Even if rows are inserted or deleted ahead of the appended row, the *RowID* doesn't change.

Example

The following example appends a row to the query for the CUSTOMER.DB table:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    rowID SmallInt
endVar

qVar.appendTable( "CUSTOMER.DB" )
rowID = qVar.appendRow( "CUSTOMER.DB" )
qVar.setCriteria( "CUSTOMER.DB", rowID, "State/Prov", "CA or HI" )
qvar.checkRow("Customer.db", rowID, CheckCheck)
qvar.writeQBE("MyQBE")
endMethod
```

appendTable method

Query

Appends a table to a query image.

checkField method

Syntax

```
appendTable ( const tableName String ) SmallInt
```

Description

appendTable adds the table specified by *tableName* to the query image in a query by example (QBE) and returns a numeric ID which can be used to manipulate the table image's contents. Even if table images are inserted or deleted ahead of the appended table, its ID doesn't change.

Example

The following example appends a table to a query image:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    rowID SmallInt
    tblID SmallInt
endVar

tblID = qVar.appendTable( "CUSTOMER.DB" )
rowID = qVar.appendRow( tblID )
qVar.setCriteria( tblID , rowID, "State/Prov", "CA or HI" )
qVar.checkRow("Customer.db", rowID, CheckCheck)
qVar.writeQBE("MyQBE")
endMethod
```

checkField method

Query

Creates a check mark in a specified field of a query table image.

Syntax

```
checkField ( const tableID SmallInt, const fieldID SmallInt, const checkType SmallInt ) Logical
checkField ( const tableID SmallInt, const fieldID SmallInt, const rowID SmallInt, const checkType SmallInt ) Logical
checkField ( const tableID SmallInt, const fieldName String, const checkType SmallInt ) Logical
checkField ( const tableID SmallInt, const fieldName String, const rowID SmallInt, const checkType SmallInt ) Logical
checkField ( const tableName String, const fieldID SmallInt, const checkType SmallInt ) Logical
checkField ( const tableName String, const fieldID SmallInt, const rowID SmallInt, const checkType SmallInt ) Logical
checkField ( const tableName String, const fieldName String, const checkType SmallInt ) Logical
checkField ( const tableName String, const fieldName String, const rowID SmallInt, const checkType SmallInt ) Logical
```

Description

checkField creates a check mark in a specified field. The table is specified numerically by *tableID* or by *tableName*. The field is specified by *fieldID* or *fieldName*. The corresponding row is specified by the row identifier *rowID*. If no row is specified, the **checkField** method defaults to the first row.

The **checkType** is one of the following qbeCheckType constants:

CheckCheck	Check mark (unique keys only)
CheckDesc	Descending order check
CheckGroup	GroupBy check
CheckNone	Invisible check
CheckPlus	Plus sign (include duplicate keys)

checkRow method

Example

The following example checks the State/Prov field in the CUSTOMER.DB table of the specified query image:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    rowID SmallInt
    tblID SmallInt
    MyQBValidateStr String
endVar

tblID = qVar.appendTable( "CUSTOMER.DB" )
rowID = qVar.appendRow( tblID )
qVar.setCriteria( tblID , rowID, "State/Prov", "CA or HI" )
qVar.checkField( tblID, rowID, "State/Prov", CheckPlus )
MyQBValidateStr = qVar.createQBEStr()
MyQBValidateStr.view()
endMethod
```

checkRow method

Query

Creates a check mark in each field of a specified row of a query table image.

Syntax

```
checkRow ( const tableName String, const rowID SmallInt, const checkType SmallInt )
Logical
```

```
checkRow ( const tableName String, const checkType SmallInt ) Logical
```

Description

checkRow creates a check mark in each field of a specified row of a table image. The table is specified numerically by *tableID* or by *tableName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

The **checkType** is one of the following qbeCheckType constants:

CheckCheck	Check mark (unique keys only)
CheckDesc	Descending order check
CheckGroup	GroupBy check
CheckNone	Invisible check
CheckPlus	Plus sign (include duplicate keys)

Example

The following example puts the CheckPlus symbol in every field in first row of the CUSTOMER.DB query table image and saves the query as ALLCust.QBE:

```
method pushButton(var eventInfo Event)
var
    qVar Query
endVar

qVar.appendTable( "Customer.db" )
qVar.checkRow( "Customer.db", CheckPlus ) ; row not specified,
                                           ; use first row

qVar.writeQBE("ALLCust.QBE")
endMethod
```

clearCheck method

Query

Deletes a check mark from a specified field or row of a query table image.

Syntax

```

clearCheck ( const tableID SmallInt, const fieldID SmallInt ) Logical
clearCheck ( const tableID SmallInt, const fieldName String ) Logical
clearCheck ( const tableID SmallInt, const rowID SmallInt, const fieldID SmallInt )
Logical
clearCheck ( const tableID SmallInt, const rowID SmallInt, const fieldName String )
Logical
clearCheck ( const tableName String, const fieldID SmallInt ) Logical
clearCheck ( const tableName String, const fieldName String ) Logical
clearCheck ( const tableName String, const rowID SmallInt, const fieldID SmallInt )
Logical
clearCheck ( const tableName String, const rowID SmallInt, const fieldName String )
Logical

```

Description

clearCheck removes a check mark from a specified field in the query by example (QBE). The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

Example

The following example removes the check mark from the State/Prov field in the CUSTOMER.DB query table image and then runs the query:

```

method pushButton(var eventInfo Event)
var
    qVar Query
endVar

    qVar.readFromFile( "monthly.qbe" )
    qVar.clearCheck( "Customer.db" , "State/Prov" )
    qVar.executeQBE()
endMethod

```

createAuxTables method

Query

Enables the use of auxiliary tables.

Syntax

createAuxTables (const *useAuxTables* Logical) Logical

Description

createAuxTables enables the use of auxiliary tables if *useAuxTables* is set to True

Example

The following example contains a query that uses auxiliary tables:

```

method pushButton(var eventInfo Event)var
myQBE Query
endvar

myQBE = Query

    Customer.db | Name |

```

```

Delete      | Johnson.. |
endQuery

myQBE.createAuxTables(True)
myQBE.executeQBE()
endMethod

```

createQBEStr method

Query

Returns the QBE string of a query.

Syntax

```
createQBEStr ( ) String
```

Description

createQBEStr returns the QBE string of a query variable. If the query by example (QBE) is invalid, **createQBEStr** returns a blank string and `errorCode()` determines the cause of the failure. The QBE **must** be a valid query against existing tables in order for this function to return a query string, so it should not be used to generate partial (incomplete) query strings or queries which will generate syntax errors if compiled or executed.

Example

The following example displays a QBE string from a modified version of the MONTHLY.QBE query.

```

method pushButton(var eventInfo Event)
var
    qVar Query
    qStr String
endVar

qVar.readFile( "Monthly.qbe" )
qVar.clearCheck( "Customer.db" , "State/Prov" )
qVar.checkField( "Customer.db" , "Name", CheckPlus )
qStr = qVar.createQBEStr()
if isblank( qStr ) then
    errorShow()
else
    qStr.view( "Query String" )
endif

endMethod

```

enumFieldStruct method

Query

Lists the field structure of an answer table.

Syntax

1. `enumFieldStruct (const tableName String)` Logical
2. `enumFieldStruct (var inMemoryTC TCursor)` Logical

Description

enumFieldStruct lists the field structure of the answer table that is generated from the query by example (QBE) statement. Syntax 1 creates a Paradox table, and Syntax 2 stores the information in a TCursor variable. **enumFieldStruct** returns True if successful; otherwise, it returns False.

executeQBE method/procedure

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* exists, **enumFieldStruct** overwrites it without confirmation. You can include an alias or path in *tableName* but if no alias or path exists, Paradox creates *tableName* in the working directory.

Syntax 2 stores the information in the TCursor variable named *inMemoryTC*. You pass *inMemoryTC* as an argument.

The following table describes the structure of the table (Syntax 1) or TCursor (Syntax 2):

Field	Type	Description
Field Name	A31	Name of field
Type	A31	Data type of field
Size	S	Size of field
Dec	S	Number of decimal places in the field (0 if field type doesn't support decimal places)
Key	AI	* = key field, blank = not key field
_Required Value	AI	T = required field, N (or blank) = Not required
_Min Value	A255	Minimum value, if specified; otherwise blank
_Max Value	A255	Maximum value, if specified; otherwise blank
_Default Value	A255	Default value, if specified; otherwise blank
_Picture Value	A175	Picture, if specified; otherwise blank
_Table Lookup	A255	Name of lookup table; including full path if the lookup table is not in :WORK:
_Table Lookup Type	AI	Type of lookup table. 0 (or blank) = no lookup table, 1 = Paradox
_Invariant Field ID	S	Ordinal position of field in the table (first field = 1, second field = 2, etc.)

Example

The following example creates the Paradox table MYANSWER.DB containing the structure of the answer table that is built by the query MYQUERY.QBE:

```
method pushButton(var eventInfo Event)
var
    qVar      Query
endVar
    qVar.readFile( "myquery.qbe" )
    qVar.enumFieldStruct("QSTRUCT.DB")
endMethod
```

executeQBE method/procedure

Query

Executes a query by example (QBE).

Syntax

1. (Method) **executeQBE** ([{const *ansTbl* String | var *ansTbl* Table | var *ansTbl* TCursor}]) Logical
2. (Procedure) **executeQBE** (var *db* Database, var *qVar* Query [{const *ansTbl* String | var *ansTbl* Table | var *ansTbl* TCursor}]) Logical

Description

executeQBE executes the query assigned to a Query variable and writes the results to :PRIV:ANSWER.DB or to the table specified in *ansTbl*. You can assign a query to a Query variable using a query statement. Create a **query** statement by calling **readFromFile** or **readfromString** or by building it with **appendTable**, **appendRow**, or **setCriteria**.

Syntax 1 calls **executeQBE** as a method. You can write the query result to *ansTbl* where *ansTbl* is a table name, a Table variable, or a TCursor. If *ansTbl* is not specified, **executeQBE** writes the results to ANSWER.DB in the private directory.

Syntax 2 calls **executeQBE** as a procedure. Specify a Database variable in *db* and a Query variable in *qVar*. You can write the query result to *ansTbl* where *ansTbl* is a table name, a Table variable, or a TCursor. If *ansTbl* is not specified, **executeQBE** writes the results to ANSWER.DB in the private directory.

The following notes apply to both syntaxes:

- If you specify the table name as a string and don't include a file extension, *ansTbl* defaults to specify a Paradox.
- If you specify *ansTbl* as a Table variable, *ansTbl* must be assigned and valid.
- If you specify *ansTbl* as a TCursor, the results are stored in memory only.
- If **executeQBE** is successful (*ansTbl* or ANSWER.DB is created) this method returns True; otherwise it returns False. **executeQBE** returns True even if the resulting table is empty.

Example 1

The following example calls **executeQBE** as a method. The **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable and then runs it with **executeQBE**. The query statement in this example is an insert query: it retrieves 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 includes Oregon customers in the results. Because OR is an ObjectPAL keyword, the *myState* variable must evaluate to a quoted string to distinguish it from the abbreviation for Oregon.

```
method pushButton(var eventInfo Event)
var
    qVar      Query
    myState   String
    tv        TableView
endVar

; add samp alias for the sample directory
addAlias("samp", "Standard", "c:\\Program Files\\Corel\\Shared\\samples")

; OR is the abbreviation for Oregon, but because it's
; also an ObjectPAL keyword, it must be enclosed in quotes.
myState = "\"OR\""

; now use myState as a tilde variable in this query statement
qVar = Query
```

executeQBE method/procedure

```
: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

qVar.executeQBE("myCust.db") ; put results into myCust.db
tv.open("myCust.db")        ; view the table

endMethod
```

Example 2

The following example calls **executeQBE** as a procedure. The **pushButton** method for the *getReceivables* button constructs a query statement, assigns it to a Query variable and then runs it with **executeQBE**. The query statement in this example is an insert query: it retrieves 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 includes Oregon customers in the results. Because OR is an ObjectPAL keyword, the *myState* variable must evaluate to a quoted string to distinguish it from the abbreviation for Oregon.

```
method pushButton(var eventInfo Event)
var
    db      Database
    qVar     Query
    myState  String
    tv      TableView
endVar

db.open() ; Get a handle to the default database.

; add samp alias for the sample directory
addAlias("samp", "Standard", "c:\\Corel\\Paradox\\samples")

; OR is the abbreviation for Oregon, but because it is also
; a Paradox keyword 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(db, qVar, "myCust.db") ; put results into myCust.db
tv.open("myCust.db")             ; view the table

endMethod
```

getAnswerFieldOrder method**Query**

Retrieves the field names of the custom field order in the answer table generated by a query.

Syntax

```
getAnswerFieldOrder ( var fieldOrder Array[] String ) Logical
```

Description

getAnswerFieldOrder retrieves an array of the fields in the answer table for the current query, when a custom field order is specified. If a custom field order is not specified for the query, this method returns an empty array. If the query compiles successfully, the array *fieldOrder* is filled with the field names in the answer table. These names can be rearranged and the array can be submitted to the **setAnswerFieldOrder** and **setAnswerSortOrder** methods. The array must be resizable.

Example

The following example retrieves the existing field order specified in MYQUERY.QBE, reorders the fields, and then uses **setAnswerFieldOrder** to put the new order in place.

```
method pushButton(var eventInfo Event)
var
    qVar Query
    arFields Array[] String
endVar
qVar.readFromFile( "myquery.qbe" )
qVar.getAnswerFieldOrder( arFields )
    if arFields.size() = 0 then      ; swap the first and third fields
                                ; in the answer table.
        arFields.exchange(1,3)
        qVar.setAnswerFieldOrder( arFields )
        qVar.executeQBE()
    endif
endMethod
```

getAnswerName method**Query**

Retrieves the name of the answer table.

Syntax

```
getAnswerName ( ) String
```

Description

getAnswerName retrieves the name of the answer table that is produced by the query.

Example

The following example allows the user to change the answer table name for MYQUERY.QBE:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    AnsTblName String
endVar
if msgQuestion("Query",
    "Would you like to change the "
    + "answer table name?") = "Yes" then
    qVar.readFromFile("MYQUERY.QBE")
    AnsTblName = qVar.getAnswerName()
    AnsTblName.view("Make changes below")
    qVar.setAnswerName(AnsTblName)
    qVar.writeQBE("MYQUERY.QBE")
endif
endMethod
```

getAnswerSortOrder method

```
endif  
endMethod
```

getAnswerSortOrder method

Query

Retrieves the custom sort order specified for the answer table.

Syntax

```
getAnswerSortOrder ( var sortFields Array[] String ) Logical
```

Description

getAnswerSortOrder retrieves the custom sort order specified for the answer table. If a custom sort order is not specified, this method returns an empty array. The array *sortFields* contains an ordered list of field names. After you retrieve an array of these field names using **getAnswerSortOrder**, you can change the sort order.

If you retrieve the list of fields and then change the answer field list (e.g., by unchecking a field), the array is instantly outdated. You must remove the modified field from your array before attempting to use this array for field sorting.

Example

The following example retrieves the field list from MYQUERY.QBE, reorders the fields, and saves the new sort order back into the query using **setAnswerSortOrder**:

```
method pushButton(var eventInfo Event)  
var  
    qVar Query  
    arFields Array[] String  
endVar  
qVar.readFromFile( "myquery.qbe" )  
qVar.getAnswerSortOrder( arFields )  
    if arFields.size() 0 then      ; swap the first and third fields  
                                   ; in the sort order.  
        arFields.exchange(1,3)  
        qVar.setAnswerSortOrder( arFields )  
        qVar.executeQBE()  
    endif  
endMethod
```

getCheck method

Query

Returns the check type for a specified field in a query image.

Syntax

```
getCheck ( const tableID SmallInt, const fieldID SmallInt ) SmallInt  
getCheck ( const tableID SmallInt, const fieldName String ) SmallInt  
getCheck ( const tableID SmallInt, rowID SmallInt, const fieldID SmallInt ) SmallInt  
getCheck ( const tableID SmallInt, rowID SmallInt, const fieldName String ) SmallInt  
getCheck ( const tableName String, const fieldID SmallInt ) SmallInt  
getCheck ( const tableName String, const fieldName String ) SmallInt  
getCheck ( const tableName String, rowID SmallInt, const fieldID SmallInt ) SmallInt  
getCheck ( const tableName String, rowID SmallInt, const fieldName String ) SmallInt
```

Description

getCheck returns the check type for a specified field in a query image. The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

The checkType is one of the following qbeCheckType constants:

CheckCheck	Check mark (unique keys only)
CheckDesc	Descending order check
CheckGroup	GroupBy check
CheckNone	Invisible check
CheckPlus	Plus sign (include duplicate keys)

Example

The following example changes the type of check used in the State/Prov field of the CUSTOMER.DB query table image from CheckPlus to CheckDesc.

```
method pushButton(var eventInfo Event)
var
    qVar Query
    qStr String
endVar
qVar.readFromFile( "monthly.qbe" )
if qVar.getCheck( "Customer.db" , "State/Prov" ) = CheckPlus then
    qVar.CheckField( "Customer.db" , "State/Prov" , CheckDesc )
    qVar.writeQBE("Monthly.QBE")
endif
endMethod
```

getCriteria method

Query

Returns the query expression used in a query image.

Syntax

```
getCriteria ( const tableID SmallInt, const fieldID SmallInt ) String
getCriteria ( const tableID SmallInt, const fieldName String ) String
getCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID SmallInt )
String
getCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName String )
String
getCriteria ( const tableName String, const fieldID SmallInt ) String
getCriteria ( const tableName String, const fieldName String ) String
getCriteria ( const tableName String, const rowID SmallInt, const fieldID SmallInt )
String
getCriteria ( const tableName String, const rowID SmallInt, const fieldName String )
String
```

Description

getCriteria returns the selection conditions and calculation statements in a specified field of a query image. The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

This expression does not include the check mark, but does contain the remaining field contents.

Example

The following example changes the selection conditions for the Name field in the CUSTOMER.DB query table image:

```
method pushButton(var eventInfo Event)
var
```

getQueryRestartOptions method

```
qVar Query
NameCriteria String
endVar
qVar.readFile( "monthly.qbe" )
NameCriteria = qVar.getCriteria ( "Customer.db" , "Name")
; default to the first row
NameCriteria = NameCriteria + " or Unisco"
qVar.setCriteria( "Customer.db" , "Name" , NameCriteria )

endMethod
```

getQueryRestartOptions method

Query

Returns a value representing the user's query restart setting.

Syntax

```
getQueryRestartOptions ( ) SmallInt
```

Description

getQueryRestartOptions returns an integer value representing the user's query restart setting. Use one of the following ObjectPAL QueryRestartOptions constants to test the value:

QueryDefault	Use the options specified interactively by using the Query Restart Options dialog box.
QueryLock	Lock other users out of the tables needed to run the query. If Paradox cannot lock a table, it does not run the query.
QueryNoLock	Continue to run the query if a change is made to the data during its execution.
QueryRestart	Restart the query. Specify QueryRestart to get a snapshot of the data as it existed at a particular instant.

Example

See the setQueryRestartOptions example.

getRowID method

Query

Returns the row identifier for a specified sequence row number.

Syntax

```
getRowID ( const tableID SmallInt, const seqNo SmallInt ) SmallInt
```

Description

getRowID returns the *rowID* for the specified sequence. The *rowID* is any number, regardless of where the row resides in the table image on the query workspace. The table is specified numerically by *tableID*.

To determine the *rowID* of a specified sequence you must convert the row's sequence number to the *rowID*. For example, the second row of the Customer.db table image might have a *rowID* of 32760.

Example

The following example returns the row identifier of the second row in CUSTOMER.DB, assigns the name secondRow, changes the criteria of the Country field, and runs the query:

```

method pushButton(var eventInfo Event)
var
    qVar Query
    secondRow SmallInt
endVar
qVar.readFromFile( "monthly.qbe" )
secondRow = qVar.getRowID( qVar.getTableID(1), 2 )
qVar.setCriteria( "Customer.db", secondRow, "Country", "Fiji" )
qVar.executeQBE()
endMethod

```

getRowNo method

Query

Returns the sequence number of a specific row.

Syntax

```
getRowNo ( const tableID SmallInt, const rowID SmallInt ) SmallInt
```

Description

getRowNo returns the sequence number of the row specified by *rowID*. The sequence number is the complement of **getRowID**. Given a unique numeric row identifier (*rowID*), **getRowNo** returns the current position (sequence) of the row in the query table image. For example, a *rowID* of 32760 might be the third row in a table image.

Example

The following example creates a Query variable and appends the CUSTOMER.DB table image to the query. The code then prints the new row's sequence number.

```

method pushButton(var eventInfo Event)
var
    qVar          Query
    seqNo,
    rowID,
    tableID       SmallInt
endVar
tableID=qVar.appendTable( "Customer.db" )
rowID = qVar.appendRow( "Customer.db" )
seqNo = qVar.getRowNo( tableID,rowID )
message( "The newly appended row is row number ",seqNo,
        " in the customer.db query image" )
endMethod

```

getRowOp method

Query

Retrieves the row operator set for a specified row.

Syntax

```

getRowOp ( const tableID SmallInt [, const rowID SmallInt] ) SmallInt
getRowOp ( const tableName String [, const rowID SmallInt] ) SmallInt

```

Description

getRowOp returns the row operator set for a specified row. The table is specified numerically by *tableID* or by *tableName*. The row can be specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

The *rowOperator* is one of the following values:

getTableID method

qbeRowDelete	Delete operator
qbeRowInsert	Insert operator
qbeRowNone	No operator
qbeRowSet	Set operator

Example

The following example deletes records containing blank Customer No fields:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    rowop SmallInt
endVar

qVar.readFromFile( "Sample.qbe" )
rowop = qVar.getRowOp( "Customer.db" )

endMethod
```

getTableID method

Query

Returns the tableID for a specified table in the query image.

Syntax

```
getTableID ( const seqNo SmallInt ) SmallInt
```

Description

getTableID returns the *tableID* for a specified table in the query image. This ID differs from the table's sequential number, and using the sequential number results in errors. You can replace the *tableID* with the table name in those methods that accept the table name as a valid entry.

Example

The following example retrieves the table ID for the third table and the row ID for the second row of the query MONTHLY.QBE. The code then uses these IDs to determine the criteria set in the Name field.

```
method pushButton(var eventInfo Event)
var
    qVar Query
    thirdTableID, secondrowID SmallInt
    condition String
endVar

qVar.readFromFile("MONTHLY.QBE")
thirdTableID = qVar.getTableID(3)
secondRowID = qVar.getRowID(thirdTableID, 2)
condition = qVar.getCriteria(thirdTableID, secondRowID, "Name")
msgInfo("Condition", "The criteria for the Name field in the "
    + "second row of the third table is " + condition)

endMethod
```

getTableNo method

Query

Returns the table number for a specified table.

Syntax

```
getTableNo ( const tableID SmallInt ) SmallInt
```


Description

getTableNo returns the table number for the table specified by *tableID*. Given a unique numeric *tableID*, **getTableNo** returns its current position in the query. For example, if a *tableID* of 32760 represents the second table on the query workspace, this method returns a value of 2.

Example

The following example displays a specified table's position in the query:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    qStr String
    seqNo, rowID, newTableID SmallInt
endVar
qVar.readFromFile( "monthly.qbe" )
newTableID = qVar.appendTable( "Vendors.db" )
seqNo = qVar.getTableNo( newTableID )
message( "The newly appended table is table number ",
        seqNo," in the query image" )
endMethod
```

hasCriteria method**Query**

Indicates whether a specific field contains query criteria:

Syntax

```
hasCriteria ( const tableID SmallInt, const fieldID SmallInt ) Logical
hasCriteria ( const tableID SmallInt, const fieldName String ) Logical
hasCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID SmallInt )
Logical
hasCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName String )
Logical
hasCriteria ( const tableName String, const fieldID SmallInt ) Logical
hasCriteria ( const tableName String, const fieldName String ) Logical
hasCriteria ( const tableName String, const rowID SmallInt, const fieldID SmallInt )
Logical
hasCriteria ( const tableName String, const rowID SmallInt, const fieldName String )
Logical
```

Description

hasCriteria returns a Logical value indicating whether a specified field contains query criteria. The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*, or omitted to default to the first row.

hasCriteria examines the field for a query expression but does **not** detect check marks. Use **getCheck** to determine whether a field is checked.

Example

The following example retrieves criteria from the Sale Date field in the table ORDERS.DB in the query and runs the query:

```
method pushButton(var eventInfo Event)
var
    qVar          Query
    newTableID    SmallInt
    DateCriteria  String
endVar
qVar.readFromFile( "monthly.qbe" )
```

insertRow method

```
if qVar.hasCriteria( "Orders.db" , "Sale Date" ) then
    DateCriteria = qVar.getCriteria( "Orders.db" , "Sale Date")
else
    DateCriteria = ""
endif
DateCriteria.view( "Enter Date Criteria" )
qVar.setCriteria( "Orders.db", "Sale Date", DateCriteria )
qVar.executeQBE()
endMethod
```

insertRow method

Query

Adds a new row above an existing row in the query.

Syntax

```
insertRow ( const tableID SmallInt, beforeRowID SmallInt ) SmallInt
```

```
insertRow ( const tableName String, beforeRowID SmallInt ) SmallInt
```

Description

insertRow adds a new row above an existing row in the query. The table is specified numerically by *tableID* or by *tableName*. The parameter *beforeRowID* specifies the ID of the row which should be pushed down by the new row.

insertRow returns a SmallInt representing the row ID of the new row.

Example

The following example creates a query, based on the CUSTOMER.DB table, that retrieves customer records for two cities. After one row is appended and its query criteria set, another row is inserted and its criteria is set.

```
method pushButton(var eventInfo Event)
var
    qVar          Query
    firstRow, secondRow SmallInt
endVar

qVar.appendTable( "CUSTOMER.DB" )
secondRow = qVar.appendRow( "CUSTOMER.DB " )
qVar.checkRow( "CUSTOMER.DB" , CheckCheck )
qVar.setCriteria( "CUSTOMER.DB", "City", "Waterville")
qVar.setCriteria( "CUSTOMER.DB", "Country", "USA")
firstRow = qVar.insertRow( "CUSTOMER.DB", secondRow)
qVar.checkRow( "CUSTOMER.DB", CheckCheck)
qVar.setCriteria( "CUSTOMER.DB", "City", "Vancouver")
qVar.setCriteria( "CUSTOMER.DB", "Country", "Canada")
qVar.writeQBE( "TwoCity.QBE" )
endMethod
```

insertTable method

Query

Inserts a new table above an existing table in the query.

Syntax

```
insertTable ( const tableName String, const beforeTableID SmallInt ) SmallInt
```

```
insertTable ( const tableName String, const beforeTableName String ) SmallInt
```

Description

insertTable inserts a new table above an existing table in the query and returns the *tableID* for the new table. The parameter *tableName* specifies the name of the table to insert. The parameters *beforeTableID* and *beforeTableName* specify the ID and name (respectively) of the table that follows the new table.

Example

The following example creates a query that includes the Customer and Orders tables. The two tables are linked with an example element on their common field, Customer No, and all fields are checked in the Customer table. The query produces an answer table that lists all customer records containing order records.

```
method pushButton(var eventInfo Event)
var
    qVar Query
endVar
qVar.appendTable("CUSTOMER.DB")
qVar.checkRow("CUSTOMER.DB", CheckCheck)
qVar.setCriteria("CUSTOMER.DB", "Customer No", "_Join1")
qVar.insertTable("ORDERS.DB", "CUSTOMER.DB")
qVar.setCriteria("CUSTOMER.DB", "Customer No", "_Join1")
qVar.executeQBE()
endMethod
```

isAssigned method**Query**

Reports whether a Query variable has an assigned value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned 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

The following example uses **isAssigned** to determine if *qVar* had been assigned a value. Although the value is not a valid query, **isAssigned** returns True.

```
method pushButton(var eventInfo Event)
var
    qVar Query
endVar

qVar = Query

    This is not a query

endQuery

msgInfo("Assigned?", qVar.isAssigned())    ; displays True

endMethod
```

isCreateAuxTables method**Query**

Reports whether the use of auxiliary tables is enabled.

isEmpty method

Syntax

```
isCreateAuxTables ( ) Logical
```

Description

isCreateAuxTables reports whether the use of auxiliary tables is enabled. If **isCreateAuxTables** returns True, auxiliary tables are used to create a query's answer table.

Example

The following example contains a query that uses auxiliary tables:

```
method pushButton(var eventInfo Event)
var
myQBE Query
endvar

myQBE = Query

      Customer.db | Name      |
      Delete     | Johnson.. |

EndQuery

if myQBE.isCreateAuxTables() = False then
  myQBE.createAuxTables(True)
else
endif
myQBE.executeQBE()
endMethod
```

isEmpty method

Query

Determines whether the query is empty.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns a Logical value indicating whether the query is empty. This method determines whether you have added anything to your query but does not determine if the query contains enough information to be run. For example, you can append an empty row to a query and use **isEmpty** to determine if the query is empty. **isEmpty** returns False because you have appended pieces of the query. **isQueryValid** also returns False, because you did not complete the query.

Example

The following example reports if the Query variable is empty, before and after a query by example (QBE) file is read into the Query variable. The Query variable is empty before it has been assigned a value. If the **readFromFile** method is successful, the Query variable contains data.

```
method pushButton(var eventInfo Event)
var
  qVar Query
endvar

msgInfo( "Before readFromFile", "Query is " +
  iif(qVar.isEmpty(), "empty", "not empty"))
qVar.readFromFile("MyQuery.QBE")
msgInfo( "After readFromFile", "Query is " +
  iif(qVar.isEmpty(), "empty", "not empty"))
endMethod
```

isExecuteQBELocal method

Query

Reports whether a query by example (QBE) was executed locally or on a server.

Syntax

```
isExecuteQBELocal ( ) Logical
```

Description

isExecuteQBELocal returns True if the query was executed locally; otherwise, it returns False. This method is especially useful when the server uses a different character set, sort order, or other feature that affects the query's result.

Example

The following example calls **isExecuteQBELocal** to determine whether a query by example (QBE) was executed locally or on a server.

```
method pushButton (var eventInfo Event)
  var
    qbeVar      Query
    dlgTitleText,
    dlgBodyText String
  endVar

  dlgTitleText = "Remote query"
  dlgBodyText  = "This query was not run on the server. \n" +
                "Check the sort order"

  qbeVar = Query

      :WestData:orders.db |CustName|Qty      |
                        |Check  |Check 10 |

  endQuery

  if qbeVar.executeQBE() then
    if qbeVar.isExecuteQBELocal() then
      msgInfo(dlgTitleText, dlgBodyText)
    endIf
  else
    errorShow()
  endIf
endMethod
```

isQueryValid method

Query

Compiles the current query and indicates whether it contains errors that prevent it from being run.

Syntax

```
isQueryValid ( ) Logical
```

Description

isQueryValid compiles the current query and indicates whether it contains errors that prevent it from being run. This is the same procedure that occurs when you interactively save a query to disk, execute a query, or request a query string. **isQueryValid** returns False if the query contains an error. To get information on the error, use **errorCode** (System type).

query keyword

Example

The following example creates a query and reports an error if **isQueryValid** returns False:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    orderID SmallInt
endVar
orderID = qVar.appendTable( "Orders.db" )
qVar.setCriteria( orderID, "Sale Date", " 1/1/95" )
if not qVar.isQueryValid() then
    errorShow() ; note that no fields are checked
endif
endMethod
```

query keyword

Query

Marks the beginning of a query statement.

Syntax

```
query
    tableName|fieldName|[ fieldName] *
        |criteria |[ criteria ] *
[ tableName|fieldName|[ fieldName] *
    |criteria |[ criteria ] * ] *
endQuery
```

Description

query marks the beginning of a query by example (QBE) statement, which assigns a query to a Query variable. A QBE statement extracts data from one or more tables according to the fields specified in *fieldName* and the selection criteria (*criteria* can be any valid QBE expression). Because this type of query is not a string, it can contain tilde variables. For more information, see **readFromString**.)

A query statement contains a Query variable, the = sign, and the keyword query followed by a blank line. The body of the query is followed by another blank line, and the query ends with the keyword endQuery.

Note

- You don't have to list all the fields in the table. The following example lists only those fields that affect the query:

```
var
    myQBE Query
endvar
myQBE = Query
    Customer|Customer No|Name |
        |Check      |A.. |
endQuery
```

This query statement retrieves customer numbers whose names start with A from the Customer table. Only two of the Customer table's fields are specified.

You can align the vertical field separators to make the code more readable; however, ObjectPAL also recognizes the following code:

```
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. The following code example separates the Customer and Orders tables with a blank line:

```
var myQBE Query endvar
myQBE = Query
```

```
Customer|Customer No|Name |Phone |
|_x      |Check|Check |
```

```
Orders  |Customer No|Balance Due|
|_x      |Check 0    |
```

```
endQuery
```

You can use absolute paths or aliases to specify where to find tables in the query definition. Paradox also searches for unqualified table names (i.e., table names without paths or aliases) in a specified database. If a database is not specified, Paradox searches the default database (the working directory).

Example

The following example uses the **pushButton** method for the *getReceivables* button to construct a query statement, assign it to a Query variable and run it with **executeQBE**. In this example, the query statement is an insert query; it retrieves records from CUSTOMER.DB and ORDERS.DB and inserts them into the existing *MyCust* table. The selection criteria uses a tilde variable called *myState* that includes Oregon customers in the results. Since OR is the abbreviation for Oregon, the *myState* variable must evaluate to a quoted string to distinguish between the selection criteria and the **OR** query expression.

```
method pushButton(var eventInfo Event)
var
    qVar      Query
    myState   String
    tv        TableView
endVar

; add samp alias for the sample directory
addAlias("samp", "Standard", "c:\\Corel\\Paradox\\samples")

; OR is the abbreviation for Oregon, but because it's
; also an ObjectPAL keyword, 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 |
```

readFromFile method

```
EndQuery

qVar.executeQBE("myCust.db") ; put results into myCust.db
tv.open("myCust.db")       ; view the table

endMethod
```

readFromFile method

Query

Assigns the contents of a query by example (QBE) file to a Query variable.

Syntax

```
readFromFile ( const qbeFileName String ) Logical
```

Description

readFromFile opens *qbeFileName* and assigns the contents to a Query variable. There are several ways to create a query file; for example, in ObjectPAL using writeQBE, or interactively using the Query Editor. Use **executeQBE** to execute the query.

If the value of *qbeFileName* does not include a path or alias, **readFromFile** searches for the file in the directory associated with a specified database. If a database is not specified, **readFromFile** searches the default database. If the value of *qbeFileName* does not include an extension, this method assumes an extension of .QBE. To specify a filename that does not have an extension, add a period to the end of the name. For example, the following table lists the resulting filenames for various values of *qbeFileName*.

Value of <i>qbeFileName</i>	QBE filename
newcust	newcust.qbe
newcust.	newcust
newcust.q	newcust.q

readFromFile returns True if it succeeds; otherwise, it returns False.

Example

The following example reads and executes the query.

```
method pushButton(var eventInfo Event)
var
    qVar    Query
endVar

; this writes results into :PRIV:ANSWER.DB
qVar.readFromFile("GetCust.qbe")
qVar.executeQBE()

endMethod
```

readFromString method

Query

Assigns a query string to a Query variable.

Syntax

```
readFromString ( const QBEStrng String ) Logical
```


Description

readFromString assigns the query string specified in *QBEString* to a Query variable. Use **executeQBE** to execute the query.

Use **readFromString** to build a QBE string from smaller strings - a QBE string can be a combination of quoted strings and string variables.

You can use absolute paths or aliases to specify where to find tables in the query definition. Paradox also searches for unqualified table names (i.e., table names without paths or aliases) in a specified database. If a database is not specified, Paradox searches the default database (the working directory). Double backslashes are required when specifying a path.

Because a QBE string is a quoted string, it cannot contain tilde variables; however you can use string variables to achieve the same effect. To include tilde variables in a query, use a query statement.

Example

The following example uses the **pushButton** method for *btnFindName* to define a query as a string value and then uses **readFromString** to assign the string to a Query variable:

```
method pushButton(var eventInfo Event)
var
    db Database
    qs String
    tv TableView
    tc TCursor
    qVar Query
endVar

; Add the sampData alias then open the database.
addAlias("sampData", "Standard", "c:\\Core1\\Paradox\\samples")
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.

    qVar.readFromString(qs)

    if qVar.executeQBE() then
        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")
end
```

removeCriteria method

```
endIf  
  
endMethod
```

removeCriteria method

Query

Removes the query expression from a specified field.

Syntax

```
removeCriteria ( const tableID SmallInt, const fieldID SmallInt ) Logical  
removeCriteria ( const tableID SmallInt, const fieldName String ) Logical  
removeCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID SmallInt  
 ) Logical  
removeCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName String  
 ) Logical  
removeCriteria ( const tableName String, const fieldID SmallInt ) Logical  
removeCriteria ( const tableName String, const fieldName String ) Logical  
removeCriteria ( const tableName String, const rowID SmallInt, const fieldID SmallInt  
 ) Logical  
removeCriteria ( const tableName String, const rowID SmallInt, const fieldName String  
 ) Logical
```

Description

removeCriteria removes the query expression in a specified field, but does **not** remove check marks. Use **setCheck** and **clearCheck** to manipulate query check marks.

The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

Example

The following example removes the criteria from the Name field in the CUSTOMER.DB table in the query:

```
method pushButton(var eventInfo Event)  
var  
    qVar Query  
endVar  
  
    qVar.readFromFile( "Myquery.qbe" )  
    qVar.removeCriteria( "Customer.db" , "Name" )  
    qVar.executeQBE()          ; execute the saved query minus the  
                                ; customer name criteria.  
endMethod
```

removeRow method

Query

Deletes a row and its contents from the query workspace.

Syntax

```
removeRow ( const tableID SmallInt, const rowID SmallInt ) Logical  
removeRow ( const tableName SmallInt, const rowID SmallInt ) Logical
```

Description

removeRow deletes a row and its contents from a query.

The table is specified numerically by *tableID* or by *tableName*. The row is specified by the row identifier *rowID*.

Example

The following example removes the second row from the ORDER.DB table in MYQUERY.QBE:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    rowID SmallInt
endVar
qVar.readFromFile( "MyQuery.qbe" )
rowID = qVar.getRowID( "ORDERS.DB", 2 ) ; get the 2nd row
qVar.removeRow( "ORDERS.DB", rowID )
qVar.executeQBE()
endMethod
```

removeTable method

Query

Deletes a table from the query.

Syntax

```
removeTable ( const tableID SmallInt ) Logical
removeTable ( const tableName String ) Logical
```

Description

removeTable deletes a table from the query. The table is specified numerically by *tableID* or by *tableName*.

Example

The following example removes the table ORDERS.DB from the query image MYQUERY.QBE:

```
method pushButton(var eventInfo Event)
var
    qVar Query
endVar

qVar.readFromFile( "MyQuery.qbe" )
qVar.removeTable( "Orders.db" ) ; remove Orders.db from
                                ; the workspace
qVar.removeCriteria("Customer.db", "Customer No" ) ; clear the
                                                ; example element link.
qVar.executeQBE()
endMethod
```

setAnswerFieldOrder method

Query

Sets the field order of the answer table that is generated by a query.

Syntax

```
setAnswerFieldOrder ( var fieldOrder Array[] String ) Logical
```

Description

setAnswerFieldOrder sets the field order of the answer table generated by a query. The parameter *fieldOrder* is an array of field names that can be used the answer table structure. Use

getAnswerName to retrieve an array of these field names and then modify the order.

setAnswerName method

If you retrieve an array of field names and change the answer field list (e.g., by unchecking a field), the array is instantly out of date. You must remove the modified field from your array before using the array for field ordering. A specified field order must contain the same number of elements as fields in the answer table.

Example

The following example retrieves the field names from MYQUERY.QBE, changes their order, and uses **setAnswerFieldOrder** to put the new order in place.

```
method pushButton(var eventInfo Event)
var
  qVar Query
  arFields Array[] String
endVar
qVar.readFromFile( "myquery.qbe" )
qVar.getAnswerFieldOrder( arFields )
  if arFields.size() 0 then      ; swap the first and third fields
                                ; in the answer table.
    arFields.exchange(1,3)
    qVar.setAnswerFieldOrder( arFields )
    qVar.executeQBE()
  endif
endMethod
```

setAnswerName method

Query

Sets the name of the answer table that is generated by a query.

Syntax

```
setAnswerName ( const tableName String ) Logical
```

Description

setAnswerName specifies *tableName* as the name of the answer table that is created by the query.

Example

The following example allows the user to change the answer table name for MYQUERY.QBE:

```
method pushButton(var eventInfo Event)
var
  qVar Query
  AnsTblName String
endVar
if msgQuestion("Query",
  "Would you like to change the "
  + "answer table name?") = "Yes" then
  qVar.readFromFile("MYQUERY.QBE")
  AnsTblName = qVar.getAnswerName()
  AnsTblName.view("Make changes below")
  qVar.setAnswerName(AnsTblName)
  qVar.writeQBE("MYQUERY.QBE")
endif
endMethod
```

setAnswerSortOrder method

Query

Specifies the sort order for fields in the answer table.

Syntax

```
setAnswerSortOrder ( var sortFields Array[] String ) Logical
```

Description

setAnswerSortOrder specifies the sort order for fields in the answer table. The array *sortFields* contains an ordered list of field names. Use **getAnswerSortOrder** to retrieve this array and then change the field name positions to create a new sort order.

If you retrieve an array of field names and change the answer field list (e.g., by unchecking a field), the array is instantly out of date. You must remove the modified field from your array before using the array for field ordering.

Example

The following example retrieves the field list from MYQUERY.QBE, reorders the fields, and uses **setAnswerSortOrder** to put the new order in place:

```
method pushButton(var eventInfo Event)
var
    qVar Query
    arFields Array[] String
endVar
qVar.readFromFile( "myquery.qbe" )
qVar.getAnswerSortOrder( arFields )
    if arFields.size() > 0 then          ; swap the first and third fields
        arFields.exchange(1,3)          ; in the sort order.
        qVar.setAnswerSortOrder( arFields )
        qVar.executeQBE()
    endif
endMethod
```

setCriteria method**Query**

Specifies the criteria for a table's field.

Syntax

```
setCriteria ( const tableID SmallInt, const fieldID SmallInt, const newCriteria String
) Logical
setCriteria ( const tableID SmallInt, const fieldName String, const newCriteria String
) Logical
setCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldID SmallInt,
const newCriteria String ) Logical
setCriteria ( const tableID SmallInt, const rowID SmallInt, const fieldName String,
const newCriteria String ) Logical
setCriteria ( const tableName String, const fieldID SmallInt, const newCriteria String
) Logical
setCriteria ( const tableName String, const fieldName String, const newCriteria String
) Logical
setCriteria ( const tableName String, const rowID SmallInt, const fieldID SmallInt,
const newCriteria String ) Logical
setCriteria ( const tableName String, const rowID SmallInt, const fieldName String,
const newCriteria String ) Logical
```

Description

setCriteria specifies a query expression string to be used as the criteria for a specific table's field. The table is specified numerically by *tableID* or by *tableName*. The field is specified by the *fieldID* or by *fieldName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row. The criteria is specified by *newCriteria*.

setLanguageDriver method

setCriteria does **not** support check marks.

Example

The following example sets the criteria for the appended row (State/Prov) to either CA or HI.

```
method pushButton(var eventInfo Event)
var
  qVar Query
  rowID SmallInt
endVar

qVar.appendTable( "CUSTOMER.DB" )
rowID = qVar.appendRow( "CUSTOMER.DB" )
qVar.setCriteria( "CUSTOMER.DB", rowID, "State/Prov", "CA or HI" )
qVar.checkRow("Customer.db", rowID, CheckCheck)
qVar.writeQBE("MyQBE")
endMethod
```

setLanguageDriver method

Query

Specifies the name of the default language driver for your system.

Syntax

```
setLanguageDriver ( const languageDriver String ) Logical
```

Description

setLanguageDriver specifies the default language driver to the driver specified by *languageDriver*. The language driver is a part of the table's definition. The language drivers for Paradox tables are listed in of the description of the Table type's **create** method.

If you execute a query on a table that uses a different language driver you can use System type's **getLanguageDriver** to identify the language driver of the table. Set the language driver for the query using *setLanguageDriver*, to create the query's answer table with the same driver.

Example

The following example sets the language driver to Czech:

```
method pushButton(var eventInfo Event)
var
  myQBE Query
endvar

myQBE = Query

Customer|Customer No | Name |
        |Check       | A.. |

endQuery
myQBE.setLanguageDriver ( "ANCZECH" )
myQBE.executeQBE()
endMethod
```

setQueryRestartOptions method

Query

Specifies the function of the underlying tables while running a query.

Syntax

```
setQueryRestartOptions ( const qryRestartType SmallInt ) Logical
```

Description

setQueryRestartOptions tells Paradox what to do if data changes while you're running a query in a multi-user environment. The argument *qryRestartType* represents one of the following ObjectPAL QueryRestartOptions constants:

QueryDefault	Use the options specified interactively by using the Query Restart Options dialog box.
QueryLock	Lock other users out of the tables needed to run the query. If Paradox cannot lock a table, it does not run the query.
QueryNoLock	Run the query even if the data changes while it's running.
QueryRestart	Restart the query. Specify QueryRestart to get a snapshot of the data as it existed at a particular instant.

Example 1

The following example calls **getQueryRestartOptions** to retrieve the user's query restart options. The code uses **executeSQL**. If the setting is not **QueryRestart**, this code calls **setQueryRestartOptions** to set it before executing the query:

```
method pushButton(var eventInfo Event)
    var
        qVar    SQL
        MyDB    database
    endVar

    MyDB.open("work")

    if qVar.getQueryRestartOptions() QueryRestart then
        setQueryRestartOptions(QueryRestart)
    endIf

    if qVar.readFromFile("newcust.sql") then
        qVar.executeSQL(MyDB)
    else
        errorShow()
    endIf
endMethod
```

Example 2

The following example calls **getQueryRestartOptions** to retrieve the current query restart options. The code uses **executeQBE**. If the setting is not **QueryRestart**, this code calls **setQueryRestartOptions** to set it and executes the query:

```
method pushButton(var eventInfo Event)
    var
        qVar    Query
    endVar

    if getQueryRestartOptions() QueryRestart then
        setQueryRestartOptions(QueryRestart)
    endIf

    if qVar.readFromFile("newcust.qbe") then
        qVar.executeQBE()
    else
        errorShow()
    endIf
endMethod
```

setRowOp method

Query

Sets the row operator for a specific row.

Syntax

```
setRowOp ( const tableID SmallInt, const rowID SmallInt, const rowOperator SmallInt)
Logical
```

```
setRowOp ( const tableID SmallInt, const rowOperator SmallInt) Logical
```

```
setRowOp ( const tableName String, const rowID SmallInt, const rowOperator SmallInt)
Logical
```

```
setRowOp ( const tableName String, const rowOperator SmallInt) Logical
```

Description

setRowOp sets one of the four row operators in a specified row. The table is specified numerically by *tableID* or by *tableName*. The row is specified by the row identifier *rowID*. If no row is specified, this method defaults to the first row.

The *rowOperator* is one of the following values:

qbeRowDelete	Delete operator
qbeRowInsert	Insert operator
qbeRowNone	No operator
qbeRowSet	Set operator

Example

The following example deletes records with blank Customer No. fields.

```
method pushButton(var eventInfo Event)
var
    qVar Query
endVar

qVar.appendTable( "Customer.db" )
qVar.setRowOp( "Customer.db" , qbeRowDelete)
qVar.setCriteria( "Customer.db" , "Customer No" , "blank" )
                                ; delete blank Customer No records.
qVar.executeQBE()

endMethod
```

wantInMemoryTCursor method

Query

Specifies how to create a TCursor resulting from a query.

Syntax

```
wantInMemoryTCursor ( [ const yesNo Logical ] )
```

Description

wantInMemoryTCursor specifies how to create a TCursor resulting from a query. When you call **wantInMemoryTCursor** with *yesNo* set to as Yes or omitted, Paradox creates a dead TCursor in system memory, with no connection to underlying tables. When *yesNo* is No, Paradox creates a TCursor in a live query view. By default, when you execute a query to a TCursor, that TCursor will point to a live query view - changes made to the TCursor will affect the underlying tables. Set **wantInMemoryTCursor** to Yes when you *don't* want a live query view.

An in-memory TCursor can be useful for performing quick analyses. Suppose you want to study the effects of giving each employee a 15 percent raise. Query the employee data to increase everyone's salary by 15 percent and execute the query to an in-memory TCursor. Now you can work with the queried data there, without affecting the actual employee data.

Example

The following example uses an in-memory TCursor to study the effects of giving every employee a 15 percent raise. The code reads and executes a predefined query and then uses the results in a calculation:

```
method pushButton(var eventInfo Event)
  var
    qVar      Query
    tcRaise15  TCursor
    nuTotalPayroll Number
  endVar

  qVar.wantInMemoryTCursor(Yes)
  qVar.readFromFile("raise15.qbe")
  qVar.executeQBE(tcRaise15)

  nuTotalPayroll = tcRaise15.cSum("Salary")
  nuTotalPayroll.view("Payroll after 15% raise:")

endMethod
```

writeQBE method/procedure

Query

Writes a query statement to a specified file.

Syntax

1. (Method) **writeQBE** (const *fileName* String) Logical
2. (Procedure) **writeQBE** (const *str* String , const *fileName* String) Logical

Description

writeQBE writes a predefined query to the file specified in *fileName*. If *fileName* exists, this method overwrites it without asking for confirmation. If *fileName* does not specify a path, Paradox writes to the working directory. **writeQBE** returns True if the write succeeds; otherwise it returns False.

Syntax 1 calls **writeQBE** as a method. It writes the query represented by an assigned Query variable to the file specified in *fileName*.

Syntax 2 calls **writeQBE** as a procedure. It writes the query string represented by *str* to the file specified in *fileName*.

Example

The following example assumes that a form contains a button named *getDest*. When the form opens, this code 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 and then opens the table. This example assumes that the :MAST: alias has already been defined.

The following code is attached to the **open** method for *pageOne*:

```
method pushButton(var eventInfo Event)
  Var
    qVar Query
  endVar
```

writeQBE method/procedure

```
; if the GetDest.qbe query file doesn'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
    qVar.writeQBE("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 that the file is available.

```
method pushButton(var eventInfo Event)
var
    qVar    Query
    tv      TableView
endVar

qVar.readFromFile("GetDest.qbe")
qVar.executeQBE("MyDest")
tv.open("MyDest")

endMethod
```

You can also use **writeQBE** method with ObjectPAL to create and save a query that your user can run interactively, using the Query Editor.

Record type

ObjectPAL's Record type is a programmatic, user-defined collection of information that resembles a **record** in Pascal or a **struct** in C. Records that are defined in ObjectPAL code are separate and distinct from records associated with a table.

The following code declares a Record data type:

```
TYPE
recordName = RECORD
    fieldName fieldType
    [ fieldName fieldType ] *
ENDRECORD
ENDTYPE
```

fieldName identifies fields or columns in the record, and *fieldType* specifies one of the data types.

Records are declared in a design object's Type window.

After you declare a Record data type, you can use the = and comparison operators to compare records. You can also use the (=) assignment operator to copy a record's contents to another record.

Several predefined record structures have been created for specific situations. For more information, see the following:

- **FormOpenInfo** (see **Open** (Form type))

- **ReportOpenInfo** (see **Open** (Report type))
- **FileBrowserInfo** (see **FileBrowser** (System type))
- **PrinterOptionInfo**, **PrinterGetOptions** (System type)), and **PrinterSetOptions** (System type)).

When declaring a record variable of these structures, use the predefined structure name instead of declaring the variable as a Record type. For example, if you want to declare a variable called *MySettings* which has the predefined structure of *FormOpenInfo*, do the following:

```
var
  MySettings    FormOpenInfo    ; note that you do not declare MySettings as type Record
endvar
```

ObjectPAL automatically creates the *MySettings* variable with the predefined structure for a *FormOpenInfo* record. Any Record methods can be used with *MySettings*.

The Record type includes several derived methods from the AnyType type.

Methods for the Record type

AnyType	←	Record
blank		view
dataType		
isAssigned		
isBlank		
isFixedType		

view method

Record

Displays the value of a variable in a dialog box.

Syntax

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

Description

view displays the value or values assigned to a Record variable in a modal dialog box. 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.

Note

- Values in a Record can't be changed when displayed in a **view** dialog box. For more information, see AnyType.

Example

The following example uses the **pushButton** method for *getAndViewRec* to declare a variable called *myRec* of the MyRecord type. This method opens a TCursor to the *Customer* table, fills *myRec* with the *Customer No* and *Name* field values, and uses view to display the record in a dialog box. The operation is then repeated for the second record in *Customer*.

The following code is attached to the Type window for *getAndViewRec* to create a user-defined type named MyRecord:

```
; getAndViewRec::Type
Type
  MyRecord = RECORD          ; define a Record structure
    ID      String
```

view method

```

                Name String
            ENDRECORD
        endType

    The following 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
```

Report type

A Report variable provides a handle to a report. You use Report variables in code to manipulate the report onscreen. Report methods control the window's size, position, and appearance, and allow you to view and print the report.

Use **load** to load a report file in the Report Design window; use **open** to open the report in the Report window, and use **print** to open a report and print it. You cannot attach methods to objects in a report but you can attach code to calculated fields.

The following table displays the methods for the Report type, including several derived methods from the Form type.

Methods for the Report type

Form	←	Report
bringToTop	isDesign	attach
create	isMaximized	close
deliver	isMinimized	currentPage
dmAddTable	isVisible	design
dmBuildQueryString	maximize	enumUIObjectNames
dmEnumLinkFields	menuAction	enumUIObjectProperties
dmGetProperty	minimize	load

dmHasTable	save	moveToPage
dmLinkToFields	saveStyleSheet	open
dmLinkToIndex	selectCurrentTool	publishTo
dmRemoveTable	setIcon	print
dmSetProperty	setPosition	run
dmUnlink	setProtoProperty	setMenu
enumDataModel	setSelectedObjects	
enumTableLinks	setStyleSheet	
getFileName	setTitle	
getPosition	show	
getProtoProperty	wait	
getStyleSheet	windowClientHandle	
getTitle	windowHandle	
hide	writeText	
isAssigned		

attach method

Report

Associates a Report variable with an open report.

Syntax

```
attach ( const reportTitle String ) Logical
```

Description

attach associates a Report variable with the 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 filename). You can use **getTitle** to return this text, or you can use **setTitle** to specify a new title.

Example

In the following example, assume the form's **open** method opened the STOCK.RSL report and retitled the window as Stock Report. The **pushButton** method for *printStock* attaches to the open report and prints its content.

```
; 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
```

close method

```
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
```

close method

Report

Closes a Report window.

Syntax

```
close ( )
```

Description

close closes a Report window. This method is the equivalent of choosing Close from the Control menu.

Example

The following example assumes that the form's **open** method opened the STOCK.RSL report and retitled the window as Stock Report. The **close** method for the form attaches to the open report and 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
```

currentPage method

Report

Returns the report's current page number.

Syntax

```
currentPage ( ) SmallInt
```

Description

currentPage returns the report's current page number.

Example

In the following example, the **pushButton** method for *plusTwoPages* attaches to an open report. If this fails, the code opens the report. When the *ordersRep* variable points to an open report, this code 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

```

design method

Report

Switches a report from a Report window to a Report Design window.

Syntax

```
design ( ) Logical
```

Description

design switches a report from the Report window to the Report Design window. This method works only with saved reports (.RSL) and not with delivered reports (.RDL).

Use **run** to switch from a Report Design window to a Report window. Use **load** to open a report in a Report Design window.

Note

- You might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. For more information, see the sleep method in the System type.

Example

In the following example, assume that the form's **open** method opened the STOCK.RSL report and retitled the window as Stock Report. The **pushButton** method for *stockDesign* attaches to the open report and 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 Design mode
endMethod

```

enumUIObjectNames method

Report

Creates a table listing the UIObjects in a report.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

enumUIObjectProperties method

Description

enumUIObjectNames creates a Paradox table listing the name and type of objects contained in a specified 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.

The following table displays the structure of *tableName*:

Field	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.open("ordprops")
tempTable.wait()
tempTable.close()
endMethod
```

enumUIObjectProperties method

Report

Lists the properties of each UIObject in a report.

Syntax

```
enumUIObjectProperties ( const tableName String ) Logical
```

Description

enumUIObjectProperties creates a Paradox table listing the name, property name, and property value of each object 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 following table displays the structure of *tableName* is:

Field	Type	Size
ObjectName	A	128
PropertyName	A	64

PropertyType	A	48
PropertyValue	A	255

Example

See the `enumUIObjectNames` example.

load method**Report**

Opens a report in the Report Design window.

Syntax

```
load ( const reportName String, [const windowStyle LongInt [ , const x LongInt, const
y LongInt, const w LongInt, const h LongInt ] ] ) Logical
```

Description

load opens *reportName* in the Report Design window. You can specify a WindowStyles constant (or combination of constants) in *windowStyle*. You can also specify the window's size and position (in twips). Arguments *x* and *y* specify the position of the upper-left corner, and arguments *w* and *h* specify the window's width and height, respectively. This method supports only saved reports (.RSL), and not delivered reports (.RDL).

Compare this method to **open**, which opens a report in the Report window.

Note

- It is possible to load a form as a report. Declare a report variable and load a form using it. (For example: `r.load("form.fsl")`)
- You might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. For more information, see the `sleep` method in the `System` type.

Example

In the following example, the **pushButton** method for the *loadOrders* button loads the `ORDERS.RSL` report in the Report Design window. This code 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
```

moveToPage method**Report**

Displays the specified page of a report.

open method

Syntax

```
moveToPage ( const pageNumber SmallInt ) Logical
```

Description

moveToPage displays the page of a report specified in *pageNumber*. This method doesn't make the report active. To make the report active, follow *moveToPage* with *bringToTop* (for more information about *bringToTop*, see the Form type).

Note

- To access the Page Number for a report, check the *PositionalOrder* property of the report in the ObjectExplorer. This property can be used in ObjectPAL as well. For example, `moveToPage(page#.PositionalOrder)`.

Example

See the **currentPage** example.

open method

Report

Opens a report.

Syntax

```
1. open ( const reportName String [, windowStyle LongInt ] ) Logical
2. open ( const reportName String, const windowStyle LongInt, const x SmallInt, const
y SmallInt, const w SmallInt, const h SmallInt ) Logical
3. open ( const openInfo ReportOpenInfo ) Logical
```

Description

open displays the report specified in *reportName*. Optional arguments specify the location of the report's upper-left corner (*x* and *y*), its width and height (*w* and *h*), and its style (*windowStyle*).

The value of *windowStyle* must be one of the WindowStyles constants. You can specify more than one window style by adding the constants. The following code opens a report window that has horizontal and vertical scroll bars:

```
salesReport.open("sales.rsl", WinStyleDefault + WinStyleHScroll + WinStyleVScroll)
```

Syntax 3 allows you to specify form settings from *openInfo*, a predefined record of type ReportOpenInfo. A ReportOpenInfo record is an instance of the Record Type, and has the following structure:

<i>x</i> , <i>y</i> , <i>w</i> , <i>h</i>	LongInt	;size and position of report
<i>name</i>	String	;name of report to open (preView)
<i>masterTable</i>	String	;master table name
<i>queryString</i>	String	;run this query (actual query string)
<i>restartOptions</i>	SmallInt	;one of the ReportPrintRestart constants
<i>SQLString</i>	String	;run this SQL query (actual query string)
<i>winStyle</i>	LongInt	;one of the WindowStyle constants

The MasterTable field can also be the name of a SQL file that produces an Answer table.

ReportOpenInfo now has a new field called SQLString, which can be used to specify an SQL statement to execute.

To rebind a report to a newly-created SQL statement, save the SQL statement to a file and specify the filename in ReportPrintInfo.MasterTable or ReportOpenInfo.MasterTable.

Note

- It is possible to open a form as a report. Declare a report variable and open a form using it. (For example: `r.open("form.fsl")`)
- You might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. For more information, see the sleep procedure in the System type.

Example

In the following example, the **pushButton** method for *openSmall* opens the ORDERS.RSL report and minimizes it by supplying the window style constant `WinStyleMinimize`:

```
; openSmall::pushButton
method pushButton(var eventInfo Event)
var
    ordersRep Report
endVar
ordersRep.open("Orders.rsl", WinStyleMinimize) ; open Orders Report minimized
endMethod
```

publishTo**Report**

Publish the report to a variety of different publishing types.

Syntax

```
publishTo ( const filename String, const PublishTo SmallInt) Logical
```

Description

publishTo will publish the specified report to a specified publishing type. For the filename argument, you must specify the output filename. **publishTo** is supplied from the constant class *PublishToFilters* and must be set by one of the following arguments:

Argument	Description
<code>publishToRTF</code>	The file will be published to richtext format.
<code>publishToWP9</code>	The file will be published to WordPerfect V9.0 format.
<code>publishToWord97</code>	The file will be published to Microsoft Word97 format.

The filename may be prefixed with an alias. If no alias or path is supplied to assumes the current working directory. If there is no extension placed on the file, then the appropriate extension is placed on it.

Note

- The only filter that will work in Paradox Runtime is **publishToRTF**.

Example

In the following example, CUSTOMER.RSL is published as an RTF file type using the **publishToRTF** procedure.

```
method run (var eventInfo Event)
var
    r report
endvar
if r.open ( "CUSTOMER.RSL" ) then
    r.publishTo ( "Testing", publishToRTF )
    r.close ()
end
```

print method

```
endif  
endmethod
```

print method

Report

Prints a report.

Syntax

1. `print () Logical`
2. `print (const reportName String, const reportPrintRestart SmallInt) Logical`
3. `print (const ri ReportPrintInfo) Logical`

Description

print prints a report. Syntax 1 instructs Paradox to open the Print dialog box for the current report, which allows the user to specify print settings. Syntax 2 allows you to specify a report name in **reportName** and set restart options in **reportPrintRestart**. The value of **reportPrintRestart** must be one of the **ReportPrintRestart** constants. Syntax 3 lets you set print settings with a ReportPrintInfo record. The predefined ReportPrintInfo records, which are of the Record Type, have the following structure:

Field name	Type	Description
endPage	LongInt	Specifies the last page in a range (defaults to the last page of the report)
makeCopies	Logical	Specifies whether copies are made by Paradox or the printer. If set to True, Paradox make copies; otherwise, the printer makes copies (defaults to True). The value is ignored if the printer cannot print multiple copies.
masterTable	String	Specifies the name of the master table for the report
name	String	Specifies the name of a report to run (if one is not already running)
nCopies	SmallInt	Specifies the number of copies (defaults to one)
orient	SmallInt	Specifies the page orientation. Use one of the three ReportOrientation Constants: Landscape, Portrait, or the windows default.
pageIncrement	SmallInt	Specifies the page increment for multi-pass printing (defaults to one)
panelOptions	SmallInt	Specifies how to handle overflow pages. Use one of the ReportPrintPanel constants (defaults to PrintClipToWidth)
printBackwards	Logical	Specifies whether to print forward (from first page to last page) or backward (from last page to first page). If set to False, Paradox prints forward; otherwise it prints backwards (defaults to False).
queryString	String	Specifies a QBE string to execute

restartOptions	SmallInt	Specifies what to do when data changes while printing a report. Use one of the ReportPrintRestart constants (defaults to PrintReturn)
SQLString	String	Specifies a SQL query string to execute
startPage	LongInt	Specifies the first page of a range (defaults to one)
startPageNum	LongInt	Specifies the page number to print on the first page of the report. Incremented for subsequent pages (defaults to one)
xOffset	LongInt	Specifies the horizontal page offset (defaults to zero)
yOffset	LongInt	Specifies the vertical page offset (defaults to zero)

Example

The following example uses Syntax 3 to print using a **ReportPrintInfo** record. To print using Syntax 1, see the **attach** example.

```
; 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
```

run method

Report

Switches a report from the Report Design window to the Report window.

Syntax

```
run ( ) Logical
```

Description

run switches a report from the Report Design window to the View Data window. This method works only with saved reports (.RSL), and not with delivered reports (.RDL).

Use **design** to switch from the View Data window to the design window.

Note

- You might need to follow a call to **open**, **load**, **design**, or **run** with a **sleep**. For more information, see the **sleep** procedure in the System type.

Example

See the **load** example.

setMenu method

Report

Associates a menu with a report.

setMenu method

Syntax

```
setMenu ( const menuVar Menu )
```

Description

setMenu associates the menu specified in *menuVar* with a report. This method performs the same function as the Menu type **show**, and adds the following:

- When the report gets focus, Paradox displays the associated menu.
- Actions resulting from choices from that menu are sent to that report.

Note

- When you build a custom menu for a report, use MenuCommands constants (e.g., MenuFilePrint) to assign ID values to menu items. Because reports do not have menuAction methods for handling menu choices, these ID values are the only values that a report can respond to.

Example

The following example is a script that opens a report, builds a simple menu and then uses **setMenu** to assign the menu to the report:

```
method run(var eventInfo Event)
  var
    reOrders    Report
    muOrderRpt  Menu
    puRptFile   PopUpMenu
  endVar

; Build a menu for the report.
  reOrders.open("orders")

; Setting the StandardMenu property to False
; (either in ObjectPAL code or interactively)
; can reduce flicker when changing menus.
  reOrders.StandardMenu = False

; IMPORTANT: When you build a custom menu for a report,
; use MenuCommands constants (like MenuFilePrint) to assign
; ID values to menu items. These are the only values a report
; can respond to, because (unlike a form) a report has no
; menuAction method you can modify to handle menu choices.

  puRptFile.addText("&Print Report", MenuEnabled, MenuFilePrint)
  puRptFile.addText("&Exit", MenuEnabled, MenuFileExit)
  muOrderRpt.addPopUp("&File", puRptFile)
  reOrders.setMenu(muOrderRpt)
endMethod
```

Script type

Script type includes methods for manipulating scripts—and the code they contain—from within an ObjectPAL method or procedure.

The Script type includes several derived methods from the Form type.

Methods for the Script type

Form	←	Script
deliver		attach
enumSource		create
enumSourceToFile		load
formReturn		methodEdit
isAssigned		run
isCompileWithDebug		
methodDelete		
methodGet		
methodSet		
save		
setCompileWithDebug		

attach method**Script**

Associates a Script variable with the active script.

Syntax

```
attach ( ) Logical
```

Description

attach associates a Script variable with the active script. Because this method must be called in code attached to the script itself, the script must be running. This means that **attach** allows a running script to create a handle to itself. Since ObjectPAL can't return Script variables or pass them as arguments, you must only use the handle within the method that created it. **attach** can be used with **enumSource** or **enumSourceToFile** to create a script that enumerates its own code.

This method returns True if it succeeds; otherwise, it returns False.

Example

The following example uses **attach** to create a script that enumerates its source to a text file. The code is attached to the script's built-in run method, which executes when you run the script.

```
method run(var eventInfo Event)
  var
    s Script
  endVar
  s.attach()
  s.enumSourceToFile("script.src", Yes)
endMethod
```

create method**Script**

Creates a script.

Syntax

```
create ( ) Logical
```

Description

create creates an empty script but **does not** display an Editor window. Use **methodSet** to add code to the script.

load method

Example

The following example uses the **pushButton** method for a button named *editScript* to create a script named MSG. The code then calls **methodSet** to attach code to its built-in **run** method, calls **save** to save the script as *NewMsg*, and calls **run** to execute it. Paradox automatically appends the .SSL extension.

```
; editScript::pushButton
method pushButton(var eventInfo Event)
  var
    theScript    Script
    stMsg        String
  endVar

  stMsg =
    "method newMsg()
      msgInfo(\"New message\", \"New message\")
    endMethod"

  theScript.create()
  theScript.methodSet("run", stMsg)
  theScript.save("NewMsg") ; Saves script as NEWMSG.SSL.
  theScript.run()         ; Calls the script's built-in run method.
endMethod
```

load method

Script

Loads a script into system memory.

Syntax

```
load ( const scriptName String ) Logical
```

Description

load loads the script specified in *scriptName* into system memory, but does not display an Editor window. If you don't specify a path or an alias in *scriptName*, Paradox looks for the script in the working directory. This method returns True if it succeeds; otherwise, it returns False.

Example

The following example uses the **pushButton** method for a button named *editScript* to load the script named MSG. MSG must have been created and saved previously. The code then uses **methodSet** to add a custom method, calls **save** to save the script, and calls **run** to execute it.

```
; editScript::pushButton
method pushButton(var eventInfo Event)
  var
    theScript    Script
    stMsg        String
  endVar

  stMsg =
    "method newMsg()
      msgInfo(\"New message\", \"New message\")
    endMethod"

  if theScript.load("msg") then
    theScript.methodSet("newMsg", stMsg)
    theScript.save()
    theScript.run() ; Executes the script's built-in run method.
  else
```



```

        errorShow("Couldn't load the script.")
    endIf
endMethod

```

methodEdit method

Script

Opens a script's method in an Editor window.

Syntax

```
methodEdit (const methodName String) Logical
```

Description

methodEdit opens the method specified by *methodName* in an Editor window. If you specify a method that doesn't exist, **methodEdit** will create it for you. **methodEdit** fails if you try to open a method that is running.

Note

- While editing a method in this manner the script cannot be run.

Example

The following example opens the script's **testMethod** method in an editor window:

```

method pushButton(var eventInfo Event)
var
    MyScript script
endvar
MyScript.load("update.ssl")
MyScript.methodEdit("testMethod")
endMethod

```

run method

Script

Runs a script.

Syntax

```
run ( ) AnyType
```

Description

run runs a script by calling the script's built-in **run** method. **run** performs the same operation as the System procedure **play**. To return a value from a script, you must call **formReturn** from within the script.

Example

The following example runs a script and makes it return a value. The following code is attached to a button in a form. It runs a script and displays the returned value in a dialog box.

```

method pushButton(var eventInfo Event)
var
    scTest      Script
    atRetVal    AnyType
endVar

scTest.load("test")
atRetVal = scTest.run()

```

addAlias method/procedure

```
        atRetVal.view()
    endMethod
```

The following code is attached to a script’s built-in **run** method. It assigns a value to a variable and returns the value to the form.

```
method run(var eventInfo Event)
    var
        atNow    AnyType
    endVar
    atNow = time()
    formReturn(atNow)
endMethod
```

Session type

A Session object represents a channel to the database engine. When you launch a Paradox application one session opens by default. You can use ObjectPAL to open additional sessions from within an application. Only the default session can be managed using Paradox interactively. You must manage other sessions using ObjectPAL.

Locks set by ObjectPAL interact as peers with locks set interactively in the same session.

Methods for the Session type

addAlias	enumFolder	removeAlias
addPassword	enumOpenDatabases	removeAllPasswords
addProjectAlias	enumUsers	removePassword
advancedWildcardsInLocate	getAliasPath	
blankAsZero	getAliasProperty	removeProjectAlias
close	getNetUserName	retryPeriod
enumAliasLoginInfo	ignoreCaseInLocate	saveCFG
enumAliasNames	isAdvancedWildcardsInLocate	saveProjectAliases
enumDatabaseTables	isAssigned	setAliasPassword
enumDriverCapabilities	isBlankZero	setAliasPath
enumDriverInfo	isIgnoreCaseInLocate	setAliasProperty
enumDriverNames	loadProjectAliases	setRetryPeriod
enumDriverTopics	lock	unLock
enumEngineInfo	open	

addAlias method/procedure

Session

Adds a public alias to a session.

Syntax

- 1. addAlias (const *aliasName* String, const *type* String, const *path* String) Logical
- 2. addAlias (const *aliasName* String, const *type* String, const *params* DynArray[] String) Logical
- 3. addAlias (const *aliasName* String, const *existingAlias* String) Logical

Description

addAlias adds public alias a to a session. To add a project alias, use **addProjectAlias**.

In Syntax 1, specify the alias name in *aliasName*, its (Standard) in *type*, and its full DOS path in *path*.

In Syntax 2, specify the alias name in *aliasName*, the SQL alias type (Interbase, Oracle, Sybase, or Informix) in type, and the parameters in *params*.

Syntax 3 copies an alias from *existingAlias* to *aliasName*.

An alias added using **addAlias** is known only to the session for which it is defined, and exists only until the session is closed. Use **saveCFG** to save public aliases in a file. By default, public aliases are stored in IDAPI.CFG. They are available from any working directory and visible to any application that uses Borland Database Engine (BDE).

Example 1

The following example adds an alias to the active session and 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 active session
addAlias("CustomerInfo", "Standard", "D:\\Corel\\Paradox\\tables\\custdata")

; now use the alias to specify which database to open
custInfo.open("CustomerInfo") ; opens the CustomerInfo database

endMethod
```

Example 2

The following example adds an Oracle type alias to the active session and 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:

```
; pgeOne::open
method open(var eventInfo Event)
var
    tv      TableView
    SQLdb    Database
    AliasInfo DynArray[] String
endVar

AliasInfo["SERVER NAME"]      = "Server1"
AliasInfo["USER NAME"]        = "guest"
AliasInfo["OPEN MODE"]        = "READ/WRITE"
AliasInfo["SCHEMA CACHE SIZE"] = "8"
AliasInfo["NET PROTOCOL"]     = "SPX/IPX"
AliasInfo["LANGDRIVER"]       = ""
AliasInfo["SQLQRYMODE"]       = ""
AliasInfo["PASSWORD"]         = "guest"

addAlias("Guest_Account", "Oracle", AliasInfo)
SQLdb.open("Guest_Account", AliasInfo)
tv.open(":Guest_Account:swilson.customer")

endMethod
```

Example 3

The following example adds an alias to the active session by copying the existing *work* alias to the a new alias named *NewAlias*:

addPassword method/procedure

```
; btnCopyWork::pushButton
method pushButton(var eventInfo Event)
    addAlias("NewAlias", "work")
endMethod
```

addPassword method/procedure

Session

Defines a password allowing access to a protected table.

Syntax

```
addPassword ( const password String )
```

Description

addPassword provides a Paradox session the password specified in *password*. Subsequent attempts to access a table protected by that password are not challenged.

The argument *password* represents an owner password or an auxiliary password. Auxiliary passwords generally confer less comprehensive rights than owner passwords. Because *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 for which they are defined, and exist only until the session is closed. Defining a password does not affect the state of tables (e.g., an open table remains open).

Access to tables opened before the password is presented is controlled by previously defined passwords. For example, if a table was opened using an auxiliary password, the access rights to that table do not change when the owner password is defined. To confer owner rights to a previously-opened table, close the table and present the owner password, and then reopen the table.

Use **removePassword** to restore protection to tables.

Note

- Passwords apply to Paradox tables only and cannot exceed 31 characters.

Example

The following example acquires a user's password, and defines it for the active 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 (up to 31 characters) to Add.")
    ses.addPassword(newPass)
else
    msgStop("Help!","Session variable is not Assigned!")
endif
endMethod
```

addProjectAlias method/procedure

Session

Adds a project alias to a session.Syntax

```
1. addProjectAlias ( const aliasName String, const type String, const path String )
Logical
```

```

2. addProjectAlias ( const aliasName String, const type String, const params DynArray[
] String ) Logical
3. addProjectAlias ( const aliasName String, const existingAlias String ) Logical

```

Description

addProjectAlias adds a project alias to a session. Use **addAlias** to add a public alias.

In Syntax 1, specify the alias name in *aliasName*, its (Standard) in type, and its full DOS path in path.

In Syntax 2, specify the alias name in *aliasName*, the SQL alias type (Interbase, Oracle, Sybase, or Informix) in type, and the parameters in *params*.

Syntax 3 copies an alias from *existingAlias* to *aliasName*.

An alias added using **addProjectAlias** is known only to the project for which it is defined, and exists only until the working directory is changed. Use **saveProjectAliases** to save project aliases in a file.

When :WORK: is set (e.g., at startup) or changed (interactively or using ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available and if a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG. You can use **loadProjectAliases** to specify a different file.

Example 1

The following example adds an alias to the active project and 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 project
addProjectAlias("CustomerInfo", "Standard", "D:\\Core1\\Paradox\\tables\\custdata")

; now use the alias specify the database to open
custInfo.open("CustomerInfo") ; opens the CustomerInfo database

endMethod

```

Example 2

The following example adds an Oracle type alias to the active project and 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.

```

; pgeOne::open
method open(var eventInfo Event)
var
    tv      TableView
    SQLdb   Database
    AliasInfo DynArray[] String
endVar

AliasInfo["SERVER NAME"]      = "Server1"
AliasInfo["USER NAME"]        = "guest"
AliasInfo["OPEN MODE"]        = "READ/WRITE"
AliasInfo["SCHEMA CACHE SIZE"] = "8"
AliasInfo["NET PROTOCOL"]      = "SPX/IPX"
AliasInfo["LANGDRIVER"]       = ""

```

advancedWildcardsInLocate procedure

```
AliasInfo["SQLQRYMODE"]      = ""
AliasInfo["PASSWORD"]        = "guest"

addProjectAlias("Guest_Account", "Oracle", AliasInfo)
SQLdb.open("Guest_Account", AliasInfo)
tv.open(":Guest_Account:mprestwood.customer")
endMethod
```

Example 3

The following example adds an alias to the active session by copying the existing *work* alias to the new alias *NewAlias*:

```
; btnCopyWork::pushButton
method pushButton(var eventInfo Event)
    addProjectAlias("NewAlias", "work")
endMethod
```

advancedWildcardsInLocate procedure

Session

Specifies whether the active session can use advanced wildcards in locate operations.

Syntax

```
advancedWildcardsInLocate ( [ const yesNo Logical ] )
```

Description

advancedWildcardsInLocate specifies whether the active session can use advanced wildcards found in pattern strings during locate operations. If *yesNo* is set to Yes (default), pattern strings used in locate operations can contain advanced wildcard characters. If *yesNo* is set to No, pattern strings in locate operations cannot contain advanced wildcards.

Example

The following example calls **advancedWildcardsInLocate** to determine whether advanced wildcards can be used in a locate operation. The code then calls **locatePattern** to use 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
```

blankAsZero method/procedure**Session**

Specifies whether to assign blank numeric fields a value of 0 in calculations.

Syntax

```
blankAsZero ( const yesNo Logical )
```

Description

blankAsZero specifies whether to assign blank numeric fields a value of 0 in calculations. If *yesNo* is set to Yes, blanks are treated as zeros. If *yesNo* is set to No blank numeric fields remain empty.

Calculations affected by **blankAsZero** include:

- calculated fields in forms and reports
- calculations in queries
- column calculations that involve the number of fields or the number of non-blank fields (e.g., those performed with cCount, cAverage, and others)

You can also use **isBlankZero** to test the state, and **blankAsZero** to set it.

Example

The following example sets **blankAsZero** to True so that a call to the **cAverage** method assigns blank field values a value of 0.

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

if tc.open("Orders.db") then
    if not isBlankZero() then
        blankAsZero(True)
    endif

    msgInfo("Average Amount Paid", tc.cAverage("Amount Paid"))

else
    msgStop("Error", "Can't open Orders table.")
endif

endMethod
```

close method**Session**

Closes a session.

Syntax

```
close ( ) Logical
```

Description

close ends a session by closing the channel to the database engine. **close** frees one user count, and leaves the Session variable unassigned.

Example

The following example assumes that the variable *ses* is assigned to an open session. This example closes the session:

enumAliasLoginInfo method

```
; 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
```

enumAliasLoginInfo method

Session

Writes server alias data to a table.

Syntax

```
enumAliasLoginInfo ( const tableName String, const aliasName String ) Logical
```

Description

enumAliasLoginInfo writes information about the server alias specified in *aliasName* to the Paradox table specified in *tableName*. This method returns True if successful; otherwise, it returns False.

enumAliasLoginInfo operates on aliases that are stored in IDAPI.CFG and on new aliases opened and stored in system memory. This method fails if the table specified in *tableName* is already open.

enumAliasLoginInfo applies only to remote databases. Standard (Paradox or dBASE) databases are not affected by this method.

The following table displays the structure of the resulting *tableName* table:

Field	Type	Description
DBName	A32*	Specifies the database name
Property	A32*	Specifies the property name (e.g., OPEN MODE, NET PROTOCOL, SERVER NAME, and USER NAME)
PropertyValue	A82	Specifies the property value

Example

The following example calls **enumAliasLoginInfo** to write alias data about an alias to a Paradox table. The code then searches the table to test whether the OPEN MODE property for the alias is set to READ/WRITE. If OPEN MODE is set to READ/WRITE, the code calls a custom procedure named **doSomething** to continue processing; otherwise, the code displays information about properties and property values in a modal dialog box.

```
method pushButton(var eventInfo Event)

  var
    db           Database
    aliasInfoTC  TCursor
    aliasName,
    infoTableName,
    fieldName1,
```



```

        fieldName2,
        propName,
        propVal      String
        propValDA     DynArray[] AnyType
    endVar

    ; initialize variables
    aliasName = "itchy"
    infoTableName = "dbAlias.db"
    fieldName1 = "Property"
    fieldName2 = "PropertyValue"
    propName = "OPEN MODE"
    propVal = "READ/WRITE"

    ; open database, get alias info
    if db.open(aliasName) then
        if enumAliasLoginInfo(infoTableName, aliasName) then
            aliasInfoTC.open(infoTableName)

            ; search for info of interest
            if aliasInfoTC.locate(fieldName1, propName) then

                ; compare expected and actual values
                if aliasInfoTC.(fieldName2) propVal then

                    ; inform user if values don't match
                    propValDA["Property:"] = aliasInfoTC.(fieldName1)
                    propValDA["Expected value:"] = propVal
                    propValDA["Actual value:"] = aliasInfoTC.(fieldName2)
                    propValDA.view("Property mismatch")
                    return
                endif

            else
                errorShow("Property not found.")
                return
            endif
        else
            errorShow("Can't write to table: " + infoTableName)
            return
        endif
    else
        errorShow("Couldn't open " + aliasName)
        return
    endif

    doSomething() ; if property values are OK, continue processing
endMethod

```

enumAliasNames method/procedure

Session

Lists the database aliases available to a session.

Syntax

1. enumAliasNames (const *tableName* String [, const *LoginInfoTableName* String]) Logical
2. enumAliasNames (var *aliasNames* Array[] String) Logical

Description

enumAliasNames lists the database aliases available to a session.

Syntax 1 creates a Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory.

The following table displays the structure of *tableName*:

Field Name	Type	Description
DBName	A32*	Specifies the database alias name
DBType	A32	Specifies the driver type
DBPath	A82	Specifies the alias path

If you include the optional argument *LoginInfoTableName*, Paradox also writes login data to the table, just as if you had called **enumAliasLoginInfo**.

The structure of the resulting table is

Field	Type	Description
DBName	A32*	Specifies the database name
Property	A32*	Specifies the property name (e.g., OPEN MODE, NET PROTOCOL, SERVER NAME, and USER NAME)
PropertyValue	A82	Specifies the property value

Syntax 2 assigns the database names to items in an array named *aliasNames* that you declare and pass as an argument.

Example

In the following example, the **pushButton** method for *getAliasButton* writes the alias names for the active session to an array. If the array does not contain the name of a specified alias, **addAlias** adds it to the session.

```
; getAliasButton::pushButton
method pushButton(var eventInfo Event)
  var
    stAliasName,
    stAliasPath  String
    arAliasNames  Array[] String
  endVar

  stAliasName = "NewCust"
  stAliasPath = "g:\netdata\newcust"

  enumAliasNames(arAliasNames) ; List names to an array.
  if arAliasNames.contains(stAliasName) then
    return
  else
    addAlias(stAliasName, "STANDARD", stAliasPath)
  endIf
endMethod
```

enumDatabaseTables method/procedure**Session**

Lists the files in a database.

Syntax

```
1. enumDatabaseTables ( const tableName String, const databaseName String, const
   fileSpec String )
2. enumDatabaseTables ( var tableNames Array[ ] String, const databaseName String,
   const fileSpec String )
```

Description

enumDatabaseTables lists the files in a database specified by *databaseName*, where *databaseName* is an alias known to the session. *fileSpec* specifies a DOS file specification that can include the wildcard *.

Syntax 1 creates a Paradox table named *tableName*. If *tableName* already exists, **enumDatabaseTables** overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The structure of the table is

Field	Type	Description
DBName	A32*	Specifies the database alias
TableName	A32*	Specifies the table name (or the name of another file, depending on the file specification)

Syntax 2 assigns the table names to items in an array *tableNames* that you pass as an argument.

Example

The following example lists the Paradox and dBASE tables (and any other file whose extension is DB followed by 0 or 1 characters) in the private directory. This code **uses** **enumDatabaseTables** as a procedure and works in the active session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
  dbName,
  fileSpec,
  tbName    String
  tv1       TableView
endVar

; Init variables.
dbName    = ":PRIV:"
fileSpec  = "*.db" ; Lists .db
tbName    = "TabList"

enumDatabaseTables(tbName, dbName, fileSpec)
tv1.open(tbName) ; Open the created table.
endMethod
```

enumDriverCapabilities procedure**Session**

Lists the capabilities of the current driver.

enumDriverCapabilities procedure

Syntax

```
enumDriverCapabilities ( const drvCapName String, const tblCapName String, const  
fldCapName String [ , const inxCapName String ] ) Logical
```

Description

enumDriverCapabilities creates three Paradox tables that list the capabilities of the current driver. If these tables already exist, Paradox overwrites them without asking for confirmation. You can also include an alias or path in the specified table names. If an alias or path is not specified, Paradox creates the tables in the working directory.

Each supported table type is described by a record. Driver capabilities are written to a table named *drvCapName* which has the following structure:

Field	Type	Description
DriverType	A32*	Specifies the driver name (e.g., dBASE)
Description	A32	Describes the driver (e.g., dBASE driver)
Category	A32	Specifies the driver category
DB	A4	Specifies whether the driver supports a true database concept
DBType	A32	Specifies the database type to be used (e.g., STANDARD)
MultiUser	A4	Specifies whether the driver supports multi-user access
ReadWrite	A4	Specifies whether the driver is read-write
Transactions	A4	Specifies whether the driver supports transactions
PassThruSQL	A4	Specifies whether the driver supports pass-through SQL
Login	A4	Specifies whether the driver requires an explicit login (for example, to access a SQL server)
CreateDb	A4	Specifies whether the driver can create a database
DeleteDb	A4	Specifies whether the driver can delete a database
CreateTable	A4	Specifies whether the driver can create a table
DeleteTable	A4	Specifies whether the driver can delete a table
MultiPasswords	A4	Specifies whether the driver supports multiple passwords

Table capabilities are written to a table named *tblCapName* which has the following structure:

Field	Type	Description
DriverType	A32*	Specifies the table type (e.g., dBASE)
TableType	A32*	Describes the table type (e.g., PDOX 5.0)
Format	A32*	Specifies the table format (e.g., CLUSTERED)

ReadWrite	A4	Specifies whether the user can read from and write to the table
Create	A4	Specifies whether the user can create a table of this type
Restructure	A4	Specifies whether the user can restructure a table of this type
ValChecks	A4	Specifies whether the user can specify validity checks for a table of this type
Security	A4	Specifies whether the table can be password-protected
RefInt	A4	Specifies whether the table can participate in a referential integrity relationship
PrimaryKey	A4	Specifies whether the table supports primary keys
Indexing	A4	Specifies whether the table can have other (secondary) indexes
NoFieldType	A6	Specifies the number of physical field types supported
MaxRecSize	A6	Specifies the maximum record size (in bytes)
MaxFlds	A6	Specifies the maximum number of fields per record

Field capabilities are written to a table named *fldCapName*, which has the following structure:

Field	Type	Description
DriverType	A32*	Specifies the driver type (e.g., dBASE)
TableType	A32*	Specifies the table type (e.g., PDOX 5.0)
Format	A32*	Specifies the table format (e.g., CLUSTERED)
FieldType	A32*	Specifies the field type
Description	A32	Specifies the field type (e.g., Long integer)
NativeType	A6	Specifies the numeric value of native field type (e.g., 266)
XType	A6	Specifies the numeric value of translated field type (e.g., 3)
XSubType	A6	Specifies the numeric value of translated field subtype (e.g., 3)
MaxUnits1	A6	Specifies the maximum places to the left of the decimal point (or number of characters) (e.g., 240)
MaxUnits2	A6	Specifies the maximum places to the right of the decimal point (e.g., 19)
Size	A6	Specifies the field size (e.g., 8)

enumDriverCapabilities procedure

Required	A4	Specifies whether the field is a required field
Default	A4	Specifies whether the field has a specified default value
Min	A4	Specifies whether the field has a specified minimum value
Max	A4	Specifies whether the field has a specified maximum value
RefInt	A4	Specifies whether the field is part of a referential integrity relationship
Other	A4	Reserved
Key	A4	Specifies whether the field can be part of an index (keyed)
Multi	A4	Specifies whether the driver supports more than one of these fields per record
MinUnits1	A6	Specifies the minimum places to the left of the decimal point (or number of characters) (e.g., 240)
MinUnits2	A6	Specifies the minimum places to the right of the decimal point (e.g., 19)
Createable	A4	Specifies whether the driver can create a table using this field type

If you include an optional argument named *inxCapName*, index capabilities are written to the table specified in *inxCapName*. *inxCapName* has the following structure:

Field	Type	Description
DriverType	A32*	Specifies the driver type (e.g., dBASE)
TableType	A32*	Specifies the table type (e.g., PDOX 5.0)
Format	A32*	Specifies the table format (e.g., CLUSTERED)
Name	A32*	Specifies an internal name describing the type of index (e.g., SECONDARY) to correspond with the description in the Description field
Format1	A32*	Specifies the index format (e.g., BTREE)
Description	A32	Describes the index (e.g., Non-maintained Secondary index)
Composite	A4	Specifies whether the index supports composite keys
Primary	A4	Specifies whether the index is a primary index
Unique	A4	Specifies whether the index is a unique index
keyDescending	A4	Specifies whether the whole key can be descending
fldDescending	A4	Specifies whether the index is field level descending
Maintained	A4	Specifies whether the index is a maintained index

Subset	A4	Specifies whether the index is a subset index
KeyExp	A4	Specifies whether the index is an expression index
CaseInsensitive	A4	Specifies whether the index is insensitive to case

Example

In the following 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
```

Categories for enumDriverCapabilities (Session type)

Value In the following 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
```

Value	Description
File	File-based (Paradox, dBASE)
SQL Server	SQL-based server
Other Server	A server that is not file or SQL-based

Field types for enumDriverCapabilities (Session type)

The following tables display the field types for Paradox and dBASE tables:

Paradox field type	Return value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY

enumDriverInfo procedure

Bytes	BYTES
Date	DATE
FmtMemo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
LongInt	LONG
Memo	MEMO
Money	MONEY
Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
TimeStamp	TIMESTAMP
dBASE field type	Return value
BINARY	BINARY
CHAR	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

enumDriverInfo procedure

Session

Lists information about available drivers.

Syntax

```
enumDriverInfo ( const tableName String )
```

Description

enumDriverInfo lists information about available driver types in a table named *tableName*. If *tableName* already exists, Paradox overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of *tableName*:

Field	Type	Description
DriverType	A32*	Specifies the driver type or name (e.g., PARADOX)
Topic	A32*	Specifies the driver function (e.g., TABLE CREATE)
Property	A32*	Specifies the property of corresponding driver function (e.g., BLOCK SIZE)
PropertyValue	A68	Specifies the value of corresponding property (e.g., 2048)

Example

The following example enumerates driver information to a table named *DriveInf* and displays the results:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tv1 TableView
endVar
; create and view the DriveInf table
enumDriverInfo("DriveInf")
tv1.open("DriveInf")
endMethod
```

enumDriverNames method/procedure**Session**

Creates a Paradox table listing the names of available drivers.

Syntax

```
enumDriverNames ( const tableName String )
```

Description

enumDriverNames writes the available driver names to *tableName*. If *tableName* already exists, Paradox overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The structure of the table is DriverType, A32*.

Example

The following example enumerates available driver names to a table named *DrivName* and displays the results:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tv1 TableView
endVar
; create and view the DrivName table
enumDriverNames("DrivName")
tv1.open("DrivName")
endMethod
```

enumDriverTopics procedure**Session**

Lists the topics currently available for each driver type.

enumEngineInfo procedure

Syntax

```
enumDriverTopics ( const tableName String )
```

Description

enumDriverTopics writes the driver topics available for each driver type to a table named *tableName*. If *tableName* already exists, Paradox overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of *tableName*:

Field name	Type	Description
DriverType	A32*	Specifies the driver type or name (e.g., PARADOX)
Topic	A32*	Specifies the driver function For Paradox and dBASE tables, the topics are INIT and TABLE CREATE

Example

The following example enumerates available driver topics to a table named *DrivTop* and displays the results:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tv1 TableView
endVar
; create and view the DrivTop table
enumDriverTopics("drivtop")
tv1.open("drivtop")
endMethod
```

enumEngineInfo procedure

Session

Creates a Paradox table listing the current Borland Database Engine (BDE) engine properties.

Syntax

```
enumEngineInfo ( const tableName String )
```

Description

enumEngineInfo creates a Paradox table that describes the contents of the BDE System Information dialog box. Each setting name and value is written to a record in a table named *tableName*. If *tableName* already exists, Paradox overwrites it without asking confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of *tableName*:

Field name	Type	Description
Property	A32*	Specifies the engine property
PropertyValue	A68	Specifies the value of corresponding property

Example

The following example enumerates engine information to a table named *EngInf* and displays the results:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tv1 TableView
endVar
enumEngineInfo("EngInf")
tv1.open("EngInf")
endMethod
```

Properties for enumEngineInfo

Engine property	Description
LANGDRIVER	Specifies the name of language driver (e.g., ASCII)
LANGDRVDIR	Specifies the language driver folder
LOCAL SHARE	Specifies whether the Local Share is active
MAXBUFSIZE	Specifies the maximum buffer size (in bytes)
MAXFILEHANDLES	Specifies the maximum number of file handles
MINBUFSIZE	Specifies the minimum buffer size (in bytes)
NET DIR	Specifies the path to NET folder
NET TYPE	Specifies the network type
SYSFLAGS	Specifies the number of system flags
VERSION	Specifies the BDE version number

enumFolder procedure**Session**

Lists the names of files in a folder or project.

Syntax

1. enumFolder (const *tableName* String [, const *fileSpec* String]) Logical
2. enumFolder (var *result* Array[] String [, const *fileSpec* String]) Logical

Description

enumFolder lists the names of files in a folder or project. By default, a project includes all the objects in :WORK: and :PRIV:. You can also add references to objects in other directories.

Syntax 1 creates a Paradox table named *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

Syntax 2 lists the files in an array named *result* which you must declare and pass as an argument. For each file, the array lists the filename (and extension, if one exists), and includes the path if the file is not in the working directory.

You can list files using a particular extension using an optional argument named *fileSpec*. For example, to list all forms in a file, specify .FSL *infileSpec*.

enumOpenDatabases method/procedure

The structure of the table created by Syntax 1 is

Field	Type	Description
Name	A128	Specifies the filename (and extension, if one exists). Includes the path if the file is not in :WORK:.
LocalName	A68	Specifies the filename without extension. Includes the path if the file is not in :WORK:.
IsReference	A4	Specifies whether the filename refers to a file in a directory other than :WORK:.
IsPrivate	A4	Specifies whether the filename refers to a file in :PRIV:.
IsTemp	A4	Reserved
Position	A10	Reserved

Example

In the following example, the method prompts the user to type a file specification (e.g., *.FSL). The file specification entered is then used by **enumFolder** to create a table listing the files that match the specification.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    filespec String
    tv1      TableView
endVar
filespec.view("Enter filename specification")
enumFolder("PartCat", filespec)
message("Table lists files that match your specification.")
tv1.open("PartCat")
endMethod
```

enumOpenDatabases method/procedure

Session

Lists the open databases.

Syntax

- 1. enumOpenDatabases (const *tableName* String) Logical
- 2. enumOpenDatabases (var *tableNames* Array[] String) Logical

Description

enumOpenDatabases lists the databases open in the active session.

Syntax 1 creates a Paradox table named *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of the resulting table:

Field name	Type	Description
DBName	A32*	Specifies the database alias name

DBType	A32	Specifies the database driver type
ShareMode	A32	Specifies the database share mode
OpenMode	A32	Specifies the database open mode

Syntax 2 writes the data to an array *tableNames* that you declare and pass as an argument.

Example

In the following example, **enumOpenDatabases** creates a table named OPENDB.DB, and then displays the table.

```
; btnOpenDB :: pushButton
method pushButton(var eventInfo Event)
  var
    tv    TableView
  endVar

  enumOpenDatabases("OPENDB.DB")
  tv.open("OPENDB.DB")
endMethod
```

enumUsers procedure

Session

Creates a Paradox table listing all known users with an open channel to the Borland Database Engine (BDE) engine.

Syntax

1. **enumUsers** (const *tableName* String) LongInt
2. **enumUsers** (var *userNames* Array[] String) LongInt

Description

enumUsers creates a list of all users with an open path to the BDE database engine.

Syntax 1 creates a table named *tableName* that lists all users with an open path to BDE. If *tableName* already exists, Paradox overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of the resulting table:

Field Name	Type	Description
UserName	A15	Specifies the network user name
NetSession	N	Specifies the network session number
ProductClass	N	Specifies the user's product class ID number
SerialNumber	A22	Specifies the serial number (version 1.0 only)

Syntax 2 lists the network names of users who currently have an open path to BDE in an array. You must declare the array before calling this procedure.

Example

The following example writes information about current users to a table named *Users* and displays the table:

getAliasPath method/procedure

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tv1 TableView
endVar
    enumUsers("users")
    tv1.open("users")
endMethod
```

getAliasPath method/procedure

Session

Returns the path for a specified alias.

Syntax

```
getAliasPath ( const aliasName String ) String
```

Description

getAliasPath returns the path for an alias named *aliasName*.

Example

The following example prompts the user for an alias name and displays the corresponding path:

```
; getShowPath::pushButton
method pushButton(var eventInfo Event)
var
    stPrompt,
    stAliasName,
    stCurrentPath,
    stMyPath      String
endVar

stPrompt      = "Enter an Alias Name."
stAliasName   = stPrompt
stMyPath      = "d:\\Core1\\Paradox\\data"

stAliasName.view(stPrompt) ; prompt for an alias name
if stAliasName = stPrompt then
    return ; User didn't click the OK button.
else
    stCurrentPath = getAliasPath(stAliasName) ; get the path
endIf

if stCurrentPath = stMyPath then
    return
else
    setAliasPath(stAliasName, stMyPath)
endIf
endMethod
```

getAliasProperty method

Session

Returns the property value for a specified server alias.

Syntax

```
getAliasProperty ( const aliasName String, const property String ) String
```

Description

getAliasProperty returns a string representing the *property* value specified by property for the server alias specified in *aliasName*. If the property is not valid for the alias, this method returns an error.

getAliasProperty operates on aliases stored in IDAPI.CFG and on new aliases that have been opened and stored in system memory.

This method only applies to remote databases, and not to standard (Paradox or dBASE) databases.

Example

The following example uses **getAliasProperty** to retrieve the value of the OPEN MODE property. This code compares the returned (actual) value with the expected value. If the returned and expected values match, the code calls a custom procedure named doSomething to continue processing. If the returned and expected values do not match, the code informs the user of a property mismatch and calls **setAliasProperty** to set the property to the expected value.

```
method pushButton(var eventInfo Event)

var
    db                Database
    aliasName,
    propName,
    expectedPropVal,
    actualPropVal      String
    propValDA          DynArray[] AnyType
endVar

; initialize variables
aliasName = "itchy"
propName = "OPEN MODE"
expectedPropVal = "READ/WRITE"

if db.open(aliasName) then

    ; get property value and compare with expected value
    actualPropVal = getAliasProperty(aliasName, propName)
    if actualPropVal = expectedPropVal then
        doSomething() ; continue processing
        return
    else

        ; inform the user if there's a mismatch
        propValDA["Property name"] = propName
        propValDA["Expected value"] = expectedPropVal
        propValDA["Actual value"] = actualPropVal
        propValDA.view("Property mismatch:")

        ; let user decide what to do
        if msgQuestion("Set property value?",
            "Set "+propName+" to " + expectedPropVal + "?" ) = "Yes" then

            ; set property to expected value and continue processing
            if setAliasProperty(aliasName, propName, expectedPropVal) then
                doSomething() ; Continue processing
                return
            else
                errorShow("Couldn't set property value.",
                    "Operation canceled.")
                return
        endif
    endif
endif
```

getNetUserName method/procedure

```
        else
            msgInfo("Operation canceled.", "Property not set.")
            return
        endIf

    endIf

    else
        msgStop(aliasName, "Couldn't open database.")
        return
    endIf

endMethod
```

getNetUserName method/procedure

Session

Returns the name of the current network user.

Syntax

```
getNetUserName ( ) String
```

Description

getNetUserName returns the name of the current network user.

Example

The following 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
```

ignoreCaseInLocate procedure

Session

Specifies whether to ignore case-sensitivity in locate operations.

Syntax

```
ignoreCaseInLocate ( [ const yesNo Logical ] )
```

Description

ignoreCaseInLocate specifies whether the active session ignores case-sensitivity during locate operations. If an optional argument named *yesNo* is set to Yes or omitted, all subsequent locate operations ignore case in string comparisons. If *yesNo* is set to No, locate operations will respect case.

Example

The following example calls **ignoreCaseInLocate** to prepare for a call to the **locate** method:

```
; findName::pushButton
method pushButton(var eventInfo Event)
var
    tc          TCursor
    loIgnoreCase Logical
endVar

if tc.open("Customer.db") then

    loIgnoreCase = isIgnoreCaseInLocate() ; Get user's setting.
```



```

if loIgnoreCase then
    ; locate values based on value as entered
    ; (do not ignore case in string compares)
    ignoreCaseInLocate(No)
endIf

; search for case-sensitive MacAnaly in Name field
if tc.locate("Name", "MacAnaly") then
    tc.edit()
    tc.Name = "Macanally"
    tc.endEdit()
else
    message("Couldn't find MacAnaly...")
endIf

ignoreCaseInLocate(loIgnoreCase) ; Restore user's setting.

else
    msgStop("Error", "Can't open Customer table.")
endIf

endMethod

```

isAdvancedWildcardsInLocate procedure

Session

Reports whether the active session is using advanced wildcards during locate operations.

Syntax

```
isAdvancedWildcardsInLocate ( ) Logical
```

Description

isAdvancedWildcardsInLocate reports whether the active session is using advanced wildcards during locate operations that include pattern strings.

Example

The following example calls **advancedWildcardsInLocate** to specify that advanced wild cards can be used in a locate operation. The code the calls to **locatePattern**, which 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")
    endIf
endIf

```

isAssigned method

```
    else
        msgStop("Error", "Can't find record")
    endIf
else
    msgStop("Error", "Can't open Orders table.")
endIf

endMethod
```

isAssigned method

Session

Reports whether a Session variable is assigned.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned reports whether a Session variable is assigned.

Example

See the **close** example.

isBlankZero method/procedure

Session

Reports whether blank values are treated as zero in calculations.

Syntax

```
isBlankZero ( ) Logical
```

Description

isBlankZero returns True if blank fields are treated as zero in calculations, or as filled fields in counting calculation (e.g., **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 **blankAsZero** example.

isIgnoreCaseInLocate procedure

Session

Reports whether the active session ignores case-sensitivity in locate operations.

Syntax

```
isIgnoreCaseInLocate ( ) Logical
```

Description

isIgnoreCaseInLocate reports whether the active session ignores case-sensitivity during locate operations.

Example

See the **ignoreCaseInLocate** example.

loadProjectAliases procedure

Session

Loads project alias specifications.

Syntax

```
loadProjectAliases ( const cfgFileName String ) Logical
```

Description

loadProjectAliases loads project alias specifications from the file specified in *cfgFileName*. If *cfgFileName* does not specify a path, Paradox searches for the file in the working directory. Paradox automatically reads project aliases from :WORK:PDOXWORK.CFG. This method lets you specify a different file.

When :WORK: is set (e.g., at startup) or changed (interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available. If a project alias has the same name as a public alias, Paradox does not load the project alias. This method returns True if it succeeds; otherwise, it returns False.

Example

The following example loads the project aliases in the **open** method of the form's first page. This code reads a list of custom aliases from C:\COREL\PARADOX\CUSTOM.CFG instead of from the Paradox default configuration file.

```
;pgel :: open
method open(var eventInfo Event)
    loadProjectAliases("C:\\COREL\\VITE8\\PARADOX\\CUSTOM.CFG")
endMethod
```

lock procedure**Session**

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

lock locks one or more of the tables specified in comma-separated pairs of tables and lock types. You can use a TCursor or a Table to specify a table. You can mix TCursor and Table variables in the list.

The following *lockType* values are listed in order of decreasing strength and increasing concurrency:

String value	Description
Full	Specifies whether the active session has exclusive access to the table. Cannot be used with dBASE tables.
Write	Specifies whether the active session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	Specifies whether the active session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If **lock** locks all the tables in the list, it returns True; otherwise, it returns False. If lock can't lock all the tables, it doesn't lock any.

open method

Example

The following example attempts to place a write lock on the **Orders** table and a read lock on the **Customer** table. If **lock** is able to lock both tables, the code displays data from both tables in a dialog box. The code then 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
    otherSes    Session
endVar

otherSes.open("other") ; Open another session
otherSes.addAlias("samples", "Standard", "c:\\Core1\\Paradox\\samples")
sampDB.open("samples", otherSes)

custTC.open("Customer.db", sampDB)
ordTB.attach("Orders.db", sampDB)

if lock(custTC, "Read", ordTB, "Write") then
    if custTC.locate("Name", "Unisco") then
        custNo = custTC."Customer No"
        ordTB.setIndex("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
    errorShow()
endIf

endMethod
```

open method

Session

Opens a session (a channel to the database engine).

Syntax

1. `open () Logical`
2. `open (const sessionName String) Logical`

Description

open opens a session (a channel to the database engine). Calling **open** with no arguments (Syntax 1) gives you a handle to the active session; it does not exhaust a user count. When you use *sessionName* to specify a session name (Syntax 2), you open another channel to the database engine and exhaust one user count. The *sessionName* value can be any valid string.

If you open multiple sessions from the same workstation, Paradox views each session as a separate user (e.g., locks set in one session block access from the other).

Example

The following example calls **open** to retrieve a handle to the active session, and to open a new session. The code then calls **blankAsZero** to specify how each session handles blank values in calculations.

Finally, the code passes the Session variables to a custom procedure named doSomething. Because different sessions have different **blankAsZero**, the results of doSomething vary.

```
; openSession::pushButton
method pushButton(var eventInfo Event)
var
    currentSes,
    otherSes    Session
endVar

; Open sessions.
currentSes.open()
otherSes.open("other")

; Set session properties.
currentSes.blankAsZero(Yes)
otherSes.blankAsZero(No)

; Pass session handles to a custom procedure.
; Results will differ depending on settings for each session.
doSomething(currentSes)
doSomething(otherSes)

endMethod
```

removeAlias method/procedure

Session

Removes an alias from a session.

Syntax

```
removeAlias ( const aliasName String ) Logical
```

Description

removeAlias removes the alias *aliasName* from a session. You cannot remove :WORK:, :PRIV:, or an open alias.

Example

The following example adds an alias to the active session and makes the new alias available to the **open** method defined for the Database type. When the alias is no longer needed, this code calls **removeAlias** to remove the alias from the active session.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    custInfo Database
endVar

; Add the CustomerInfo alias to the current session.
addAlias("CustomerInfo", "Standard", "D:\\Corel\\Paradox\\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, close the
; database and remove the alias from the current session.

custInfo.close()
removeAlias("CustomerInfo")

endMethod
```

removeAllPasswords method/procedure**Session**

Removes passwords defined for a session.

Syntax

```
removeAllPasswords ( )
```

Description

removeAllPasswords removes passwords defined for a session. This method withdraws the passwords required to access protected tables, but does not remove security from tables.

Example

The following example removes all the passwords from the session named 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
```

removePassword method/procedure**Session**

Removes a password defined for a session.

Syntax

```
removePassword ( const password String )
```

Description

removePassword removes a password defined for a session. This method withdraws the password specified in the argument *password*, but does not unprotect the table. *password* is case-sensitive.

Example

In the following example, the *getRemovePass* button acquires a password to remove from the user and removes the password from the active session. Subsequent attempts to open tables protected by that password 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
```

removeProjectAlias procedure**Session**

Removes a project alias. For information about aliases, see Aliases in the Paradox online Help.

Syntax

```
removeProjectAlias ( const alias String ) Logical
```

Description

removeProjectAlias removes the project alias specified in *alias*.

When the working directory is set (e.g., at startup) or changed (interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available. If a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG; however, you can use **loadProjectAliases** to specify a different path and file.

Example

The following example uses **addProjectAlias** in the page's built-in **arrive** method to add an alias to the current project. The code then uses **removeProjectAlias** in the page's built-in **depart** method to remove the alias.

The following code is attached to the page's built-in **arrive** method:

```
;pgel :: arrive
method arrive(var eventInfo MoveEvent)
    ;Add the CustomerInfo alias to the project.
    addProjectAlias("CustomerInfo", "Standard", "D:\\COREL\\PARADOX\\SAMPLES")
endMethod
```

The following code is attached to the page's built-in **depart** method.

```
;pgel :: depart
method depart(var eventInfo MoveEvent)
    ;Remove the CustomerInfo alias from the project.
    if not removeProjectAlias("CustomerInfo") then
        errorShow("Could not remove project alias CustomerInfo.")
    endIf
endMethod
```

retryPeriod method/procedure**Session**

Returns the number of seconds allowed to retry an operation on a locked record or table.

Syntax

```
retryPeriod ( ) SmallInt
```

Description

retryPeriod returns the number of seconds allowed to retry an operation on a locked record or table. If the **retryPeriod** is set to 0 (default), operations are not retried.

Example

The following example displays the current retry period:

```
; getShowRetry::pushButton
method pushButton(var eventInfo Event)
var
    rp smallint
endVar
```

saveCFG method/procedure

```
; 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
```

saveCFG method/procedure

Session

Saves the active session's alias information to a file.

Syntax

```
saveCFG ( [const fileName String] ) Logical
```

Description

saveCFG saves the BDE configuration for the active session in *fileName*. The configuration file specified by *fileName* can be loaded using the -o command-line option to set session information at startup.

Note

- In the absence of a filename, changes are saved to the current session's configuration file.

Example

The following example saves the current BDE settings to MyConfig.cfg.

```
; saveconfiguration::pushButton
method pushButton(var eventInfo Event)
; saves the BDE setting to file MyConfig.cfg
saveCfg("MyConfig.cfg")
endMethod
```

saveProjectAliases procedure

Session

Saves project alias specifications to a file. For information about aliases, see Aliases in the Paradox online Help.

Syntax

```
saveProjectAliases ( [ const fileName String ] ) Logical
```

Description

saveProjectAliases saves project alias specifications to a file. You can use the optional argument *fileName* to specify a filename. If you omit *fileName*, Paradox saves the alias to :WORK:PDOXWORK.CFG.

When :WORK: is set (e.g., at startup) or changed (interactively or through ObjectPAL), Paradox discards all current project aliases and loads those project aliases that are specific to the new working directory. Public aliases remain active and available. If a project alias has the same name as a public alias, Paradox does not load the project alias. By default, Paradox reads project aliases from :WORK:PDOXWORK.CFG; however, you can use **loadProjectAliases** to specify a different path and file.

Example

The following example uses **saveProjectAliases** to save new project aliases to MYPROJ.CFG:


```

;pgel :: open
method open(var eventInfo Event)
    ; Add project alias.
    addProjectAlias("MYPROJ", "Standard", "D:\\COREL\\PARADOX\\MYPROJ")

    ; Save project aliases.
    saveProjectAliases("MYPROJ.CFG")
endMethod

```

setAliasPassword method/procedure

Session

Sets the in-memory password for a specified alias.

Syntax

```
setAliasPassword ( const aliasName, const password String ) Logical
```

Description

setAliasPassword sets the in-memory password for the alias specified in *aliasName* to the value specified in *password*. Passwords have a maximum length of 31 characters. The next time you open that alias, you do not have to supply the password.

Calling **setAliasPassword** has the same effect as defining a password interactively using the Alias Manager dialog box. **setAliasPassword** has no effect on the password stored and maintained on the server. This method returns True if successful; otherwise, it returns False.

Example

The following example calls **setAliasPassword** to define the password for a specified alias. When the call to open executes, this code opens the database without prompting the user for a password.

```

method pushButton(var eventInfo Event)
    var
        aliasName,
        aliasPassword String
        db Database
    endVar

    ; initialize variables
    aliasName = "bedrock"
    aliasPassword = "fred" ; Max length: 31 characters

    ; set alias password and open database
    if setAliasPassword(aliasName, aliasPassword) then
        db.open(aliasName) ; opens without prompting for password
    else
        errorShow("Couldn't set alias password.")
        return
    endIf
endMethod

```

setAliasPath method/procedure

Session

Sets the path for an alias.

Syntax

```
setAliasPath ( const aliasName String, const aliasPath String ) Logical
```

setAliasProperty method

Description

setAliasPath sets the path *aliasPath* for the alias *aliasName*.

Example

See the **getAliasPath** example.

setAliasProperty method

Session

Sets the value of a specified property for a specified alias.

Syntax

```
setAliasProperty ( const aliasName String, const property String, const propertyValue String ) Logical
```

Description

setAliasProperty sets the value specified in *property*, to the value specified in *propertyValue*, for the alias specified in *aliasName*. This method returns True if successful; otherwise, it returns False.

Properties that you set using this method are displayed in the Alias Manager dialog box. Since property settings are *not* automatically saved to IDAPI.CFG, you must use the Session procedure **saveCFG** to save alias properties to a file.

This method applies only to remote databases, and not to standard (Paradox or dBASE) databases.

Example

See the **getAliasProperty** example.

setRetryPeriod method/procedure

Session

Sets the number of seconds allowed to retry an action on a locked table or record.

Syntax

```
setRetryPeriod ( const period SmallInt ) Logical
```

Description

setRetryPeriod specifies the number of seconds to allowed retry an action on a locked table or record. If you set setRetryPeriod to 0, actions are not retried.

Example

The following example prompts the user to specify a retry period and sets the session's retry period 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
```

unlock procedure**Session**

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

unlock unlocks one or more of the tables specified in a comma-separated list of tables and lock types.

unlock removes locks explicitly placed by a particular user or application but does not affect locks placed automatically by Paradox. The *lockType* value must be one of the following: Exclusive, Write, Read, or Full. Read and Full apply only to Paradox tables.

If one **unlock** attempt fails, previous locks are not restored - the tables remain unlocked. You don't have to specify a session in which to use this method, because session data is set when you **open** a TCursor or **attach** to a Table.

To ensure maximum concurrent availability of tables, unlock tables when the lock is no longer required. When you lock a table twice, you must unlock it twice. You can use the **lockStatus** method (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. If you try to unlock a table that isn't locked or cannot be unlocked, **unlock** returns False.

Example

See the **lock** example.

SmallInt type

SmallInt values are small integers that can be represented by a short series of digits. A SmallInt variable occupies 2 bytes of storage.

ObjectPAL converts SmallInt values to range from -32,768 to 32,767. If you attempt to assign a value outside of this range to a SmallInt variable, an error occurs.

```
var
    x, y, z SmallInt
endVar

x = 32767 ; The upper limit value for a SmallInt variable.
y = 1
z = x + y ; This statement causes an error.
```

When ObjectPAL performs an operation on SmallInt values, the result must also be a SmallInt value. To work with a boundary value (in either the positive or negative direction), convert it to a type that can accommodate it. In the following example, ObjectPAL converts a SmallInt to a LongInt before performing the addition. The result is assigned to a LongInt variable which can handle the large value.

```
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 = LongInt(x) + y
```

Notes

- The SmallInt value -32,768 cannot be stored in a Paradox table. Paradox considers -32,768 to be a blank. This value can be used in calculations and stored in a dBASE table. Store such large numbers as LongInt or Number data types.
- 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 returns a Number value, even though sin does not appear in the methods for the SmallInt type:

```
var
    abc LongInt
    xyz Number
endVar
abc = 43
xyz = abc.sin()
```

The SmallInt type includes several derived methods from the Number and AnyType types.

Methods for the SmallInt type

AnyType	←	Number	←	LongInt
blank		abs		bitAND
dataType		acos		bitIsSet
isAssigned		asin		bitOR
isBlank		atan		bitXOR
isFixedType		atan2		int
view		ceil		smallInt

cos
cosh
exp
floor
Fraction
fv
Ln
Log
max
min
mod
number
numVal
pmt
pow
pow10
pv
rand
round
sin
sinh
sqrt
tan
tanh
truncate
numbers

bitAND method

SmallInt

Performs a bitwise AND operation on two values.

Syntax

bitAND (const *value* SmallInt) SmallInt

Description

bitAND 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

bitIsSet method

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
```

bitIsSet method

SmallInt

Reports whether a bit is 1 or 0.

Syntax

```
bitIsSet ( const value SmallInt ) Logical
```

Description

bitIsSet examines the binary representation of an integer and reports whether the **value** bit is 0 or 1. This method returns True if the bit is 1, and False if it 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 position less than the position of the bit to test, counting from the right. For example, to specify the third bit from the right, use .

Example I

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 and assigns the result to *bitNum*. This 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
```

Example 2

The following 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
```

bitOr method**SmallInt**

Performs a bitwise OR operation on two values.

Syntax

```
bitOR ( const value SmallInt ) SmallInt
```

Description

bitOR performs 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

bitXOR method

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 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
```

bitXOR method

SmallInt

Performs a bitwise XOR operation on two values.

Syntax

```
bitXOR ( const value SmallInt ) SmallInt
```

Description

bitXOR 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 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
```

int procedure

SmallInt

Casts a value as an integer.

Syntax

```
int ( const value AnyType ) SmallInt
```

Description

int casts the numeric expression *value* to an integer. If *value* is of a more precise type (e.g., Number), precision is lost.

Example

The following example assigns a number to *nn*, views the value of *nn* in a dialog box and 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
```

smallInt procedure**SmallInt**

Casts a value as a small integer.

Syntax

```
smallInt ( const value AnyType ) SmallInt
```

Description

smallInt casts the numeric expression *value* to a SmallInt. If *value* is of a more precise type (e.g., Number), precision is lost.

Example

The following example assigns a number to *x*, casts *x* to SmallInt, and assigns the result to *s*. The decimal precision of *x* is lost when it is cast as 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
```

SQL type

An ObjectPAL SQL variable represents an SQL statement. You can use ObjectPAL to create and execute SQL commands from methods in the same way that you create and execute SQL commands interactively. SQL commands can be executed from an SQL file, an SQL statement, or a string. Some queries require Paradox to create temporary tables in the private directory.

executeSQL method/procedure

Methods for the SQL type

executeSQL
getQueryRestartOptions
isAssigned
readFromFile
readFromString
setQueryRestartOptions
wantInMemoryTCursor
writeSQL

executeSQL method/procedure

SQL

Executes an SQL statement.

Syntax

Method:

1. executeSQL (const *db* Database) Logical
2. executeSQL (const *db* Database, CONST *ansTbl* String) Logical
3. executeSQL (const *db* Database, VAR *ansTbl* Table) Logical
4. executeSQL (const *db* Database, VAR *ansTbl* TCursor) Logical

Procedure:

1. executeSQL (const *db* Database, const *qbeVar* SQL) Logical
2. executeSQL (const *db* Database, const *qbeVar* SQL, *ansTbl* String) Logical
3. executeSQL (const *db* Database, const *qbeVar* SQL, *ansTbl* Table) Logical
4. executeSQL (const *db* Database, const *qbeVar* SQL, *ansTbl* TCursor) Logical

Description

executeSQL executes a pass through SQL query created in an ObjectPAL method or procedure.

In Syntax 1 the answer table is not specified. **executeSQL** writes to ANSWER.DB in the private directory.

In Syntax 2 the answer table is specified as a string. If you do not include a file extension, the answer table is a Paradox table by default.

In Syntax 3 *ansTbl* is a Table variable. *ansTbl* must be assigned and valid.

In Syntax 4 a TCursor is opened onto the answer set. The TCursor may be an in-memory table or a cursor onto the answer set.

executeSQL returns True if the query is executed on the server (even if the resulting table is empty); otherwise, it returns False.

An SQL query in ObjectPAL code begins with an SQL variable, the = sign, and the keyword SQL followed by a blank line. The code continues with the SQL statements that make up the body of the query, followed by another blank line. The query ends with the keyword **endSQL**. Because this query is not a quoted string, it can contain tilde variables.

Note

- **executeSQL** is a pass through function. The SQL statements are sent directly to the server as if by another user. SQL statements do not execute within the context of a database handle or active transaction.

Example 1

```

method pushButton(var eventInfo Event)
var
    itemNameSQL SQL
    ViewName tableview
    db database
endVar

db.open (":sample:")      ; this will open the connection to the local table or SQL
table via the alias

itemNameSQL =             ; the following stores the SQL statement into the variable
SQL
Select * from Biolife.db
endSQL
executeSQL(db, itemNameSQL, ":sample:myanswer.DB") ;execute the SQL statement into an
answer table
ViewName.open(":sample:myanswer.DB") ;displays the answer table
endMethod

```

Example 2

```

method pushButton(var eventInfo Event)
var
    itemNameSQL SQL
    ViewName tableview
    db database
endVar

db.open (":sample:")      ;this will open the connection to the local table or SQL
table via the alias

                                ;for the following create a SQL file within the sample alias
and insert Select * from      ;Biolife.db and save the file with the name
                                Embedded_SQL_File.SQL
itemNameSQL.readfromfile("Embedded_SQL_File.SQL") ;reads the SQL file located in the
sample alias and

;stores the SQL statement into the variable
executeSQL(db, itemNameSQL, ":sample:myanswer.DB") ;execute the SQL statement into an
answer table
ViewName.open(":sample:myanswer.DB")      ;displays the answer table
endMethod

```

isAssigned method**SQL**

Reports whether an SQL variable has an assigned value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned returns True if an SQL variable has been assigned a value; otherwise, it returns False.
isAssigned does not determine if the assigned SQL statement is valid.

readFromFile method

Example

In the following example, the call to **isAssigned** returns True. The SQL variable *sqlVar* has been assigned a value even though the value is not a valid SQL variable.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    sqlVar SQL
endVar
sqlVar = SQL
    This is not a valid SQL statement
endSQL
msgInfo("Assigned?", sqlVar.isAssigned())    ; displays True
endMethod
```

readFromFile method

SQL

Assigns the contents of an SQL file to an SQL variable.

Syntax

```
readFromFile ( const sqlFileName SQL ) Logical
```

Description

readFromFile assigns the contents of *sqlFileName* to an SQL variable. *SqlFileName* is created with **writeSQL** or interactively with the **SQL** Editor. Do not use the SQL and **endSQL** keywords. Use **executeSQL** to execute the query.

If *fileName* does not include a path or alias, **readFromFile** searches for the file in the directory associated with the specified database (or the default database, if a database is not specified). If the value of *fileName* does not include an extension, **readFromFile** assumes an extension of .SQL. To specify a filename that does not have an extension, type a period after the name. The following table lists the filenames different *fileName* values:

fileName value	SQL filename
newcust	newcust.sql
newcust.	newcust
newcust.s	newcust.s

readFromFile returns True if it succeeds; otherwise, it returns False.

Note

- **readFromFile** is a pass through function. The SQL statements are sent directly to the server as if by another user. SQL statements do not execute within the context of a database handle or active transaction.

Example

The following example creates a pop-up menu listing the SQL files stored in the private directory. When the user chooses a file from the menu, this code calls **readFromFile**. **readFromFile** reads the query, assigns it to an SQL variable, executes the query, and stores the results in a TCursor. The code then passes the TCursor to a custom procedure (assumed to be defined elsewhere) for additional processing.

```

method pushButton(var eventInfo Event)
var
    myAlias,
    aliasTableName,
    sqlFileName,
    sqlFileSpec      String
    aliasNamTC,
    answerTC          TCursor
    sqlPop            PopUpMenu
    db                Database
    sqlFS             FileSystem
    sqlFileAr         Array[] String
    sqlVar            SQL
endVar

; initialize variables
myAlias = "itchy"
aliasTableName = ":PRIV:aliasNam.db"
sqlFileSpec = ":PRIV:*.SQL"

enumAliasNames(aliasTableName) ; create a table of aliases

aliasNamTC.open(aliasTableName)
if aliasNamTC.locate("DBName", myAlias) then
    db.open(myAlias) ; use alias to get database handle to server
else
    msgStop("Stop",
        "The alias " + myAlias +
        " has not been defined.")
    return ; exit the method
endIf

; build a pop-up menu listing SQL files in the target directory
if sqlFS.findFirst(sqlFileSpec) then
    sqlFS.enumFileList(sqlFileSpec, sqlFileAr)
    sqlPop.addArray(sqlFileAr)
    sqlFileName = sqlPop.show() ; variable stores user's menu choice

    ; read and execute the SQL file chosen by the user
    sqlVar.readFromFile(sqlFileName)
    if sqlVar.executeSQL(db,answerTC) then
        doSomething(answerTC) ; call custom proc to process data
    else
        errorShow("readFromFile failed")
    endIf
endIf

else
    msgStop("File not found:", sqlFileSpec)
endIf

endMethod

```

readFromString method

SQL

Assigns a query string to an SQL variable.

Syntax

```
readFromString ( const sqlString SQL ) Logical
```

readFromString method

Description

readFromString assigns the SQL query string specified in *sqlString* to an SQL variable. Do not enclose the string between the *SQL* and **endSQL** keywords. Use **executeSQL** to execute the query.

Notes

- **readFromFile** is a pass through function. The SQL statements are sent directly to the server as if by another user. SQL statements do not execute within the context of a database handle or active transaction.

Example

The following example prompts the user to type an SQL keyword and uses that keyword in an SQL string. If the user enters a valid SQL keyword and the query executes successfully, the results are stored in a TCursor and passed to a predefined custom procedure for additional processing.

```
method pushButton(var eventInfo Event)
var
    sqlKeyword,
    promptString,
    bigOrderString    String
    aliasNamTC,
    bigOrderTC         TCursor
    db                  Database
    myAlias,
    aliasTableName     String
    sqlVar              SQL
endVar

; Initialize variables.
myAlias = "itchy"
aliasTableName = ":PRIV:aliasNam.db"
promptString = "Enter an SQL keyword (e.g. SELECT):"

enumAliasNames(aliasTableName)

; Prompt user to enter an SQL keyword.
sqlKeyword.view("SQL Keyword")
if sqlKeyword = promptString then
    return ; Exit method if user doesn't enter a keyword.
endif

; Use alias to open database.
aliasNamTC.open(aliasTableName)
if aliasNamTC.locate("DBName", myAlias) then
    db.open(myAlias) ; Use alias to get database handle to server
else
    msgStop("Stop", "The alias " + myAlias +
            " has not been defined.")
    return
endif

; Combine SQL statements and String variable sqlKeyword
; to create an SQL string.
bigOrderString = sqlKeyword +
    "CustName, Order_no, Sale_date, Qty
    FROM      Customer
    WHERE     Qty 1000 "

; Read and execute the query and process the results.
```

```

sqlVar.readFromString(bigOrderString)
if sqlVar.executeSQL(bigOrderTC) then
  doSomething(bigOrderTC) ; call custom proc to process data
else
  errorShow()
endIf

endMethod

```

wantInMemoryTCursor method

SQL

Specifies how to create a TCursor resulting from a SQL query.

Syntax

```
wantInMemoryTCursor ( [ const yesNo Logical ] )
```

Description

wantInMemoryTCursor specifies how to create a TCursor from a SQL query. When you execute a SQL query to a TCursor, that TCursor points to a live query view and changes made to the TCursor affect the underlying tables. When you call **wantInMemoryTCursor** with *yesNo* set to Yes or omitted, Paradox creates the TCursor in system memory, without a connection to underlying tables.

An in-memory TCursor is especially useful for performing quick what-if analyses. For example, to study the effect of giving each employee a 15 percent raise, you can query the employee data to increase all salaries by 15 percent. If you execute the query to an in-memory TCursor, you can manipulate the data there, without affecting the actual employee data.

Example

The following example uses an in-memory TCursor to study the effects of giving all employees a 15 percent raise. The code reads a predefined query from a file and uses the results in a calculation.

```

method pushButton(var eventInfo Event)
  var
    qVar          SQL
    tcRaise15     TCursor
    nuTotalPayroll Number
    MyDB          Database
  endVar

  MyDB.open("work")
  qVar.wantInMemoryTCursor(Yes)
  qVar.readFile("raise15.sql")
  qVar.executeSQL(MyDB, tcRaise15)

  nuTotalPayroll = tcRaise15.cSum("Salary")
  nuTotalPayroll.view("Payroll after 15% raise:")
endMethod

```

writeSQL method/procedure

SQL

Writes an SQL statement or an SQL string to a file.

Syntax

Method:

```
1. writeSQL ( const fileName String ) Logical
```

Procedure:

```
2. writeSQL ( const sqlString String, const fileName String ) Logical
```

Description

writeSQL writes a predefined SQL statement or SQL string to the file specified in *fileName*. If *fileName* already exists, Paradox overwrites it without asking for confirmation. **writeSQL** returns True if successful; otherwise, it returns False. This method does not evaluate the SQL commands.

Syntax 1 is a method—use dot notation to specify an SQL variable (e.g., `sqlVar.writeSQL("bigOrder.sql")`).

Syntax 2 is a procedure—use a String variable as the first argument (e.g., `writeSQL(sqlString, "bigOrder.sql")`).

Example

The following example prompts the user to type a table name and stores the name in a String variable. The code then it uses the String variable as a tilde variable in an SQL statement. The call to **writeSQL** writes the SQL statement (including the expanded tilde variable) to a file. If the user types ORDERS as the table name, the resulting SQL file would contain the following statement:

```
SELECT * FROM ORDERS
```

writeSQL does not determine whether the SQL statements are valid.

```
method pushButton(var eventInfo Event)
    var
        sqlString      SQL
        userTableName,
        sqlFileName,
        promptString    String
    endVar

    ; Initialize variables.
    sqlFileName = "user001.sql"
    promptString = "Enter table name here."
    userTableName = promptString

    ; Display a view() dialog box and prompt user for input.
    userTableName.view("Select * from table:")

    ; If user enters a string, use it in a tilde variable
    ; in the following SQL query.
    if userTableName promptString then
        sqlString =

            SQL

            SELECT * FROM ~userTableName
        endSQL
        writeSQL(sqlString, sqlFileName) ; Write user's query to a file.
    endIf
endMethod
```

StatusEvent type

StatusEvent type methods control messages that appear in the desktop Status Bar. Using StatusEvent type methods, you can attach code to built-in event methods to determine where and why messages are displayed. You can block messages or display them in a different status area, or in another object (e.g., a field object or text file). You can also use StatusEvent type methods to specify the text to be displayed in the message.

You can use the StatusReasons ModeWindow1, ModeWindow2, ModeWindow3, and StatusWindow to refer to the areas of the status bar shown below. Paradox and ObjectPAL place no restrictions (other than the size of the area) on the messages you display in these areas. How you use them is up to you, but consistency is recommended.

The StatusEvent type includes several derived methods from the Event type.

Methods for the StatusEvent type

Event	←	StatusEvent
errorCode		reason
getTarget		setReason
isFirstTime		setStatusValue
isPreFilter		statusValue
isTargetSelf		
reason		
setErrorCode		
setReason		

reason method

StatusEvent

Reports why a StatusEvent occurred.

Syntax

reason () SmallInt

Description

reason returns an integer value that reports why a StatusEvent occurred. StatusEvent reasons occur each time a built-in **status** method is called. ObjectPAL uses StatusReasons constants to test the value returned by **reason**.

Example

The following example copies all the messages that are sent to the Status Bar to a field. Assume that a form contains a field named *fldStatus*. The form's built-in **status** method examines the event packet to determine the reason. If the reason is StatusWindow, the form's built-in **status** method sends the status value to a field named *fldStatus*.

```
;frm1 :: status
method status(var eventInfo StatusEvent)
if eventInfo.isPreFilter()
then
; This code executes for each object on the form.
else
; This code executes only for the form.
if eventInfo.reason() = StatusWindow then
fldStatus.Value = eventInfo.statusValue()
endif
endif
endIf
endMethod
```

setReason method

StatusEvent

Specifies a reason for generating a StatusEvent.

setStatusValue method

Syntax

```
setReason ( const reasonId SmallInt )
```

Description

setReason specifies a reason for generating a StatusEvent. StatusEvent reasons indicate which Status Bar window received the message. ObjectPAL uses StatusReasons constants to set the reason for a StatusEvent.

Example

In the following 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 code changes the reason to ModeWindow1, and sets the message value to the name of the object that initiated the 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
```

setStatusValue method

StatusEvent

Specifies the text of a status message.

Syntax

```
setStatusValue ( const statusValue AnyType )
```

Description

setStatusValue specifies the text of a status message.

Example

See the **setReason** example.

statusValue meth

StatusEvent

Returns the text of a status message.

Syntax

```
statusValue ( ) AnyType
```

statusValue meth

Description

statusValue returns the text of a status message.

Example

The following example makes the default status messages more prominent to a user by copying each message to a field on the form. This feature is controlled by the *magnifyMessage* button 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
```

The following 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
```

Example 2

In this example, code is placed in the **status** method at the form's page level and traps for a change in the Persistent Field View setting. *modeWindow3* refers to the right field of the message line and displays the current view setting (e.g., Field View, Persistent Field View or Memo View). If the field is in Persistent Field View, a custom method named **persistFldVw** is called to perform predefined actions.

```
method status(var eventInfo StatusEvent)
if eventInfo.reason() = modeWindow3 then

    if eventInfo.statusValue()="Persist " then    ; note "Persist " is
                                                    ; followed by a space
        persistFldVw()
                                                    ; call custom method

    endIf
endIf
endMethod
```

String type

Strings store and manipulate alphanumeric data. A String variable's length is limited to the virtual memory on your computer. Strings occupy 1 byte of storage space per character. Empty stings are represented by double quotes ("")

String lengths may also be limited according to their use. For example, if you assign a String variable to an Alpha field in a Paradox table, the String variable cannot exceed the width of the Alpha field.

The String type includes several derived methods from the AnyType type.

Notes

- 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.
- Virtual memory is related to available disk space. For more information, see your Windows documentation.

Methods for the String type

AnyType	←	String	
blank		advMatch	oemCode
dataType		ansiCode	readFromClipboard
isAssigned		breakApart	rTrim
isBlank		chr	search
isFixedType		chrOEM	searchEx
view		chrToKeyName	size
		fill	sizeEx
		format	space
		ignoreCaseInStringCompares	string
		isIgnoreCaseInStringCompares	strVal
		isSpace	substr
		keyNameToChr	toANSI
		keyNameToVKCode	toOEM
		lower	upper
		lTrim	vkCodeToKeyName
		match	writeToClipboard

advMatch method

String

Searches text for a specified string.

Syntax

```
advMatch ( const pattern String [ , var matchVar String ] * ) Logical
```

Description

advMatch returns True if *pattern* is found within the string; otherwise, it returns False. To specify *pattern*, use a string and the optional symbols listed in the table. By default, this method is case sensitive by default. Use the String procedure **ignoreCaseInStringCompares** to change the case-sensitivity.

advMatch assigns matched patterns to *matchVar* variables as the patterns are found. The portions of the string that match wildcard elements are assigned to the variables from left to right. Because there multiple matches might be found, 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 must use two backslashes to instruct **advMatch** to treat a special character as a literal. For example, `\\(` tells **advMatch** to treat the parenthesis as a literal character.

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. In this case, one backslash and plus symbol (`\\+`) yields a literal character.

Symbol	Matches
\\	Include special characters (e.g., \t for Tab) as regular characters. Use two backslashes in quoted strings.
[]	Match the enclosed set. (e.g., [aeiou0-9] matches a, e, i, o, u, and 0 through 9)
[^]	Do not match the enclosed set. (e.g., [^ aeiou0-9] match anything except a, e, i, o, u, and 0 through 9)
()	Grouping
^	Beginning of string
\$	End of string

advMatch method

..	Match anything
*	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

The following example demonstrates **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-cs] 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

```

ansiCode procedure

String

Returns the ANSI code of a one-character string.

Syntax

```
ansiCode ( const char String ) SmallInt
```

Description

ansiCode returns the ANSI code of a one-character string. The returned value is an integer between 1 and 255.

Example

The following example assumes that a form contains four field objects: *showAllChars*, *ANSIField*, *OEMField*, and *KeyNameField*. The **keyPhysical** method for *showAllChars* translates each character in the string to its ANSI code, OEM code, and key-name equivalent. These character codes are then 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

```

breakApart method

```
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
```

breakApart method

String

Splits a string into an array of substrings.

Syntax

```
breakApart ( var tokenArray Array[ ] String [ , const separators String ] )
```

Description

breakApart splits a string into an array of substrings and each substring is written to an element of an array named *tokenArray*. You can specify one or more delimiting characters in *separators*. If you omit *separators*, substrings are delimited by a space. Delimiting characters are not included in *tokenArray*. **breakApart** is especially useful for importing data from a text file into a table.

Note

- Two empty delimiters 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 delimiters. By default, the method treats spaces as delimiters. The second call to **breakApart** specifies the asterisk as a delimiter. Empty array elements are created each time an asterisk immediately follows another asterisk. The third call specifies question mark, comma, and semicolon as delimiters.

```
; 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

```

chr procedure

String

Returns the one-character string represented by an ANSI code.

Syntax

```
chr ( const ansiCode SmallInt ) String
```

Description

chr returns a one-character string containing the ANSI character that corresponds to *ansiCode*. If *ansiCode* is not an integer between 1 and 255, **chr** fails.

You can **use** chr to generate characters that are not easily accessible with the keyboard.

Example

In the following example, the **pushButton** method for a button named *showChar* assigns the ANSI character 167 to the *sectionChar* variable. The code then converts character 167 to its key name, assigns it to *sectionKeyName*, and 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

```

chrOEM procedure

String

Returns the one-character string of an OEM code.

Syntax

```
chrOEM ( const oemCode SmallInt ) String
```

chrToKeyName procedure

Description

chrOEM returns a one-character string containing the OEM character that corresponds to *oemCode*. If *oemCode* is not an integer between 1 and 255, **chrOEM** fails.

You can use **chrOEM** to generate characters that are not easily accessible with the keyboard.

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
```

chrToKeyName procedure

String

Returns the virtual key code string of a one-character string.

Syntax

```
chrToKeyName ( const char String ) String
```

Description

chrToKeyName returns the virtual key code of *char* as a string. A key name is a virtual key code (e.g., VK_BACK for Backspace). This method returns the Keyboard constant name as a string (e.g., VK_BACK). Alphanumeric characters and symbols have one-character key names (e.g., J for the letter J).

Example

See the **chr** example.

fill procedure

String

Returns a string containing repeated instances of a character.

Syntax

```
fill ( const fillCharacter String, const fillNumber LongInt ) String
```

Description

fill returns a string containing repeated instances of 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.

In Paradox 8, the *fillNumber* parameter was changed to a LongInt.

Example

In the following example, the **pushButton** method for the *fillAndView* button creates two strings using the **fill** procedure. The first string is created by filling a variable with the same letter five times. The second string is created by repeating the string Shakespeare four times.

```
; fillAndView::pushButton
method pushButton(var eventInfo Event)
var
    str String
endVar
str = fill("X", 5)
str.view()                                ; displays the string XXXXX
```

```
str = fill("Shakespeare ", 4) ; add a space after
                                ; every occurrence
str.view()
; displays: Shakespeare Shakespeare Shakespeare Shakespeare
endMethod
```

format procedure

String

Controls the format of displayed or printed values.

Syntax

```
format ( const FormatSpec String, const value AnyType ) String
```

Description

format controls the format of displayed or printed values. *formatSpec* is a string expression containing one or more format specifications to be applied to String.

The following table lists the default format specifications and valid data types for each format category. You can also use AnyType values as data types, if the values can be interpreted consistently with the format category.

Format	Meaning	Valid data types	Default
Width	Set allowable field width and decimal precision	All	Entire data value
Alignment	Alignment within width	All	AR (right-aligned) for all numeric types, AL (left-aligned) for all others (including point)
Case	Uppercase or lowercase strings	All string types	No 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
	U.S. or Int'l separators	U.S.	

format procedure

Sign	Format of positive and negative numbers	All numeric	See following
	Positive		No leading positive sign 999
	Negative		Leading minus sign -999
Date	Specify date formats	Date & DateTime	mm/dd/yy(yy) for Date or hh:mm:ss am(pm), mm/dd/yy(yy) for DateTime
Time	Specify time formats	Time & 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.

Type	Spec	Meaning
Width	Wn	Specifies the total format width, including special characters, leading symbols or spaces, decimal point, and whole number separators
	W.n	Specifies the number of decimal places (W12.2 specifies a 12 character field, two of which are after the decimal point)
	W.W	Use decimal places from Windows numbers
	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 the symbol that precedes a number
	E\$W	Include currency symbol from Windows
	EDd	d specifies a decimal point character
	EDW	Use the Windows decimal point character
	ENc	c specifies whole-number separator
	ENW	Use the Windows whole number separator
	ELn	n specifies the number of leading zeros
	ELW	Use the Windows leading zero setting

	EPO	No symbol spacing
	EP-	Make symbol spacing for negatives
	EP+	Make symbol spacing for positives
	EPB	Make symbol spacing for all numbers
	EPW	Use the 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
	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\$)

format procedure

	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-l
	S+	The same as S-l + l
	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 l
	DM2	Month as 0l
	DM3	Month as Jan
	DM4	Month as January
	DML	Month from Windows Long Date
	DMS	Month from Windows Short Date
	DD1	Day as l
	DD2	Day as 0l
	DDL	Day from Windows Long Date
	DDS	Day from Windows Short Date
	DY1	Year as l
	DY2	Year as 0l
	DY3	Year as l90l
	DYL	Year from Windows Long Date
	DYS	Year from Windows Short Date

	DO(s)	specifies order and separators, use %W for weekday,%D for numeric day, %M for month, and %Y for year. Separators are literal (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
	D1	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)
	DEYEA(s)	s specifies A.D. dates
	DEYEB(s)	s specifies B.C. dates
Time	TH1	Hours as IT
	TH2	Hours as 01
	THW	Hours from Windows
	TM1	Minutes as I
	TM2	Minutes as 01
	TMW	Minutes from Windows
	TS1	Seconds as I
	TS2	Seconds as 01
	TSW	Seconds from Windows
	TNA(s)	s is a string that follows times before noon
	TNP(s)	s is a string that follows times after noon
	TNW	Noon settings from Windows

format procedure

	T0(s)	s specifies the order and separators, use %H for hours, %M for minutes, %S for seconds, %N for am/pm
	TOW	Order and separators from Windows
Logical	LT(s)	s specifies the representation of the logical True value
	LF(s)	s specifies the representation of the logical False value
	LY	Logical values as Yes and No
	LO	Logical values as On and Off

Example

In the following examples assume that a form contains a field named *formatField* and a button named *demoFormat*. The **pushButton** method for *demoFormat* demonstrates different format specifications. In each example, the method fills the *formatField* with the formatted string and displays a copy of the format specification in a dialog box (using *view*). The method does not move to the next example until the View dialog box is closed, allowing you to examine both the format specification and the formatted output before proceeding.

```
; demoFormat::pushButton
method pushButton(var eventInfo Event)
var
  x AnyType
  fs, formatField String
endVar

fs = "\"w6\", \"This is a test\""
formatField = format("w6", "This is a test")
; displays This i
formatfield.view("format: "+fs)

fs = "\"w7\", 1234567"
formatField = format("w7", 1234567)
; displays 1234567
formatfield.view("format: "+fs)

fs = "\"w9.2\", 1234.567"
formatField = format("w9.2", 1234.567)
; displays 1234.57
formatfield.view("format: "+fs)

; Here are some examples of alignment specifications:
fs = "\"w20,ac\", \"This is\""
formatField = format("w20,ac", "This is")
; displays This is
formatfield.view("format: "+fs)

fs = "\"w20,ac\", \"The Title\""
formatField = format("w20,ac", "The Title")
; displays The Title
formatfield.view("format: "+fs)

fs = "\"w20,ac\", \"Of the Book\""
formatField = format("w20,ac", "Of the Book")
; displays Of the Book
formatfield.view("format: "+fs)
```



```

fs = "\"w20,a1\",123456"
formatField = format("w20,a1",123456)
; displays 123456
formatfield.view("format: "+fs)

fs = "\"w20,ar\",123456"
formatField = format("w20,ar",123456)
; displays 123456
formatfield.view("format: "+fs)

; 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
formatfield.view("format: "+fs)

fs = "\"cl\", \"JUMPS OVER THE LAZY\""
formatField = format("cl", "JUMPS OVER THE LAZY")
; displays jumps over the lazy
formatfield.view("format: "+fs)

fs = "\"cc\", \"dOG.\""
formatField = format("cc", "dOG.")
; displays Dog.
formatfield.view("format: "+fs)

fs = "\"cc\", \"widgets'r us \" + \"too\""
formatField = format("cc", "widgets'r us " + "too")
; displays Widgets'R Us Too
formatfield.view("format: "+fs)

; 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
formatfield.view("format: "+fs)

fs = "\"w10.2, e$ci\", x"
formatField = format("w10.2, e$ci", x) ; displays $34.567,89
formatfield.view("format: "+fs)

fs = "\"w13.2, e$c\", x"
formatField = format("w13.2, e$c", x) ; displays $34,567.89
formatfield.view("format: "+fs)

fs = "\"w14.2, e$cb, a1\", x"
formatField = format("w14.2, e$cb, a1", x) ; displays $ 34,567.89
formatfield.view("format: "+fs)

fs = "\"w15.2, e$cz, a1\", x"
formatField = format("w15.2, e$cz, a1", x) ; displays $0000034,567.89
formatfield.view("format: "+fs)

fs = "\"w15.2, e$c*, a1\", x"
formatField = format("w15.2, e$c*, a1", x) ; displays $*****34,567.89
formatfield.view("format: "+fs)

; Here are some examples of sign specifications:
x = -3456.12
fs = "\"w8.2, s+\", x"

```

isEmpty

```
formatField = format("w8.2, s+", x)           ; displays -3456.12
formatfield.view("format: "+fs)

fs = "\"w11.2, e$c, sc\"", x"
formatField = format("w11.2, e$c, sc", x)      ; displays $3,456.12CR
formatfield.view("format: "+fs)

fs = "\"w14.2, e$c*, sp\"", x"
formatField = format("w14.2, e$c*, sp", x)    ; displays ($***3,456.12)
formatfield.view("format: "+fs)

fs = "\"w13.2, e$c*, s+\"", x"
formatField = format("w13.2, e$c*, s+", x)    ; displays -$***3,456.12
formatfield.view("format: "+fs)

fs = "\"w14.2, e$c*, sd\"", x"
formatField = format("w14.2, e$c*, sd", x)    ; displays $***3,456.12CR
formatfield.view("format: "+fs)

; Here are some miscellaneous examples:
fs = "\"D2\"", Date(\"3/7/1948\")
formatField = format("D2", Date("3/7/1948")) ; displays March 07,1948
formatfield.view("format: "+fs)

fs = "\"W9.2, AL\"", 1234.123"
formatField = format("W9.2, AL", 1234.123)
; displays 1234.12
formatfield.view("format: "+fs)

fs = "\"W9.2, AR\"", 1234.123"
formatField = format("W9.2, AR", 1234.123)
; displays 1234.12 right aligned in same field
formatfield.view("format: "+fs)

; 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
formatfield.view("format: "+fs)

; To display a date including the era (B.C. or A.D.):
fs = "\"DEYEA(A.D.)EB(B.C.)O(%M/%D/%Y %E)\",
    date(\"11/15/81\")\"
formatField = format("DEYEA(A.D.)EB(B.C.)O(%M/%D/%Y %E)",
    date("11/15/81"))
; displays 11/15/81 A.D.
formatfield.view("format: "+fs)

endMethod
```

isEmpty

String

Performs the same function as **isBlank**.

ignoreCaseInStringCompares procedure

String

Specifies whether to consider case when comparing strings.

Syntax

```
ignoreCaseInStringCompares ( const yesNo Logical )
```

Description

ignoreCaseInStringCompares specifies whether to consider case when comparing strings. By default, string comparisons are case sensitive (e.g., Q and q are not the same). If you use **ignoreCaseInStringCompares(Yes)**, string comparisons become case insensitive. Once you call **ignoreCaseInStringCompares(Yes)**, it stays in effect until you call **ignoreCaseInStringCompares(No)**.

To determine whether case is being considered, use **isIgnoreCaseInStringCompares**.

Example

In the following example, the **pushButton** method for the *tryCompare* button determines whether Paradox is set to ignore case in string comparisons. If **isIgnoreCaseInStringCompares** returns Yes, this code uses **ignoreCaseInStringCompares** to set it to No. The code then compares an uppercase and lowercase string. A message window informs the user that the strings are not equivalent. The code then sets **isIgnoreCaseInStringCompares** to Yes and compares the two strings again, 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
```

isIgnoreCaseInStringCompares procedure**String**

Reports whether case is considered when comparing strings.

Syntax

```
isIgnoreCaseInStringCompares ( ) Logical
```

Description

isIgnoreCaseInStringCompares returns True if case is considered when comparing strings; otherwise, it returns False.

To specify whether to consider case, use **ignoreCaseInStringCompares**.

Example

See the **ignoreCaseInStringCompares** example.

isSpace method**String**

Reports whether a string contains white space or is empty.

Syntax

```
isSpace ( const string String ) Logical
```

Description

isSpace returns True if string contains only white space or is empty (""); otherwise, it returns False. White space characters include spaces, tabs, carriage returns, linefeeds, and formfeeds.

Example

The following example creates several strings and determines whether they contain only white space or are empty. The following code is 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
```

keyNameToChr procedure**String**

Returns the one-character string represented by a virtual key-code string.

Syntax

```
keyNameToChr ( const keyName String ) String
```

Description

keyNameToChr returns the one-character string represented by the virtual key code *keyName*.

keyName must be a Keyboard constant (e.g., VK_BACK for Backspace) but must be supplied as a string (e.g., VK_BACK). Alphanumeric characters and symbols have one-character key names (e.g., J for the letter J).

Example

See the **keyNameToVKCode** example.

keyNameToVKCode procedure**String**

Returns the VK_Code of a virtual key-code string.

Syntax

```
keyNameToVKCode ( const keyName String ) SmallInt
```

Description

keyNameToVKCode returns the virtual key code (VK_Code) of the character represented by the virtual key code *keyName*, given as a string.

keyName must be a Keyboard constant (e.g., VK_BACK for Backspace) but must be supplied as a string (e.g., VK_BACK). Alphanumeric characters and symbols have one-character key names (e.g., J for the letter J).

Example

In the following example, the **pushButton** method for *showCode* sets a string variable named *keyStr* to an open bracket ([). The code 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
```

lower method

String

Converts a string to lowercase letters.

Syntax

```
lower ( ) String
```

Description

lower 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. The code 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
```

lTrim method

String

Removes leading blanks from a string.

Syntax

```
lTrim ( ) String
```

Description

lTrim removes spaces and Tab characters from the left end of a string.

Example

In the following 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 non-printing characters and then displays the trimmed version.

match method

```
; 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
```

match method

String

Compares a string with a pattern.

Syntax

```
match ( const pattern String [ , var matchVar String ] * ) Logical
```

Description

match compares a string with a pattern. If the string matches the pattern, **match** extracts the components that match the wildcard elements. The value of *pattern* consists of characters interlaced with the wildcard operators **..** and **@**. The **..** matches multiple characters (or no characters), and **@** matches any single character. **match** ignores or considers case depending on your system settings. Use **isIgnoreCaseInStringCompares** to determine the system setting and use **ignoreCaseInStringCompares** to turn case-sensitivity on or off.

matchVar is a variable to which the matching components are assigned. **match** assigns matched patterns to **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. 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.

Notes

- Quotes in *pattern* require special handling, periods do not. To embed a quote, precede it with a backslash (\). **match** treats periods as alphanumeric characters.
- Earlier versions of PAL required backslashes to delimit periods.

Example

The following example demonstrates **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 ignoreCaseInString() 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"
```

oemCode procedure

String

Returns the OEM code of a one-character string.

Syntax

```
oemCode ( const char String ) SmallInt
```

Description

oemCode returns the OEM code of *char*. *char* is a one-character string. The OEM code is an integer between 1 and 255.

Example

See the **ansiCode** example.

readFromClipboard method

String

Reads text from the Clipboard.

Syntax

```
readFromClipboard ( ) Logical
```

Description

readFromClipboard reads text from the Clipboard. This method reads text in CF_TEXT format.

readFromClipboard returns True if successful; otherwise it returns False.

Example

In the following example, a form has two buttons: *readFromClipboard* and *writeToClipboard*. The first button reads text from the Clipboard into a String variable that is stored in a table. The second button reads a String value from a table and writes it to the Clipboard.

The following code is attached to the **pushButton** method for *btnReadFromClipboard*:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
    vrString String
    tcString TCursor
endVar

; Open table to hold Strings
tcString.open("mystrings.db")
if vrString.readFromClipboard() then
    ; Add a record to the table and insert the value
    tcString.insertRecord()
    tcString.stringField = vrString
    tcString.unlockRecord()
endif
tcString.close()
endMethod
```

The following code is attached to the **pushButton** method for *btnWriteToClipboard*:

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

rTrim method

```
var
    vrString String
    tcString TCursor
endVar

; Open table to which contains strings
tcString.open("mystrings.db")
; Make sure there is data in the table
if tcString.nRecords() 0 then
    ; Copy a value to the String variable
    vrString = tcString.stringField
    ; Write it out to the Clipboard
    vrString.writeToClipboard()
endIf
tcString.close()
endMethod
```

rTrim method

String

Removes trailing blanks from a string.

Syntax

```
rTrim ( ) String
```

Description

rTrim removes spaces, tabs, carriage returns, and linefeed characters from the right end of a string.

Example

In the following example, the **pushButton** method for *trimRight* creates a string with trailing spaces. The code displays the original string, uses **rTrim** to remove the trailing non-printing characters and 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
```

search method

String

Returns the position of one string inside another string.

Syntax

```
search ( const str String ) SmallInt
```

Description

search searches for *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 case-sensitive. Use **ignoreCaseInStringCompares** to make the search case-insensitive.

Example

The following example searches for parts of the string *Goliath* and *Golgolithic*. This 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
```

searchEx method

String

Returns the position of one string inside another string.

Syntax

```
searchEx ( const str String ) LongInt
```

Description

searchEx searches for *str* within a target string. Use **searchEx** when working with very large string values. The **searchEx** returns a LongInt, while **search** returns a SmallInt value.

If *str* is found, **searchEx** 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, **searchEx** is case-sensitive. Use **ignoreCaseInStringCompares** to make it case-insensitive.

Example

See the **search** example.

size method

String

Returns the number of characters in a string.

Syntax

```
size ( ) SmallInt
```

Description

size returns the number of characters (including spaces) in a string as a SmallInt.

Note

- The maximum size of a string was increased in version 8, and is now limited by available virtual memory only. **size** has been retained for compatibility with existing applications;

sizeEx method

however, **sizeEx** (which returns a LongInt) is preferred because it returns the length of both small and large strings.

Example

In the following example, the **pushButton** method for *getSize* assigns a string to the variable *sourceText*. The code then displays the sentence and its size in a dialog box. The example then uses **size** to retrieve 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 sh
endMethod
```

sizeEx method

String

Returns the number of characters in a string.

Syntax

```
sizeEx ( ) LongInt
```

Description

sizeEx returns the number of characters (including spaces) in a string.

Use **sizeEx** when working with very large string values since the returned length is expressed as a LongInt (**size** returns a SmallInt).

Example

See the **size** example.

spacespace method

String

Creates a string containing a specified number of spaces.

Syntax

```
space ( const numberOfSpaces LongInt ) String
```

Description

space creates a string containing the number of spaces specified by *numberOfSpaces*.

The *numberOfSpaces* parameter was changed to LongInt in version 8.

Example

See the **isSpace** example.

string procedure**String**

Casts a value as a string.

Syntax

```
string ( const value AnyType [ , const value AnyType ] * ) String
```

Description

string casts a value as a string. If you specify multiple arguments, **string** will cast them all to strings and concatenate them to one string.

Example

In the following example, the **pushButton** method for *getNumToString* requests a number from the user. The code 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
```

strVal procedure**String**

Converts a value to a string.

Syntax

```
strVal ( const value AnyType ) String
```

Description

strVal converts *value* to a string. The data type specified in *value* can be an AnyType type.

Example

See the **size** example.

subStr method**String**

Returns a portion of a string.

Syntax

```
substr ( const startIndex LongInt [ , const numberOfChars LongInt ] ) String
```

Description

substr returns a portion of a string that starts at *startIndex* and continues for the number of characters specified by *numberOfChars*. The value of *startIndex* must be greater than 0 and less than or equal to

toANSI method

the size of the string. If *numberOfChars* is 0, **substr** returns a null string. If *numberOfChars* is omitted, **substr** returns the character that lies at the position specified by *startIndex*.

The *startIndex* and *numberOfChars* parameters were changed to `LongInt` in version 8.

Example

The following example assumes that a form contains a button named *getPhone* and four fields named *wholePhone*, *phAreaCode*, *phExchange*, and *phNumber*. This example uses **substr** to extract 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
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
```

toANSI method

String

Converts a string of OEM characters to ANSI characters.

Syntax

```
toANSI ( ) String
```

Description

toANSI converts a string of OEM characters to ANSI characters.

Example

In the following example, the **pushButton** method for a button named *showANSI* displays a string in two ways: as text, in the title of the dialog box and as ANSI code in the window of the dialog box. The last character in the string is the copyright symbol ((c)). This symbol appears in the title of the dialog box but is replaced by an underscore (_) in the window of the dialog box.

```
; 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
```

toOEM method

String

Converts a string of ANSI characters to OEM characters.

Syntax

```
toOEM ( ) String
```

Description

toOEM converts a string of ANSI characters to OEM characters.

Example

In the following example, the **pushButton** method for a button named *showOEM* displays a string in two ways: as text, in the title of the dialog box and as OEM code in the window of the dialog box. The last character in the string is the copyright symbol ((c)). This symbol appears in the title of the dialog box but is replaced by an underscore (_) in the window of the dialog box.

```
; 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
```

upper method**String**

Converts a string to uppercase letters.

Syntax

```
upper ( ) String
```

Description

upper converts a string to uppercase letters. Use **lower** to convert a string to lowercase letters.

Example

In the following example, the **pushButton** method for *makeUpper* retrieves a string from the user and converts it to uppercase letters. 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
```

vrCodeToKeyName method**String**

Converts a virtual key code constant to a virtual key code string.

Syntax

```
vrCodeToKeyName ( const vkCode SmallInt ) String
```

Description

vrCodeToKeyName returns the virtual key code name, as a string, of the character represented by the integer value *vkCode*.

This method returns the name of a Keyboard constant (e.g., VK_BACK for Backspace) as a string (e.g., VK_BACK). Alphanumeric characters and symbols have one-character key names (e.g., J for the letter J).

Example

See the **ansiCode** example.

writeToClipboard method**String**

Writes a string to the Clipboard.

Syntax

```
writeToClipboard ( ) Logical
```

Description

writeToClipboard writes a string to the Clipboard. This method copies strings in the CF_TEXT format. **writeToClipboard** returns True if successful and False if unsuccessful. The text copied to the Clipboard is ANSI.

Example

In the following example, a form has two buttons: *readFromClipboard* and *writeToClipboard*. The first button will read text from the Clipboard into a String variable which will then be stored in a table. The second button read a String value from a table and writes it out to the Clipboard.

The following code is attached to the **pushButton** method for *btnReadFromClipboard*:

```
; btnReadFromClipboard::pushButton
method pushButton(var eventInfo Event)
var
    vrString    String
    tcString    TCursor
endVar

; Open table to hold Strings
tcString.open("mystrings.db")
if vrString.readFromClipboard() then
    ; Add a record to the table and insert the value
    tcString.insertRecord()
    tcString.stringField = vrString
    tcString.unlockRecord()
endif
tcString.close()
endMethod
```

The following code is attached to the **pushButton** method for *btnWriteToClipboard*:

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

```

var
  vrString  String
  tcString  TCursor
endVar

; Open table to which contains strings
tcString.open("mystrings.db")
; Make sure there is data in the table
if tcString.nRecords() = 0 then
  ; Copy a value to the String variable
  vrString = tcString.stringField
  ; Write it out to the Clipboard
  vrString.writeToClipboard()
endif
tcString.close()
endMethod

```

System type

The System type contains methods and procedures for displaying messages, locating system information, setting printer options, manipulating the File Browser, working with the online Help system, and more.

Methods and procedures for the System type

beep	dlgNetUserName	enumRTLConstants	fileBrowser
close	dlgNetWho	enumRTLErrors	fileBrowserEx
compileInformation	dlgRename	enumRTLMethods	formatAdd
constantNameToValue	dlgRestructure	enumWindowHandles	formatDelete
constantValueToName	dlgSort	enumWindowNames	formatExist
cpuClockTime	dlgSubtract	errorClear	formatGetSpec
debug	dlgTableInfo	errorCode	formatSetCurrencyDefault
deleteRegistryKey	enableExtendedCharacters	errorHasErrorCode	formatSetDateDefault
desktopMenu	enumDesktopWindowHandles	errorHasNativeErrorCode	formatSetDateTimeDefault
dlgAdd	enumDesktopWindowNames	errorLog	formatSetLogicalDefault
dlgCopy	enumEnvironmentStrings	errorMessage	formatSetLongIntDefault
dlgCreate	enumExperts	errorNativeCode	formatSetNumberDefault
dlgDelete	enumFonts	errorPop	formatSetSmallIntDefault
dlgEmpty	enumFormats	errorShow	formatSetTimeDefault
dlgNetDrivers	enumFormNames	errorTrapOnWarnings	formatStringToDate
dlgNetLocks	enumPrinters	execute	formatStringToDateTime
dlgNetRefresh	enumRegistryKeys	executeString	formatStringToNumber
dlgNetRetry	enumRegistryValueNames	exit	formatStringToTime
dlgNetSetLocks	enumReportNames	expertsDir	getDefaultPrinterStyleSheet
dlgNetSystem	enumRTLClassNames	fail	getDefaultScreenStyleSheet
getDesktopPreference	msgInfo	searchRegistry	racerHide
getLanguageDriver	msgQuestion	sendKeys	tracerOff
getMouseScreenPosition	msgRetryCancel	sendKeysActionID	tracerOn
getRegistryValue	msgStop	setDefaultPrinterStyleSheet	ttracerSave

beep procedure

getUserLevel	msgYesNoCancel	setDefaultScreenStyleSheet	tracerToTop
helpOnHelp	pixelsToTwips	tracerShow	tracerWrite
helpQuit	play	setDesktopPreference	twipsToPixels
helpSetIndex	projectViewerClose	setMouseScreenPosition	version
helpShowContext	projectViewerIsOpen	setMouseShape	winGetMessageID
helpShowIndex	projectViewerOpen	setMouseShapeFromFile	winPostMessage
helpShowTopic	printerGetInfo	setRegistryValue	winSendMessage
helpShowTopicInKeywordTable	printerGetOptions	setUserLevel	writeEnvironmentString
isAppBarVisible	printerSetCurrent	showApplicationBar	writeProfileString
isErrorTrapOnWarnings	printerSetOptions	sleep	
isMousePersistent	readEnvironmentString	sound	
isTableCorrupt	readProfileString	startWebBrowser	
message	resourceInfo	sysInfo	
msgAbortRetryIgnore	runExpert	tracerClear	

beep procedure

System

Sounds the Windows default beep.

Syntax

beep ()

Description

beep sounds the Windows default beep. The beep is audible only if a sound device is installed and active.

Under certain circumstances, use **sound** to play a sound with a specific pitch and duration.

Example

The following example prompts you to enter a number and beeps if the number is out of range. This code is attached to a button's **pushButton** method:

```
; 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
```

close procedure

System

Closes the active form.

Syntax

```
close ( [ const returnValue AnyType ] )
```

Description

close returns a value to the calling form when *returnValue* (optional) is specified. This method does not generate an error if *returnValue* is specified and there is no calling form. Starts the process of closing the form, which includes removing the focus and departing.

Example

The following example closes the active form after asking 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
```

compileInformation procedure**System**

Lists information about the most recently compiled form.

Syntax

```
compileInformation ( var info DynArray[ ] AnyType )
```

Description

compileInformation lists information about the most recently compiled form. It writes the data to a dynamic array (DynArray) named *info* that you declare and pass as an argument. You can use **compileInformation** for analyzing large forms, libraries, scripts, and reports.

The following table displays the structure of the *info* DynArray:

Index	Definition
CodeSize	Compiled size of the code segment (in bytes).
CompileTime	Compile time (in milliseconds)
DataSize	Compiled size of the data segment (in bytes)
MethodCount	Number of methods that have code and/or comments
SourceSize	Size of the uncompiled source code (in bytes)
SymbolTableSize	Compiled size of the symbol table (in bytes)

Example

The following example writes compiler information to a dynamic array *dynCompileInfo*, and then displays it in a view dialog box:

constantNameToValue procedure

```
;analyzeObject::pushButton
method pushButton(var eventInfo Event)
var
    dynCompileInfo    Dynarray[]    AnyType
endVar

    compileInformation(dynCompileInfo)
    dynCompileInfo.view()
endmethod
```

constantNameToValue procedure

System

Returns the numeric value of a constant named *constantName*.

Syntax

```
constantNameToValue ( const constantName String ) AnyType
```

Description

constantNameToValue returns values for predefined ObjectPAL constants only. This method does not return values for user-defined constants.

Note

- For readability, ease of maintenance, and portability, use constants rather than numeric values.

Example

The following example returns the numeric value for an action constant named 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
```

constantValueToName procedure

System

Reports the name of a constant.

Syntax

```
constantValueToName ( const groupName String, const value AnyType, var constName
String ) Logical
```

Description

constantValueToName writes the name of a constant to *constName*. The constant's *value* equals value and that belongs to the group *groupName*, where *groupName* is one of the Types of Constants. This method returns True if successful; otherwise, it returns False.

Works for names of predefined ObjectPAL constants only; not for user-defined constants.

Example

See the **constantNameToValue** example.

cpuClockTime procedure

System

Returns the number of milliseconds that have passed since the computer was booted.

Syntax

```
cpuClockTime ( ) LongInt
```

Description

cpuClockTime returns the number of milliseconds that have passed since the computer was booted. The minimum clock increment is 55 milliseconds. This procedure is useful for measuring the interval between two events.

Example

The following example compares execution times for two for loops: one with an undeclared variable, the other with a declared variable. The code executes significantly faster when the variable is declared, although execution times vary by system.

```
; 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
```

debug procedure

System

Halts execution of a method and invokes the Debugger.

Syntax

```
debug ( )
```

deleteRegistryKey method

Description

debug halts execution of a method and invokes the debugger. **debug** statements have the same effect as setting a breakpoint, although unlike breakpoints, **debug** statements are saved with the method's source code. This procedure is useful for setting persistent breakpoints in methods while you are developing an application.

debug statements are only activated when you click Program, Compile With Debug; otherwise, they are ignored. This allows you to toggle **debug** statements without having to remove them from your code.

Turn Program, Compile With Debug *on* to test the application. Turn Program, Compile With Debug *off* to deliver the application.

Note

- **debug** works only in methods and procedures that you write, not for methods and procedures in the ObjectPAL run-time library.

Example

The following example executes a **for** loop. Halfway through the loop, a call to **debug** suspends execution and opens an Editor window containing the code. Click Program, Run to resume execution, or use the other Debugger features. Assume the command Program, Compile With Debug has been chosen from the ObjectPAL Editor menu.

```
; startDebugAt50::pushButton
method pushButton(var eventInfo Event)
var
  i SmallInt
endVar
for i from 1 to 100
  message(i)
  if i = 50 then
    debug()      ; will work only if Program, Compile With Debug
                  ; ObjectPAL Editor menu command is checked
  endIf
endFor
endMethod
```

deleteRegistryKey method

System

Deletes a registry key and/or value.

Syntax

```
deleteRegistryKey ( const key String, const value String, const rootKey LongInt )
Logical
```

Description

deleteRegistryKey deletes the registry key specified by *key*. **deleteRegistryKey** returns True if successful; otherwise, it returns False. If the parameter *value* is not empty, *key*'s value name is deleted, but not *key* itself. If value is empty, then only *key* is deleted. If *key* has subkeys a warning is generated, and *key* is not deleted.

You can set the rootKey with the predefined RegistryKeyType Constants.

Example

The following example adds and then deletes a registry key. If the value parameter is blank, the entire key is deleted; otherwise, the value and corresponding data are deleted.

```

var
  ar Array[] String
endvar

  setRegistryValue( "Software\\Core1\\Paradox\\8.0\\Pdoxwin\\Designer\\MyKey",
    "MyKeyValue", "MyKeyData", RegKeyCurrentUser )

  enumRegistryKeys( "Software\\Core1\\Paradox\\8.0\\Pdoxwin\\Designer",
    RegKeyCurrentUser, ar )
  ar.view()

  deleteRegistryKey( "Software\\Core1\\Paradox\\8.0\\Pdoxwin\\Designer\\MyKey",sa "",
    RegKeyCurrentUser )

  enumRegistryKeys( "Software\\Core1\\Paradox\\8.0\\Pdoxwin\\Designer",
    RegKeyCurrentUser, ar )
  ar.view()

```

desktopMenu procedure

System

Displays the Paradox desktop menu.

Syntax

```
desktopMenu ( )
```

Description

desktopMenu displays the Paradox desktop menu. This method is useful when you use a form as a dialog box that doesn't have an associated menu.

After you call **desktopMenu**, the Paradox desktop menu persists until:

- the current form or report loses focus
- a call to **removeMenu** restores the default menu for the form or report
- a call to **show** displays a custom menu

Example

The following example calls **desktopMenu** in the **setFocus** method on the page of a dialog box to display the Paradox default menu:

```

;pgel :: setFocus
method setFocus(var eventInfo Event)
  desktopMenu()
endMethod

```

dlgAdd procedure

System

Displays the Add Records In <table> To dialog box.

Syntax

```
dlgAdd ( const tableName String )
```

Description

dlgAdd displays the Add Records In <table> To dialog box.

tableName specifies the source table.

ObjectPAL code suspends execution until the user closes this dialog box.

dlgCopy procedure

Example

The following example displays the Add Records In <table> To dialog box and inserts the Customer table name as the source table. To complete the example, type the target table name and close the dialog box.

```
; showAddDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Add Records In <table> To dialog box with Customer as the source
dlgAdd("customer.db")
endMethod
```

dlgCopy procedure

System

Displays the Copy <table> To dialog box.

Syntax

```
dlgCopy ( const tableName String )
```

Description

dlgCopy displays the Copy <table> To dialog box. The argument *tableName* specifies the source table. ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Copy <table> To dialog box and specifies the Customer table name as the source table. To complete the example, type the target table name and close the dialog box.

```
; showCopyDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Copy <table> To dialog box with the Customer table as the source
dlgCopy("customer.db")
endMethod
```

dlgCreate procedure

System

Displays the Create Table dialog box.

Syntax

```
dlgCreate ( const tableName String )
```

Description

Displays the Create Table dialog box. The argument *tableName* specifies the name of the table to create. When you choose a table type and close the dialog box, this procedure opens a Table Type dialog box for the specified table type.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Table Type dialog box. To complete the example, choose the table type, fill out the field roster, and save the table.

```
; showCreateDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Type dialog box -- table name is not used
dlgCreate("sometbl.db")
endMethod
```

dlgDelete procedure**System**

Displays a warning dialog box prompting the user to confirm deletion of the table.

Syntax

```
dlgDelete ( const tableName String )
```

Description

dlgDelete displays a warning dialog box prompting the user to confirm deletion of the table. The argument *tableName* specifies the name of table to delete.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays a warning dialog box and inserts the Customer table name as the table to delete. To complete the example, close the dialog box and confirm the deletion.

```
; showDeleteDlg::pushButton
method pushButton(var eventInfo Event)
; invoke warning dialog box for the Customer table
dlgDelete("Customer.db") ; same as Tools, Utilities, Delete
endMethod
```

dlgEmpty procedure**System**

Displays a warning dialog box prompting the user to confirm the emptying of the table.

Syntax

```
dlgEmpty ( const tableName String )
```

Description

dlgEmpty displays a warning dialog box prompting the user to confirm the emptying of the table. The argument *tableName* specifies the name of table to empty.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the warning dialog box and inserts the Customer table name as the table to empty. To complete the example, close the dialog box and confirm the data deletion.

```
method pushButton(var eventInfo Event)
; Displays the warning dialog box for Customer table
dlgEmpty("Customer.db")
endMethod
```

dlgNetDrivers procedure**System**

Opens the Borland Database Engine (BDE) page of the Preferences dialog box.

Syntax

```
dlgNetDrivers ( )
```

Description

dlgNetDrivers opens the BDE page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes the dialog box.

Example

The following example opens the BDE page of the Preferences dialog box:

dlgNetLocks procedure

```
; showNetDrivers::pushButton
method pushButton(var eventInfo Event)
; invoke the BDE page of the Preferences dialog box
dlgNetDrivers()
endMethod
```

dlgNetLocks procedure

System

Creates and displays a table displaying lock information.

Syntax

```
dlgNetLocks ( )
```

Description

dlgNetLocks displays the Select File dialog box and prompts you to choose a table. Click Open to create a Paradox table named LOCKS.DB in your private directory. If the table already exists, Paradox overwrites it without asking for confirmation. If the table is already open, this procedure fails.

Here is the structure of LOCKS.DB:

Field name	Type & size	Description
Type	S 25	Lock type value
Username	A 14	User name of lock owner
Net Session	S	Net level session number
Our Session	S	BDE session number (if the lock is a BDE lock)
Record Number	A 33	Record number of locked record (if Type = Record Lock (Write))

Paradox creates the Locks table and displays it in a Table window.

Example

The following example opens the Select File dialog box. After you choose a file, **dlgNetLocks** creates and displays a Locks table.

```
; showNetLocks::pushButton
method pushButton(var eventInfo Event)
; creates a table of lock info :PRIV:LOCKS.DB, then displays it
dlgNetLocks()
endMethod
```

Lock type values for dlgNetLocks (System type)

- 0 = Record lock
- 1 = Special record lock
- 2 = Group lock
- 3 = Image lock
- 4 = Table open (no lock)
- 5 = Table read lock
- 6 = Table write lock
- 7 = Table exclusive lock
- 9 = Unknown lock

dlgNetRefresh procedure**System**

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetRefresh ( )
```

Description

dlgNetRefresh displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box.

For more information, see Database page (Preferences dialog box).

Example

The following example opens the Database page of the Preferences dialog box:

```
; showNetRefresh::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog
dlgNetRefresh()
endMethod
```

dlgNetRetry procedure**System**

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetRetry ( )
```

Description

dlgNetRetry displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box.

For more information, see Database page (Preferences dialog box).

Example

The following example opens the Database page of the Preferences dialog box:

```
; showNetRetryDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetRetry()
endMethod
```

dlgNetSetLocks procedure**System**

Displays the Table Locks dialog box, allowing you to place a lock on a table.

Syntax

```
dlgNetSetLocks ( )
```

Description

dlgNetSetLocks displays the Table Locks dialog box, allowing you to place a lock on a table. ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example opens the Table Locks dialog box:

```

; showSetLocks::pushButton
method pushButton(var eventInfo Event)
dlgNetSetLocks() ; invoke the Table Locks dialog box
endMethod

```

dlgNetSystem procedure

System

Displays the Borland Database Engine (BDE) page of the Preferences dialog box:

Syntax

```
dlgNetSystem ( )
```

Description

dlgNetSystem displays the BDE page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example opens the Borland Database Engine (BDE) page of the Preferences dialog box:

```

; showNetSystem::pushButton
method pushButton(var eventInfo Event)
; invoke the BDE page of the Preferences dialog box
dlgNetSystem()
endMethod

```

dlgNetUserName procedure

System

Displays the Database page of the Preferences dialog box. The Database page shows the current user's network name.

Syntax

```
dlgNetUserName ( )
```

Description

dlgNetUserName displays the Database page of the Preferences dialog box. The Database page displays the current user's network name. ObjectPAL code suspends execution until the user closes this dialog box.

For more information, see Database page (Preferences dialog box).

Example

The following example opens the Database page of the Preferences dialog box, which shows the current network user's name:

```

; showUserName::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetUserName()
endMethod

```

dlgNetWho procedure

System

Displays the Database page of the Preferences dialog box.

Syntax

```
dlgNetWho ( )
```

Description

dlgNetWho displays the Database page of the Preferences dialog box. ObjectPAL code suspends execution until the user closes this dialog box.

For more information, see Database page (Preferences dialog box).

Example

The following example opens the Database page of the Preferences dialog box:

```
; showUserList::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetWho()
endMethod
```

The following example opens the Database page of the Preferences dialog box:

```
; showUserList::pushButton
method pushButton(var eventInfo Event)
; invoke the Database page of the Preferences dialog box
dlgNetWho()
endMethod
```

dlgRename procedure**System**

Displays the Rename <table> To dialog box.

Syntax

```
dlgRename ( const tableName String )
```

Description

dlgRename displays the Rename <table> To dialog box. The argument *tableName* specifies the table to rename.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Rename <table> To dialog box and specifies Customer as the table to rename. To complete the example, type a new name and close the dialog box.

```
; showRenameDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Table Rename <table> To dialog box
dlgRename("customer.db")
endMethod
```

dlgRestructure procedure**System**

Displays the Restructure Table dialog box.

Syntax

```
dlgRestructure ( const tableName String )
```

Description

dlgRestructure displays the Restructure Table dialog box. The argument *tableName* specifies the table to restructure, including the filename's extension. If *tableName* does not specify a path,

dlgRestructure searches for the table in the working directory.

dlgSort procedure

If *tableName* does not specify an extension, or specifies an extension of .DB, **dlgRestructure** displays the Restructure Paradox Table dialog box.

If *tableName* specifies an extension of .DBF, **dlgRestructure** displays the Restructure dBASE Table dialog box.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Restructure Table dialog box and specifies Customer as the table to restructure. To complete the example, modify the structure and close the dialog box.

```
; showRestructureDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Restructure Table dialog box for Customer table
dlgRestructure("customer.db")
endMethod
```

dlgSort procedure

System

Displays the Sort Table dialog box.

Syntax

```
dlgSort ( const tableName String )
```

Description

tableName specifies the name of table to sort.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Sort Table dialog box and chooses Customer as the table to sort. To complete the example, create a sort specification and close the dialog box.

```
; showSortDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Sort Table dialog box
dlgSort("customer.db")
endMethod
```

dlgSubtract procedure

System

Displays the Subtract Records In <table> From dialog box.

Syntax

```
dlgSubtract ( const tableName String )
```

Description

dlgSubtract displays the Subtract Records In <table> From dialog box. The argument *tableName* specifies the table from which to subtract records.

The dialog box opens with the argument *tableName* already specified, prompting the user to choose what to subtract from *tableName*. ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays the Subtract Records In <table> From dialog box and specifies Customer as the source table from which to subtract records. To complete the example, close the dialog box.

```
; showSubtractDlg::pushButton
method pushButton(var eventInfo Event)
; invoke the Subtract Records In <table> From dialog box
dlgSubtract("customer.db") ;
endMethod
```

dlgTableInfo procedure**System**

Displays the Structure Information dialog box.

Syntax

```
dlgTableInfo ( const tableName String )
```

Description

dlgTableInfo displays the Structure Information dialog box. The argument *tableName* specifies the table from which to obtain the structure information.

ObjectPAL code suspends execution until the user closes this dialog box.

Example

The following example displays 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")
endMethod
```

enableExtendedCharacters procedure**System**

Determines whether you can type extended character codes from the numeric keypad without enabling the NumLock key.

Syntax

```
enableExtendedCharacters ( const yesNo Logical ) Logical
```

Description

enableExtendedCharacters determines whether you can type extended character codes from the numeric keypad without enabling the *NumLock* key. If *yesNo* is set to True, you can type extended characters without *NumLock*. If *yesNo* is set to False, *NumLock* must be on to enter extended character codes; otherwise, keypad keys function as navigation keys. This setting affects all forms, and remains active while Paradox is running. This setting is not saved when you exit.

enableExtendedCharacters is used in international applications or other environments where keyboards do not have NumLock keys. This method returns True if successful; otherwise, it returns False.

Example

The following example enables extended characters when the form opens:

```
method open(var eventInfo Event)
```

enumDesktopWindowHandles procedure

```
    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:
        doDefault
        enableExtendedCharacters(Yes)
    endIf

endMethod
```

enumDesktopWindowHandles procedure

System

Lists the window handles of open windows on the Paradox desktop.

Syntax

```
enumDesktopWindowHandles ( var windowHandles DynArray [ ] AnyType [, const className String ] )
```

Description

enumDesktopWindowHandles lists the handles of open windows on the Paradox desktop. This procedure writes the list to a dynamic array (DynArray) named *windowHandles*. The *windowHandles* index contains the handle and specifies the name of the window. The optional *className* argument specifies that the DynArray contains only windows whose *className* equals the name of the window class.

Example

The following example builds and displays a dynamic array (DynArray) of all the window titles open on the Paradox desktop:

```
method pushButton(var eventInfo Event)
var
    winHandles DynArray[] String
endvar

enumDesktopWindowHandles(winHandles) ; enumerate desktop window
                                     ; handles to a DynArray
winHandles.view()                   ; lists all windows open
                                     ; in the Paradox desktop

endMethod
```

enumDesktopWindowNames procedure

System

Lists the names of open windows on the Paradox desktop.

Syntax

```
1. enumDesktopWindowNames ( const tableName String ) Logical
2. enumDesktopWindowNames ( const windowNames Array [ ] String [, const className String] )
```

Description

enumDesktopWindowNames lists the names of open windows owned by the Paradox desktop. Syntax 1 creates a Paradox table named *tableName* that lists the name, class, position, and size of each window. If *tableName* does not specify a path, **enumDesktopWindowNames** creates the table in the working directory. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails.

The following table displays the structure of tableName:

Field name	Type & size	Description
WindowName	A 64	Window name (if the window has no name, this field is empty)
ClassName	A 63	Window type
Position	A 12	Coordinates of upper-left corner (e.g., 456, 553)
Size	A 12	Coordinates of lower-right corner (e.g., 889, 221)
Handle	I	Window handle
ChildId	I	ID number of child window (0 = no child window)
ParentHandle	I	Handle of parent window
InstanceHandle	I	Handle of window instance

Syntax 2 fills the array specified by *winArray* with the names of the windows. You must declare *winArray* before calling this method. Applications are listed in Windows z-order—the top window is listed first in the array, the window in the second layer is listed second, and so on. The optional argument *className* specifies that *winArray* displays only the names of windows whose class is equal to *className*.

Compare this method to *enumWindowNames*, which lists all of the Windows applications running on your system.

Example

The following example writes the open desktop window titles to an array. The code then 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 view
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
tempTV.open("wNameTbl")           ; show the table
endMethod
```

enumEnvironmentStrings procedure

System

Lists all of the items from the DOS environment.

Syntax

```
enumEnvironmentStrings ( var values DynArray[ ] String ) Logical
```

Description

enumEnvironmentStrings lists all of the items from the DOS environment. This method writes the items to a dynamic array named *values*, which you declare and pass as an argument.

enumExperts procedure

Example

The following example creates and displays a dynamic array named *dyn* that lists items from the DOS environment:

```
;thisButton::pushButton
method pushButton(var eventInfo Event)
  var
    dyn  DynArray[] String
  endVar

  enumEnvironmentStrings(dyn)
  dyn.view()
endmethod
```

enumExperts procedure

System

Lists all of the experts available to Paradox.

Syntax

- 1. enumExperts (const *expertType* String, var *expertNames* DynArray [] AnyType)
- 2. enumExperts (const *expertType* String, const *expertName* String)

Description

enumExperts lists the experts available to Paradox. The *expertType* parameter specifies the type of experts that are included in the list. Syntax 1 fills a dynamic array (DynArray) named *expertNames* with the names of the experts. Syntax 2 lists the experts in a table. The following table displays the format of the table created in Syntax 2:

Field	Type & size	Description
Expert	Alpha 25	Registered expert name
Name	Alpha 25	Visible expert name
Description	Alpha 255	Help description text
File Name	Alpha 255	Expert filename (including the path)
Icon	Graphic	Experts icon graphic

The valid values for *expertType* are:

Document	Identifies document experts (e.g., Table or Form experts)
Object	Identifies experts that are activated when placing an object on a form or report (e.g., the Button Expert)
CoreUI	Identifies experts selected from a menu (e.g., Text Import Expert)

Example

The following example enumerates all Paradox experts.

```
method pushButton(var eventInfo Event)
  Var
    da      DynArray[] AnyType
```



```

endVar

enumExperts( "Document", da )

da.view()
endMethod

```

enumFonts procedure

System

Creates a table listing the fonts installed on your system.

Syntax

```

1. enumFonts ( const tableName String )Logical
2. enumFonts ( const deviceType SmallInt, var fontList Array[] String )Logical

```

Description

enumFonts creates a table listing the fonts on your system. The argument *tableName* specifies the table. By default, **enumFonts** creates *tableName* in your working directory. If *tableName* already exists, this procedure overwrites it without asking for confirmation. If *tableName* is open, enumFonts fails.

The following table displays the structure of *tableName*:

Field name	Type & size	Description
FaceName	A 64	Font name. (e.g., Arial)
FontSize	A 8	Font size in printer's points. (e.g., 12)
Attribute	A 64	Display/print attribute.(e.g., Normal)

Syntax 2 builds an array of fonts in *fontList*. The argument *deviceType* has two possible values: 1 (indicating screen display fonts), and 2 (indicating printer fonts).

Example

The following example creates and lists system fonts in a table named FONTS.DB. The code then searches a TCursor for a font named Modern. If Modern is in the table, the code sets the Font.TypeFace property of an unlabeled field object named balanceField to Modern.

```

; getFonts::pushButton
method pushButton(var eventInfo Event)
var
    tempTV TableView
endVar
enumFonts("fonts.db")      ; write font names to a table
tempTV.open("fonts.db")    ; show the table
endMethod

```

enumFormats procedure

System

Lists the current formats.

Syntax

```

enumFormats ( const formatType String, var formats DynArray[ ] String ) Logical

```

enumFormNames procedure

Description

enumFormNames lists the current formats. The data type of the argument *formatType* is Date, Number, Time, DateTime or Logical. This method writes the list to formats, a dynamic array that you declare and pass as an argument.

This method returns True if successful; otherwise, it returns False.

Example

The following example creates and displays a dynamic array named *dyn* that lists the formats for Date:

```
; btnInspectFormat :: pushButton
method pushButton(var eventInfo Event)
var
    s    String
    dyn  DynArray[] String
endVar

s = "Date"
s.view("Enter format to inspect")
enumFormNames(s, dyn)
dyn.view()
endmethod
```

enumFormNames procedure

System

Creates an array listing open forms.

Syntax

```
enumFormNames ( var formNames Array[ ] String )
```

Description

enumFormNames creates an array named *formNames* that lists the open forms. You must declare *formNames* as a resizable array before calling enumFormNames. Forms are listed in Windows z-order—the top form is listed first in the array, the form in the second layer is listed second, and so on.

Example

The following example writes the filenames of open forms to an array named *openForms*. The code then displays openForms.

```
; getFormNames::pushButton
method pushButton(var eventInfo Event)
var
    openForms Array[] String
endVar
enumFormNames(openForms)
openForms.view() ; Lists filenames of open forms.
endMethod
```

enumPrinters procedure

System

Lists the printers installed on your system.

Syntax

```
enumPrinters ( var printers Array[ ] String ) Logical
```

Description

enumPrinters lists the printers installed on your system. **enumPrinters** fills an array named printers with elements that each contain the name, driver name, and port (separated by commas) of every printer installed on your system. You must declare printers as a resizable array before calling this method.

If the printer name is Postscript Printer, the driver is PSCRIPT.DRV, and the port is LPT1:

```
PostScript Printer,pscript,LPT1:
```

You pass an array item to **printerSetCurrent** to specify the active printer. Use the String method **breakApart** to separate the components (e.g., to display a list of printer names).

Example

The following example retrieves a list of printers installed on your system.

```
method run(var eventInfo Event)
var
    arPrinters    array[]anytype
endVar

enumPrinters(arPrinters) ; Get a list of installed printers.
arprinters.view()       ; View the above List

endMethod
```

enumRegistryKeys method**System**

Fills an array with keys from the registry.

Syntax

```
enumRegistryKeys ( const key String, const rootKey LongInt , var keyinfo Array[]
String, ) Logical
```

Description

enumRegistryKeys fills an array with keys from the registry. **enumRegistryKeys** returns True if successful; otherwise, it returns False. An array named keyinfo is populated with the full key path from the specified **key** and **rootKey**. The subkeys of **key** are also placed in the array. If **key** is blank, the subkeys of **rootKey** are enumerated.

Set **rootKey** with the predefined RegistryKeyType Constants.

Example

The following example builds an array of the registry keys that contain the string "software\Corel":

```
enumRegistryKeys( "software\\Corel", regKeyLocalMachine, ar )
;\\ values in array
ar[1] = "software\\Corel"
```

The following example displays all registry keys residing under the "Software\Corel\Paradox\8.0\Pdowin" key:

```
var
    ar Array[] String
endvar

enumRegistryKeys( "Software\\Corel\\Paradox\\8.0\\Pdowin", RegKeyCurrentUser, ar )
ar.view()
```

enumRegistryValueNames procedure**System**

Fills a dynamic array with values and data from the registry.

Syntax

```
enumRegistryValueNames ( const key String, const rootKey LongInt, var keyInfo Array[]
String ) Logical
```

Description

enumRegistryValueNames fills an array named *keyInfo* with the value names of the registry specified in *key*. **enumRegistryValueNames** returns True if successful; otherwise, it returns False.

key is entered as a path similar to a file path; however, unlike a file path, wildcards are not expanded.

key cannot contain a single backslash and cannot be empty. Its maximum size is 65,534 bytes. **keyInfo** contains the value names for the specified *key*. **rootKey** is analogous to a directory drive. Set **rootKey** with the predefined RegistryKeyType Constants.

Example

The following example lists all the value names under the Software\Corel\Paradox\8.0\Pdowin\Designer key. The code assigns the value names and their corresponding values to a DynArray, and displays it:

```
var
    ar      Array[]      String
    dyn     DynArray[]   AnyType
    i       SmallInt
endvar

enumRegistryValueNames( "Software\\Corel\\Paradox\\8.0\\Pdowin\\Designer",
RegKeyCurrentUser, ar )

if ar.size() > 0 then
    for i from 1 to ar.size()
        dyn[ ar[ i ] ] = getRegistryValue(
"Software\\Corel\\Paradox\\8.0\\Pdowin\\Designer", ar[ i ], RegKeyCurrentUser )
    endfor
endif

dyn.view()
```

enumReportNames procedure**System**

Creates an array listing open reports.

Syntax

```
enumReportNames ( var reportNames Array[ ] String )
```

Description

enumReportNames fills an array named *reportNames* with the names of open reports in your desktop. You must declare *reportNames* as a resizable array before calling this method. Reports are listed in Windows z-order—the top report is listed first in the array, the report in the second layer is listed second, and so on.

Example

The following example writes the open report names to an array named *openReports* and then displays the array:

```

; getReportNames::pushButton
method pushButton(var eventInfo Event)
var
    openReports Array[] String
endVar
enumReportNames(openReports)
openReports.view()          ; lists open reports
endMethod

```

enumRTLClassNames procedure

System

Creates a table listing the object types or classes known to ObjectPAL.

Syntax

```
enumRTLClassNames ( const tableName String ) Logical
```

Description

enumRTLClassNames creates a table named *tableName* listing the object types (classes) in the ObjectPAL run-time library. By default, **enumRTLClassNames** saves *tableName* in the working directory. If *tableName* already exists, **enumRTLClassNames** overwrites it without asking for confirmation. If *tableName* is open, **enumRTLClassNames** fails. This method returns True if successful; otherwise, it returns False.

The following table displays the structure of *tableName*:

Field name	Type & size	Description
ClassName	A 32	ObjectPAL type name. (e.g., UIObject)

Example

The following example writes the run-time library class names to a table named *Rtlclass*. The code then displays the table.

```

; getRTLClasses::pushButton
method pushButton(var eventInfo Event)
var
    tempTV TableView
endVar
enumRTLClassNames("rtlclass.db")      ; write class names to table
tempTV.open("rtlclass")                ; show the table
endMethod

```

enumRTLConstants procedure

System

Creates a table listing the constants defined by ObjectPAL.

Syntax

```
enumRTLConstants ( const tableName String ) Logical
```

Description

enumRTLConstants creates a table named *tableName* listing all the constants defined in the ObjectPAL run-time library. By default, **enumRTLConstants** creates *tableName* in the working directory. If *tableName* already exists, **enumRTLConstants** overwrites it without asking for confirmation. If *tableName* is open, **enumRTLConstants** fails.

The following table displays the structure of *tableName*:

enumRTLErrors method

Field name	Type & size	Description
GroupName*	A 32	One of the types of constants (e.g., ActionDataCommands)
ConstantName*	A 48	Symbolic name of the constant (e.g., DataArriveRecord)
Type	A 48	Data type of the constant (e.g., SmallInt)
Value	A 64	Value of the constant (e.g., 3111)

(* = key field)

Note

- Although Paradox provides the constant's values, refer to constants by name in your code. Use the **constantValueToName** and **constantNameToValue** methods to convert values and constants.

Example

The following example writes the run-time library constant descriptions to a table named *Rtlconst*. The code then displays the table.

```
; getRTLConsts::pushButton
method pushButton(var eventInfo Event)
var
    tempTV TableView
endVar
enumRTLConstants("rtlconst.db") ; write constants names to table
tempTV.open("rtlconst")         ; show the table
endMethod
```

enumRTLErrors method

System

Lists the error codes and messages used by ObjectPAL.

Syntax

```
enumRTLErrors ( const tableName String ) Logical
```

Description

enumRTLErrors creates a table named *tableName* listing the error codes and messages used by ObjectPAL. By default, **enumRTLErrors** creates *tableName* in the working directory. If *tableName* already exists, **enumRTLErrors** overwrites it without asking for confirmation.

The following table displays the structure of *tableName*:

Field	Type & size	Description
ErrorNo*	N	Error number (decimal)
ErrorNoX	A 8	Error number (hex)
Name	A 48	Error constant name, if it exists (e.g., peNoMemory). If an error constant name does not exist, the Name field displays the following string<Unmapped Error>
Value	M 230	Error message (e.g., Insufficient memory for this operation)

(* = key field)

This method returns True if successful; otherwise, it returns False. If you pass **enumRTLErrors** an invalid table name, this procedure fails and returns False.

Example

The following example writes the run-time library error codes and descriptions to a table named *Rtlerror*. The code then displays the table.

```
;getRTLErrors::pushButton
method pushButton(var eventInfo Event)
var
    tv TableView
endVar

enumRTLErrors("Rtlerror.db")
tv.open("Rtlerror.db")
endMethod
```

enumRTLMethods procedure

System

Creates a table listing the RTL methods and RTL procedures in ObjectPAL.

Syntax

```
enumRTLMethods ( const tableName String ) Logical
```

Description

enumRTLMethods creates a table named *tableName* listing the RTL methods and procedures used by ObjectPAL. By default, **enumRTLMethods** creates *tableName* in the working directory. If *tableName* already exists, **enumRTLMethods** overwrites it without asking for confirmation. If *tableName* is open, **enumRTLMethods** fails.

The following table displays the structure of *tableName*:

Field name	Type & size		Description
ClassName*	A	32	ObjectPAL type name (e.g., FileSystem)
MethodType*	A	8	Method (for methods) or Proc (for procedures)
MethodName*	A	64	Name of method or procedure (e.g., isDir)
MethodArgs*	A	255	Arguments to the method or procedure (e.g., const dirName String)
ReturnType*	A	32	Data type of returned value or blank if no return value (e.g., Logical)

(* = key field)

Example

The following example writes the run-time library method descriptions to a table named *Rtlmeth*. The code then displays the table.

```
; getRTLMethods::pushButton
method pushButton(var eventInfo Event)
var
    tempTV TableView
endVar
```

enumWindowHandles procedure

```
enumRTLMethods("rtlmeth.db")    ; write method names to table
tempTV.open("rtlmeth")          ; show the table
endMethod
```

enumWindowHandles procedure

System

Lists the open window handles.

Syntax

```
enumWindowHandles ( var windowHandles DynArray [ ] AnyType [, const className String ]
)
```

Description

enumWindowHandles lists the handles of the open windows running under Windows. This procedure writes the list to a dynamic array (DynArray) named *windowHandles*. The *windowHandles* index contains the handle and the value is the name of the window. The optional *className* argument specifies that the generated list contains only those windows whose *className* equals the name of the window class.

Example

The following example builds and displays a dynamic array (DynArray) of all the window handles:

```
method pushButton(var eventInfo Event)
var
    winHandles DynArray[] String
endvar

enumWindowHandles(winHandles)    ; enumerate desktop window
                                ; handles to a DynArray
winHandles.view()                ; lists all open windows

endMethod
```

enumWindowNames procedure

System

The following example creates a list of the applications currently running under Windows.

Syntax

```
1. enumWindowNames ( const tableName String ) Logical
2. enumWindowNames ( var windowNames Array [ ] String [, const className String] )
```

Description

enumWindowNames creates a list of applications currently running under Windows. Syntax 1 creates a table named *tableName* listing the name, class, position, size, and handles to each open application on your system. By default, **enumWindowNames** creates *tableName* in the working directory. If *tableName* already exists, **enumWindowNames** overwrites it without asking for confirmation. If *tableName* is open, **enumWindowNames** fails.

The following table displays the structure of *tableName*:

Field name	Type & size	Description
WindowName	A 64	Name of window, or blank if no name
ClassName	A 64	Window type
Position	A 12	Coordinates of upper-left corner (e.g., 456, 553).

Size	A 12	Coordinates of lower-right corner (e.g., 889, 221).
Handle	I	Window handle
ChildId	I	ID number of child window (0 = no child window)
ParentHandle	I	Handle of parent window
InstanceHandle	I	Handle of window instance

Syntax 2 fills an array named *winArray* with the names of all current applications, in Windows z-order—the top application is listed first in the array, the application in the second layer is listed second, and so on. You must declare *winArray* before calling this procedure. An optional argument named *className* specifies that only those windows whose class is equal to *className* appear in *winArray*.

Compare this method to **enumDesktopWindowNames**, which lists only the open windows owned by Paradox.

errorClear procedure

System

Clears the error stack.

Syntax

```
errorClear ( )
```

Description

errorClear clears the error stack of all error codes and error messages.

Example

The following example clears the error stack:

```
; clearError::pushButton
method pushButton(var eventInfo Event)
errorClear()           ; clear the error stack
endMethod
```

errorCode procedure

System

Returns a number representing the most recent run-time error or error condition.

Syntax

```
errorCode ( ) SmallInt
```

Description

errorCode returns a number representing the most recent run-time error or error condition.

ObjectPAL provides error constants for these integers (e.g., *peObjectNotFound*). Use **enumRTLErrors** to create a list of error codes and error messages.

Calling **errorCode** is not the same as calling **eventInfo**. **setErrorCode**, which adds error information to the event packet, but not to the error stack.

Example

The following example uses a try clause to attempt to attach to an object *boxOne* to 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 identify the error and then takes appropriate action.

errorHasErrorCode method

```
; handleErrorcode::pushButton
method pushButton(var eventInfo Event)
var
  obj UIObject
endVar
try
  obj.attach("boxOne")
  obj.color = Red
onFail
  if errorCode() = peObjectNotFound then
    obj.create(BoxTool, 180, 180, 360, 360)
    obj.name = "boxOne"
    obj.visible = Yes
    retry
  else
    fail()
  endIf
endTry
endMethod
```

errorHasErrorCode method

System

Searches for a specific error code in the error stack.

Syntax

```
errorHasErrorCode ( const errCode SmallInt ) Logical
```

Description

errorHasErrorCode searches the error stack for the error specified by *errCode*. *errCode* is an Errors constant or a user-defined error constant. **errorHasErrorCode** returns True if the error is found; otherwise, it returns False.

Use **enumRTLErrors** to create a list of error codes and error messages.

Example

The following example searches the error stack for a key violation:

```
if errorHasErrorCode(peKeyViol) then

    ; error handling code goes here

endif
```

errorHasNativeErrorCode method

System

Searches for an SQL error code in the error stack.

Syntax

```
errorHasNativeErrorCode ( const errCode LongInt ) Logical
```

Description

errorHasNativeErrorCode searches the error stack for an SQL error code. The SQL error is specified by the argument *errCode*. Error codes vary depending on the server and may overlap with some Paradox error codes. **errorHasNativeErrorCode** returns True if the error is found; otherwise, it returns False.

Example

The following example searches the error stack for the server error associated with the *peServerPathIllegal* constant. The constant is set to an error code listed in the server's documentation:

```
if errorHasNativeErrorCode(peServerPathIllegal) then

    ; error handling code goes here

endIf
```

errorLog procedure**System**

The following example adds error information to the error stack.

Syntax

```
errorLog ( const errorCode SmallInt, const errorMessage String )
```

Description

errorLog adds error information to the error stack. Use Errors constants or user-defined error constants to specify the value of *errorCode*. Use **enumRTLErrors** to create a list of error codes and error messages.

Calling **errorLog** is not the same as calling eventInfo. **setErrorCode**, which adds error information to the event packet, but not to the error stack.

Example

The following example uses a try clause to attempt to attach to an object *boxOne* to 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()
```

`errorMessage procedure`

```
    endIf  
  endTry  
endMethod
```

errorMessage procedure

System

Returns a string containing the most recent run-time error message or error condition from the error stack.

Syntax

```
errorMessage ( ) String
```

Description

errorMessage returns a string containing the most recent run-time error message or error condition from the error stack. This method returns the empty string ("") if no error has occurred.

errorMessage is especially useful for logging error messages during a session.

Example

See the **errorLog** example.

errorNativeCode method

System

Returns the SQL server's error code.

Syntax

```
errorNativeCode ( ) LongInt
```

Description

errorNativeCode returns the SQL server's error code. The SQL server's error code varies depending on the server and might overlap some Paradox error codes. If **errorCode** returns the constant *peGeneralSQL*, **errorNativeCode** returns the server's error code. **errorNativeCode** usually returns zero.

Example

The following example determines whether a server has error occurred. If a server error has occurred, the code displays the error code.

```
if errorCode() = peGeneralSQL then  
  message("SQL server error number " + string(errorNativeCode()))  
endIf
```

errorPop procedure

System

Removes the most recently added error code and error message from the error stack.

Syntax

```
errorPop ( ) Logical
```

Description

errorPop removes the most recently added error code and error message from the error stack. This procedure allows you to access the stack layer below the current layer.

Example

See the **errorLog** example.

errorShow procedure**System**

Displays the current error information in the Error log box.

Syntax

```
errorShow ( [ const topHelp String [ , const bottomHelp String ] ] ) Logical
```

Description

errorShow displays the current error information in the Error log box. The argument *topHelp* labels the top portion of the dialog box, and *bottomHelp* the bottom.

Example

The following example uses a button named *tryAnError* to log several errors onto the error stack, and uses **errorShow** to display them:

```
; 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
```

errorTrapOnWarnings procedure**System**

Specifies whether to handle warning errors as critical errors.

Syntax

```
errorTrapOnWarnings ( const yesNo Logical )
```

Description

errorTrapOnWarnings specifies whether to handle warning errors as critical errors. By default, warning errors are not trapped in a try...onFail block. If you set the argument *yesNo* to Yes, **errorTrapOnWarnings** traps warning errors as critical errors. This procedure affects only the active form.

Example

The following example attempts to open an invalid form. If **errorTrapOnWarnings** is set to yes, and error message is produced; otherwise, no message is produced.

```
; 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
```

commands and programs**System**

Executes a program or DOS command.

executeString method

Syntax

```
execute ( const programName String [ , const wait Logical [ , const displayMode
SmallInt ] ] ) Logical
```

Description

execute executes a program or DOS command. The argument *programName* specifies the program or DOS command to be launched. An optional argument named *wait* specifies whether ObjectPAL suspends execution until you close the program. An optional ExecuteOption named *displayMode* specifies the video display mode used when executing the command.

If you have to specify a path to *programName*'s directory, use double backslashes (\\) in the path names.

Note

- The *wait* statement is not valid when executed within the WindowsNT environment.

Example

The following example launches the Windows Clock application in the default window and waits for you to close it before resuming execution:

```
; showClock::pushButton
method pushButton(var eventInfo Event)
    execute("clock.exe", Yes, ExeShowNormal) ; execute Windows Clock
endMethod
```

executeString method

System

Converts a string to an ObjectPAL script and runs the script.

Syntax

```
executeString ( const scriptText String [ , const otherText String ] ) AnyType
```

Description

executeString converts a string to an ObjectPAL script and runs the script. This method inserts the string in the script's built-in run method. You can declare types, constants, and variables within the string. The optional *otherText* argument allows you to include ObjectPAL constructs (e.g., procedures or a Uses clause). The *otherText* argument refers to constructs included before the script's built-in run method.

To return a value from **executeString**, use **formReturn**.

If the string contains syntax errors, the Script window remains on the desktop.

Example

The following example calls a routine from Windows and runs it:

```
method run(var eventInfo Event)
var
    msgText, usesText string
endvar

; Note the backslash char protects quotes inside the quoted string
msgText = "MessageBox(0,\"A Message\", \"Hello World\" , 1)"

usesText = "Uses USER32
    MessageBox(hwnd CLONG,
                str1 CPTR,
                str2 CPTR,
                boxType CLONG) CLONG"
```

```

        endUses"
    ; Now display the message box
    executeString(msgText, usesText)
endMethod

```

exit procedure

System

Exits the Paradox application.

Syntax

```
exit ( )
```

Description

exit closes Paradox. If you try to exit Paradox without saving your changes, **exit** prompts you to save your work.

Example

The following example creates an Exit button which asks for confirmation and closes Paradox:

```

; btnExit::pushButton
method pushButton(var eventInfo Event)
    var
        stQuit String
    endVar

    stQuit = msgYesNoCancel("Exit", "Do you want to quit?")
    if stQuit = "Yes" then
        exit() ; If user chooses Yes, then exit.
    endIf
endMethod

```

expertsDir procedure

System

Returns the path of the registered experts directory.

Syntax

```
expertsDir () string
```

Description

expertsDir returns a string representing the path of the registered Paradox experts.

Example

The following example returns the path of the experts directory:

```

method run(var eventInfo Event)
    var
        str string
    endvar
    str = expertsDir()
    str.view("The experts directory is...")
endMethod

```

fail procedure

System

Causes a method to fail.

Syntax

```
fail ( [ const errorNumber SmallInt, const errorMessage String ] )
```

Description

fail causes a method to fail. Executing **fail** in the onFail section of a try...onFail block forces a jump to the next highest block (if one exists). **fail** then jumps to the implicit try...onFail block that ObjectPAL wraps around every method. Use an Errors constant or a user-defined error constant to set a value for **errorNumber**, which specifies an error code on failure. *errorMessage* (optional) specifies a displayed error message.

Example

The following example returns the path of the experts directory:

```
method run(var eventInfo Event)
  var
    str string
  endvar
  str = expertsDir()
  str.view("The experts directory is...")
endMethod
```

FileBrowser**System**

See **fileBrowserEx**.

fileBrowserEx replaces **fileBrowser** in this version. **fileBrowser** still functions, and can still be used when working with forms created using older versions of Paradox. However, only **fileBrowserEx** guarantees that the full filename, including its drive or alias, is returned.

fileBrowserEx procedure**System**

Displays the Paradox File Browser and returns the names of the files you select.

Syntax

```
1. fileBrowserEx ( var selectedFile String [ , var browserInfo FileBrowserInfo ] )
Logical
2. fileBrowserEx ( var selectedFiles Array[ ] String [ , var browserInfo
FileBrowserInfo ] ) Logical
```

Description

fileBrowserEx suspends ObjectPAL execution until you close the File Browser. This method returns True if you select at least one file; otherwise, it returns False (even if you click OK to close the dialog box).

Use Syntax 1 to return one filename in **selectedFile**. Use Syntax 2 to return an array of filenames in the resizable array **selectedFiles**.

In either syntax, you can provide an optional record that specifies the data that the File Browser displays. For example, you can instruct the File Browser to display Paradox tables only, forms only, forms and reports, and so on. ObjectPAL provides a special predefined Record structure called **FileBrowserInfo** that you use only with the **fileBrowserEx** procedure.

The following table displays the structure of **FileBrowserInfo**:

Field	Type	Description
Title	String	The dialog box title
Options	LongInt	Handling instructions for the filename that the user inputs
AllowableTypes	LongInt	The permitted file types, based on file extensions
SelectedType	LongInt	One of the allowable types
FileFilters	String	The file specification in the edit box
CustomFilter	String	One or more file masks in the Files Of Type list box. Each file mask contains the list box text, and the file mask. The two parts are separated by a delimiter character (). The mask can include any valid filename character, and the ? and * wildcard characters. To display all Paradox tables, use the following custom filter: "Paradox tables *.db ". To display a list box that allows you to display all Paradox tables or all dBASE tables, use the following custom filter: "Paradox tables *.db dBASE tables *.dbf ". The string's last character determines the delimiter.
Alias	String	The alias or drive name listed in the Alias box
Path	String	The path of the selected file or files. The value is returned by the File Browser and cannot be set directly.
Drive	String	The drive of the selected file or files. The value is returned by the File Browser and cannot be set directly.
DefaultExt	String	The default file extension. Use DefaultExt and NewFileOnly to allow users to omit the file extension when naming a new file.
PathOnly	Logical	The path only of the selected file or files, without filename.
NewFileOnly	Logical	If True, the File Browser behaves like the Save As dialog box. If False, the File Browser behaves like the Open dialog box.

This record structure is built into ObjectPAL. Simply declare a variable of type **FileBrowserInfo** and assign values to the fields in its structure.

When you call **fileBrowserEx**, values from the File Browser dialog box are inserted in the Alias, Path, and Drive fields. This allows you to determine what you selected in the File Browser.

The *AllowableTypes* field specifies what appears in the list box for the Types panel in the File Browser. The *SelectedType* field indicates which of the *AllowableTypes* is currently selected. Use **FileBrowserFileTypes** constants for values in the *SelectedType* and *AllowableTypes* fields.

Note

- If you prefer not to use the full **fileBrowserInfo** record structure, you can declare a record with your own selection of fbi fields and pass it to the **fileBrowserEx()** procedure.

Example 1

The following example calls **fileBrowserEx** twice. First, **fileBrowserEx** returns one filename. If that filename is a table name, **fileBrowserEx** opens a Table window. Next, **fileBrowserEx** returns an array of filenames and displays the array in a dialog box. The array of filenames is selected by pressing SHIFT and clicking files.

```
; fileBrowserExButton::pushButton
method pushButton(var eventInfo Event)
var
    oneFile      String
    manyFiles Array[] String
    tView        TableView
endVar
fileBrowserEx(oneFile) ; display the File Browser, and wait
                        ; for you to choose one file
                        ; variable oneFile stores the filename chosen
if isTable(oneFile) then
    tView.open(oneFile) ; open a Table window for the chosen file
endif

fileBrowserEx(manyFiles) ; let you select multiple files and store
                        ; the filenames in an array
manyFiles.view()         ; displays your choices
endMethod
```

Example 2

The following example uses a **FileBrowserInfo** record to pass information. Attach the following code to a button's built-in **pushButton** method. When it executes, this code displays the Browser, waits for you to choose a file, and displays information about your 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
fbi.CustomFilter = "(Bitmap image) *.bmp|*.bmp|(Other graphics files)
*.jpg;*.pcx|*.jpg;*.pcx||"

; Display the Browser and process your selection
if fileBrowserEx(selectedFile, fbi) then
    message("You selected ", selectedFile)
else
    message("You selected cancel")
endif

endMethod
```

formatAdd procedure**System**

Adds a format.

Syntax

```
formatAdd ( const formatName String, const formatSpec String ) Logical
```

Description

formatAdd adds a format. It creates a format named *formatName* which is described by **formatSpec**. The new format is available to the current session. This method returns True if successful; otherwise, it returns False.

Note

- **formatAdd** does not save Field width (Wn), Alignment (AR, AL, AC), and Case specifiers (CU, CL, CC) in the new format definition. However, save decimal precision (W.n) is preserved. 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 and 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,ENW, 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
```

formatDelete procedure**System**

Deletes a format.

Syntax

```
formatDelete ( const formatName String ) Logical
```

Description

formatDelete deletes the format specified by the argument *formatName* from the current session.

Note

- This procedure works only for custom formats.

Example

The following example deletes the custom format named *FourCurrency*:

```
; deleteAFormat::pushButton
method pushButton(var eventInfo Event)
if formatExist("FourCurrency") then
    formatDelete("FourCurrency")
else
    msgInfo("FYI", "Format was not found.")
endMethod
```

formatExist procedure

```
endIf  
endMethod
```

formatExist procedure

System

Reports whether a format exists.

Syntax

```
formatExist ( const formatName String ) Logical
```

Description

formatExist reports whether the format *formatName* is available in the current session. This method returns True if the format is available; otherwise, it returns False.

Example

The following example determines whether a custom format named *FourCurrency* exists.

If *FourCurrency* does not exist, the code adds the format specification and displays a number formatted in the new format.

```
; 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
```

formatGetSpec procedure

System

Returns the format specification for a named format.

Syntax

```
formatGetSpec ( const formatName String ) String
```

Description

formatGetSpec returns the format specification for the format specified by *formatName*. You can pass the return value to **formatStringToDate** and **formatStringToNumber** to format a string into a date or number.

Example

The following example uses **formatGetSpec** and **formatStringToDate** to assign a date to a Date type variable the Windows Long format:

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

```

var
    d Date
endVar

d = formatStringToDate("Friday, January 08, 1965", formatGetSpec("Windows Long"))
d.view()
endMethod

```

formatSetCurrencyDefault procedure

System

Sets the default display format for Currency values.

Syntax

```
formatSetCurrencyDefault ( const formatName String ) Logical
```

Description

formatSetCurrencyDefault sets the default display format for Currency values. This setting remains in effect throughout the session.

Example

See the **formatExist** example.

formatSetDateDefault procedure

System

Sets the default display format for Date values.

Syntax

```
formatSetDateDefault ( const formatName String ) Logical
```

Description

formatSetDateDefault sets the default display format for Date values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setDateFormat* to set the default display format for Date values to the Windows Long format. The code then displays a date in the new 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

```

formatSetDateTimeDefault procedure

System

Sets the default display format for DateTime values.

formatSetLogicalDefault procedure

Syntax

```
formatSetDateTimeDefault ( const formatName String ) Logical
```

Description

formatSetDateTimeDefault sets the default display format for DateTime values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setDateTimeFormat* to set the default display format for DateTime values. The code then uses view to display a DateTime value in the new 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
```

formatSetLogicalDefault procedure

System

Sets the default display format for Logical values.

Syntax

```
formatSetLogicalDefault ( const formatName String ) Logical
```

Description

formatSetLogicalDefault sets the default display format for Logical values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setLogicalFormat* to set the default display format for Logical values to the Male/Female format. The code then displays a logical value in the new 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
```

formatSetLongIntDefault procedure**System**

Sets the default display format for LongInt values.

Syntax

```
formatSetLongIntDefault ( const formatName String ) Logical
```

Description

formatSetLongIntDefault sets the default display format for LongInt values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setIntegerFormat* to set the default display format for LongInt values to the Integer format. The code then displays a long integer in the new 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
```

formatSetNumberDefault procedure**System**

Sets the default display format for Number values.

Syntax

```
formatSetNumberDefault ( const formatName String ) Logical
```

Description

formatSetNumberDefault sets the default display format for Number values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setNumberFormat* to set the default display format for Number values to the Scientific format. The code then displays a number 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 3.489283e+3.
else
    msgStop("Stop", "Requested format does not exist.")
endif
```

`formatSetSmallIntDefault` procedure

```
endIf  
endMethod
```

formatSetSmallIntDefault procedure

System

Sets the default display format for SmallInt values.

Syntax

```
formatSetSmallIntDefault ( const formatName String ) Logical
```

Description

formatSetSmallIntDefault sets the default display format for SmallInt values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setSmallIntFormat* to set the default display format for SmallInt values to the Integer format. The code then displays a small integer 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
```

formatSetTimeDefault procedure

System

Sets the default display format for Time values.

Syntax

```
formatSetTimeDefault ( const formatName String ) Logical
```

Description

formatSetTimeDefault sets the default display format for Time values. This setting remains in effect throughout the session.

Example

The following example uses the **pushButton** method for a button named *setTimeFormat* to set the default display format for Time values to the format hh:mm:ss am. The code then display a time in the new default format.

```
; setTimeFormat::pushButton  
method pushButton(var eventInfo Event)  
var  
    someTime Time  
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

```

formatStringToDate procedure

System

Uses a format specification to translate a String value to a Date value.

Syntax

```
formatStringToDate ( dateString String, formatSpec String ) Date
```

Description

formatStringToDate uses a format specification to translate a String value to a Date value. This method translates *dateString* (a string value representing a date) to a Date type value using to the format specification in *formatSpec*. This method returns the Date value and leaves the String value unmodified.

formatSpec is the format specification of a named format—not the format name itself. To retrieve the format specification of a named format, use **formatGetSpec**.

Example

The following example formats a String value as a valid date. The code is attached to the built-in `changeValue` method of an Alpha field and executes when you type a value and leave the field (e.g., press ENTER).

If the field object is bound to a Date field (instead of an Alpha field), Paradox validates the date without writing ObjectPAL code.

```

method changeValue(var eventInfo ValueEvent)
    var
        stUserDate    String
        daValidDate    Date
    endVar

    doDefault

    ; Assume user enters "09-94-23" into this Alpha field object.
    stUserDate = self.Value

    try
        ; Format your value as a valid date.
        daValidDate = formatStringToDate(stUserDate, "D0(%M-%Y-%D)")

        ; formatStringToDate does not change the String value.
        ; It returns a Date value. The following statement displays
        ;         You entered: 09-94-23
        ;         Valid date: 09/23/94

        msgInfo("You entered: " + stUserDate,
            "Valid date: " + String(daValidDate))

    onFail
        ; If user's value cannot be formatted as a date,
        ; display a message.
        msgStop(stUserDate, "Cannot format that value as a Date.")
    endTry
endMethod

```

formatStringToDateTime method

System

Translates a String value to a DateTime value.

Syntax

```
formatStringToDateTime ( const dateTimeString String, const formatSpec String )  
DateTime
```

Description

formatStringToDateTime translates **dateTimeString** to a DateTime value, using the format specification in **formatSpec**. If successful, **formatStringToDateTime** returns a DateTime value and leaves the **dateTimeString** value unmodified. The value of **formatSpec** must be the format specification of a named format—not the format name. To retrieve the format specification of a named format, use **formatGetSpec**.

Example

The following example converts the specified string to the DateTime data type and displays the result:

```
view( formatStringToDateTime( "23:59:59, 3/23/99", "TH10(%H:%M:%S, %D)" ) )
```

formatStringToNumber procedure

System

Uses a format specification to translate a String value to a Number value.

Syntax

```
formatStringToNumber ( numberString String, formatSpec String ) Number
```

Description

formatStringToNumber translates *numberString* (a string value that represents a number) to a Number value, using the format specification in *formatSpec*. If successful, this procedure returns the Number value and leaves the String value unmodified.

The value of *formatSpec* must be the format specification of a named format—not the format name. To retrieve the format specification of a named format, use **formatGetSpec**.

Example

In the following example, two strings are concatenated to form a number in scientific notation format. **formatStringToNumber** is used to assign the value to a Number variable and the formatted and unformatted values are displayed in a dialog box. **formatStringToNumber** assigns the formatted value to a Number variable, but leaves the String value unmodified.

```
;btnScientific :: pushButton  
method pushButton(var eventInfo Event)  
  var  
    st1,  
    st2,  
    stSciNot String  
    nuResult Number  
  endVar  
  
  st1 = "1.e"  
  st2 = "+2"  
  stSciNot = st1 + st2  
  nuResult = formatStringToNumber(stSciNot, "S-4")  
  
  ; The following statement displays  
  ; Before format: 1.e+2  
  ; After format: 100.00
```

```

    msgInfo("Before format: " + stSciNot,
            "After format: " + String(nuResult))
endMethod

```

formatStringToTime method

System

Translates a String value to a Time value.

Syntax

```
formatStringToTime (const timeString String, const formatSpec String ) Time
```

Description

formatStringToTime translates **timeString** to a Time value, using the format specification in **formatSpec**. If successful, **formatStringToTime** returns a Time value and leaves the String value unmodified. The value of **formatSpec** must be the format specification of a named format—not the format name. To retrieve the format specification of a named format, use **formatGetSpec**.

Example

The following example converts the specified string to the Time data type and displays the result:

```
view( formatStringToTime( "23:59:59", "TH10(%H:%M:%S)" ) )
```

getDefaultPrinterStyleSheet procedure

System

Returns the name of the default printer style sheet used by documents designed for the printer.

Syntax

```
getDefaultPrinterStyleSheet ( ) String
```

Description

getDefaultPrinterStyleSheet returns the name of the default printer style sheet used by documents designed for the printer.

Use **getStyleSheet** and **setStyleSheet** for forms and reports that use different style sheets.

Use **getDefaultScreenStyleSheet** to retrieve the default screen style sheet. This screen style sheet is used when you create design documents for the screen.

Note

- Printer style sheet files have an .FP extension and screen style sheet files have and .FT the extension. Printer and screen style sheets are not interchangeable.

Example

See the **setDefaultPrinterStyleSheet** example.

getDefaultScreenStyleSheet procedure

System

Returns the name of the default screen style sheet used by design documents that are created for the screen.

Syntax

```
getDefaultScreenStyleSheet ( ) String
```

Description

getDefaultScreenStyleSheet returns the full path and filename of the default style sheet for screen documents (e.g., C:\COREL\PARADOX\COREL.FT).

`getDesktopPreference` procedure

Use `getStyleSheet` and `setStyleSheet` for forms and reports that use different style sheets.

Use `getDefaultPrinterStyleSheet` to retrieve the name of the default printer style sheet, used whenever you create design documents that are designed for the printer.

Note

- Printer style sheet files have an .FP extension and screen style sheet files have and .FT the extension. Printer and screen style sheets are not interchangeable.

Example

See the `setDefaultScreenStyleSheet` example.

`getDesktopPreference` procedure System

Retrieves a desktop preference value.

Syntax

```
getDesktopPreference (const section AnyType, const name AnyType) AnyType
```

Description

`getDesktopPreference` returns the value of the desktop preference specified by the section and name arguments. The value returned corresponds to one of the DesktopPreferenceTypes Constants.

Example

The following example displays the sets the title name preference and then retrieves and displays the name:

```
method pushButton(var eventInfo Event)
setDesktopPreference( PrefProjectSection, prefTitleName,"Paradox pour Windows" )

x = getDesktopPreference( PrefProjectSection, prefTitleName )

x.view()
endmethod
```

`getLanguageDriver` procedure System

Returns the default language driver name for the system.

Syntax

```
getLanguageDriver ( ) String
```

Description

`getLanguageDriver` returns the default language driver name for the system.

Example

The following example displays the system's language driver name on the Status Bar:

```
;btnDefaultDriver :: pushButton
method pushButton(var eventInfo Event)
    message(getLanguageDriver())
endmethod
```

`getMouseScreenPosition` procedure System

Returns the mouse position as a Point data type.

Syntax

```
getMouseScreenPosition ( ) Point
```

Description

getMouseScreenPosition returns the coordinates (in twips) of the pointer relative to the screen. Use Point type methods (e.g., x and y) to retrieve more information.

getMouseScreenPosition retrieves the mouse position at the precise time of an event. The coordinates of the current mouse position might be different.

Example

In the following example, the mouse moves one inch down and one inch to the left when you click the nervousMouse button:

```
; nervousMouse::pushButton
method pushButton(var eventInfo Event)
var
    mouseP,
    newMouseP Point
endVar
mouseP = getMouseScreenPosition()
newMouseP = mouseP + Point(1440, 1440)
setMouseScreenPosition(newMouseP) ; move pointer 1 inch down and
                                   ; 1 inch to the right
endMethod
```

getRegistryValue method**System**

Retrieves a registry value.

Syntax

```
getRegistryValue ( const key String, const value String , const rootKey LongInt )
AnyType
```

Description

getRegistryValue retrieves data from a specified key and value in the registry. If **getRegistryValue** is successful, the registry value is returned as an AnyType; otherwise, it returns an empty string.

key is a path similar to a file path. However, wildcards are not expanded in the key. **key** cannot contain a single backslash and cannot be empty. Its size is limited to 65,534 bytes.

The value is a string that is limited to 65,534 bytes. **value** can contain backslashes and can be empty.

rootKey is analogous to a directory drive. Set **rootKey** with the predefined RegistryKeyType Constants.

Example

The following example retrieves the current ObjectPAL Level from the registry and displays it:

```
var
    strLevel    String
endvar

strLevel = getRegistryValue( "Software\\Corel\\Paradox\\8.0\\Pdoxwin\\Properties",
"Level",
    RegKeyCurrentUser )
strLevel.view()
```

getUserLevel procedure**System**

Returns your ObjectPAL level property setting (Advanced or Beginner).

Syntax

```
getUserLevel ( ) String
```

Description

getUserLevel returns Advanced or Beginner to specify your ObjectPAL level property setting. Use **setUserLevel** to change this setting.

Note

- The ObjectPAL level property setting does not affect code execution. The setting only affects the ObjectPAL language elements that are displayed in the user interface.

Example

See the **setUserLevel** example.

helpOnHelp procedure**System**

Displays information about using the Windows Help system and opens Help if necessary.

Syntax

```
helpOnHelp ( ) Logical
```

Description

helpOnHelp opens the WINHLP32.HLP file by default.

To open another Help file

- 1 Open the Help project file in a text editor.
- 2 Add a SetHelpOnFile macro to the [CONFIG] section, specifying the Help file you want to use in How to Use Help.
- 3 Compile the Help file.

The following macro, when placed in the [CONFIG] section of the Help project file changes the Help file, causes helpOnHelp to open:

```
[CONFIG]
SetHelpOnFile("howhelp.hlp")
```

Example

The following example opens a Help file when you click Help, Help On Help from a custom menu:

```
method menuAction(var eventInfo MenuEvent)
  var
    siMenuChoice SmallInt
  endVar

  siMenuChoice = eventInfo.id()

  switch
    case siMenuChoice = UserMenu + MenuHelpOnHelp :
      helpOnHelp()
      ; Handle other cases here
    endSwitch
endmethod
```

helpQuit procedure

System

Notifies the Help application that it is no longer needed by the current application.

Syntax

```
helpQuit ( const helpFileName String ) Logical
```

Description

helpQuit notifies the Windows Help application (WINHELPEXE) that the Help file **helpFileName** is no longer needed by the current Paradox application. If the directory where **helpFileName** resides is not specified in the path, you must specify its full path. If no other applications require the Help application, Windows closes it.

Example

The following example executes when you choose an item from a custom menu. If you click File, Close Form, **helpQuit** notifies the Help application that it is no longer needed and closes the current form.

```
method menuAction(var eventInfo MenuEvent)
  const
    ; Typically, menu choice constants are defined elsewhere,
    ; with the rest of the menu-building code. The following
    ; constant is defined here so the example will compile.
    kMyMenuFileCloseForm = 104
  endConst

  var
    siMenuChoice   SmallInt
    stHelpFileName String
  endVar

  siMenuChoice = eventInfo.id()
  stHelpFileName = "c:\pdxapps\ordentry\ordentry.hlp"

  switch
    case siMenuChoice = UserMenu + kMyMenuFileCloseForm :
      helpQuit(stHelpFileName) ; Tell Help we don't need it any more.
      close() ; Close the form.
    ; Handle other cases here
  endSwitch
endMethod
```

helpSetIndex procedure

System

Specifies what help file will be used as the Help contents (index).

Syntax

```
helpSetIndex ( const helpFileName String, const indexId LongInt ) Logical
```

Description

helpSetIndex specifies what help file will be used as the Help contents (index). This procedure instructs the Windows Help application (WINHELPEXE) to use the topic in **helpFileName** (specified by **indexId**) as the Contents topic. If **helpFileName** does not reside in the directory specified in your path, you must specify the full path or the directory.

When you open a Help file, WinHelp displays the Contents topic by default. When you create a Help file, you specify the Contents topic using the Contents option in the [CONFIG] section of the Help

helpShowContext procedure

project file. For example, when placed in the project file's [CONFIG] section, the following SetContents macro sets the Contents topic for a Help file to topic number 100 in CWH.HLP.

```
[CONFIG]
SetContents("cwh.hlp", 100)
```

If you do not use the *SetContents* option, the Contents topic is the first topic in the first file listed in the [FILES] section of the Help project file.

You can use **helpSetIndex** to specify a Contents topic from within an application.

Example

The following example sets the Contents topic for a Help file to topic number 100 in the file ORDENTRY.HLP:

```
method setHelpContents() Logical
    return helpSetIndex("c:\pdxapps\ordentry\ordentry.hlp", 100)
endMethod
```

helpShowContext procedure

System

Displays the Help topic specified by helpId in the file helpFileName .

Syntax

```
helpShowContext ( const helpFileName String, const helpId LongInt ) Logical
```

Description

helpShowContext instructs the Windows Help application to search **helpFileName** for the topic identified by helpId; and to display the topic. If the directory where **helpFileName** resides is not in your path, you must specify its full path.

In a Help source file, each topic is identified by a context ID. A context ID is a string defined by a # footnote. The context ID is mapped to an integer value in the [MAP] section of the Help project file (.HPJ). helpShowContext uses this mapped integer value to locate the Help topic.

Example

The following example instructs the Windows Help application to display context-sensitive Help for the active object in a form. Assume that the form contains three buttons and two field objects. The code is attached to a button whose TabStop property is set to False. If the code is attached to a button whose TabStop property is set to True, the button becomes active when clicked.

```
helpButton::pushButton
const
; These integer values must also be listed
; in the [MAP] section of the Help project file.
    kNewOrdBtn   = LongInt(1020)
    kEditOrdBtn  = LongInt(1021)
    kDelOrdBtn   = LongInt(1022)
    kCustNameFld = LongInt(2020)
    kOrderNoFld  = LongInt (2021)
endConst

method pushButton(var eventInfo Event)

var
    stObjName,
    stHelpFileName String
    liContextId   LongInt
endVar
```



```

stObjName = active.name ; Get the name of the active object.
stHelpFileName = "c:\\pdoxapps\\ordentry\\ordentry.hlp"

switch
  case stObjName = "newOrdBtn"      : liContextId = kNewOrdBtn
  case stObjName = "editOrdBtn"     : liContextId = kEditOrdBtn
  case stObjName = "delOrdBtn"      : liContextId = kDelOrdBtn
  case stObjName = "custNameFld"    : liContextId = kCustNameFld
  case stObjName = "orderNoFld"     : liContextId = kOrderNoFld
endSwitch

if not helpShowContext(stHelpFileName, liContextId) then
  errorShow("Could not display Help topic.")
endif

endMethod

```

helpShowIndex procedure

System

Displays the contents topic (index) of a specified Help file.

Syntax

```
helpShowIndex ( const helpFileName String ) Logical
```

Description

helpShowIndex instructs the Windows Help application (WINHELPEXE) to display the Contents topic (index) in the Help file specified by **helpFileName**. If the directory where **helpFileName** resides is not on your path, you must specify its full path.

When you open a Help file, WinHelp displays the Contents topic by default. When you create a Help file, you specify the Contents topic using the Contents option in the [CONFIG] section of the Help project file. For example, when placed in the project file's [CONFIG] section, the following SetContents macro sets the Contents topic for a Help file to topic number 100 in CWH.HLP.

```
[CONFIG]
SetContents("cwh.hlp", 100)
```

If you do not use the *SetContents* option, the Contents topic is the first topic in the first file listed in the [FILES] section of the Help project file.

Example

The following example executes when you choose an item from a custom menu. If you click Help, Contents, **helpShowIndex** instructs the Help application to display the Contents topic for the specified Help file.

```

method menuAction(var eventInfo MenuEvent)
  const
    ; Typically, menu choice constants are defined elsewhere,
    ; with the rest of the menu-building code. The following
    ; constant is defined here so the example will compile.
    kMyMenuHelpContents = 501
  endConst

  var
    siMenuChoice  SmallInt
    stHelpFileName String
  endVar

  siMenuChoice = eventInfo.id()

```

helpShowTopic procedure

```
stHelpFileName = "c:\\pdoxapps\\ordentry\\ordentry.hlp"

switch
  case siMenuChoice = UserMenu + kMyMenuHelpContents :
    helpShowIndex(stHelpFileName) ; Display the Contents topic.
    ; Handle other cases here
endSwitch

endMethod
```

helpShowTopic procedure

System

Displays help for a specified context ID.

Syntax

```
helpShowTopic ( const helpFileName String, const topicKey String ) Logical
```

Description

helpShowTopic instructs the Windows Help application to search the file **helpFileName** for the topic associated with *topicKey*, and to display the topic. If the directory where **helpFileName** resides is not on your path, you must specify its full path. *topicKey* must match a keyword defined by a *K* footnote in the Help source file. If *topicKey* does not match a keyword, the search fails and the Windows Help application displays an error message.

Example

The following example prompts you to type a word or phrase and then searches for the text in the specified Help file:

```
method pushButton(var eventInfo Event)
  var
    stHelpFileName,
    stTopicKey,
    stPromptText    String
  endVar

  stHelpFileName = "c:\\pdoxapps\\ordentry\\ordEntry.hlp"
  stPromptText   = "Enter a word or phrase here."
  stTopicKey      = stPromptText

  stTopicKey.view("Enter text to search for.")
  if stTopicKey stPromptText then
    helpShowTopic(stHelpFileName, stTopicKey)
  endIf
endMethod
```

helpShowTopicInKeywordTable procedure

System

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

helpShowTopicInKeywordTable instructs the Windows Help application to search the file **helpFileName** for the topic associated with **keyTableLetter** and **topicKey**, and to display the topic. If the directory where **helpFileName** resides is not in your path, you must specify its full path. The

value of **keyTableLetter** must match a multi-key index specified in the [OPTIONS] section of the Help project file. For example, if a Help project file includes the following code, assign *L* to **keyTableLetter**.

```
[OPTIONS]
MULTIKEY=L
```

The value of **topicKey** must match a keyword defined using a multi-key index footnote in the Help source file. If topicKey does not match, the search fails and the Windows Help application displays an error message.

Example

The following example prompts you to type PARADOX or dBASE and then searches for field types in the keyword table of the specified Help file. Assume that an application is handling a user's request for Help on the topic field types.

```
method pushButton(var eventInfo Event)
    var
        stHelpFileName,
        stPromptText,
        stUserChoice,
        stTopicKey,
        stKeyTableLetter    String
    endVar

    stHelpFileName    = "c:\pdxapps\ordentry\ordEntry.hlp"
    stPromptText      = "Enter PARADOX or dBASE here."
    stUserChoice      = stPromptText
    stTopicKey        = "field types"

    stUserChoice.view("Do you want Paradox Help or dBASE Help?")
    if stUserChoice = stPromptText then
        switch
            case stUserChoice = "PARADOX" : stKeyTableLetter = "P"
            case stUserChoice = "dBASE"   : stKeyTableLetter = "D"
            otherwise : return
        endSwitch

        helpShowTopicInKeywordTable(stHelpFileName, stKeyTableLetter, stTopicKey)
    endIf
endMethod
```

isErrorTrapOnWarnings procedure

System

Reports whether this session handles warning errors as critical errors.

Syntax

```
isErrorTrapOnWarnings ( ) Logical
```

Description

isErrorTrapOnWarnings reports whether this session handles warning errors as critical errors. This method returns True if the active session treats warning errors as critical errors; otherwise, it returns False.

Example

The following example uses the pushButton method for **btnToggleWarning** to toggle between critical and non-critical warning errors:

isMousePersistent method

```
; btnToggleWarning :: pushButton
method pushButton(var eventInfo Event)
    errorTrapOnWarnings(not isErrorTrapOnWarnings())
    msgInfo("Warning errors are critical", isErrorTrapOnWarnings())
endmethod
```

isMousePersistent method

System

Reports if mouse persistence is turned on.

Syntax

```
isMousePersistent ( ) Logical
```

Description

isMousePersistent reports if mouse persistence is on. **isMousePersistent** returns True if mouse persistence is turned on, and False if mouse persistence is turned off. To set mouse persistence, use **setMouseShape** or **setMouseShapeFromFile**.

Example

In the following example, a form has two buttons: *btnNonPersistent* and *btnPersistent*. The **pushButton** method for each button uses **setMouseShape()** to set the mouse shape of the cursor. The first button has mouse persistence turned off, and the second button had mouse persistence turned on. The second button, *btnPersistent*, also contains a **mouseEnter** method which uses **isMousePersistent()** to evaluate the persistency of the cursor and revert it to its original state. When the first button is pressed, the pointer changes. However, when the cursor moves off the button, it reverts to its original setting. When the second button is pressed, the cursor changes and remains unmodified until the cursor moves back over the second button. This triggers the mouseEnter method of the second button and reverts the cursor back to its original state.

The following code is attached to the **pushButton** method for *btnNonPersistent*:

```
; btnNonPersistent::pushButton
method pushButton(var eventInfo Event)
    ; Set the shape to MouseWait and persistence to False
    setMouseShape(MouseWait,FALSE)
endMethod
```

The following code is attached to the **pushButton** method for *btnPersistent*:

```
; btnPersistent::pushButton
method pushButton(var eventInfo Event)
    ; Set the shape to MouseWait and persistence to TRUE
    setMouseShape(MouseWait,TRUE)
endMethod
```

The following code is attached to the **mouseEnter** method for *btnPersistent*:

```
; btnPersistent::mouseEnter
method mouseEnterpushButton(var eventInfo MouseEvent)
    if isMousePersistent() then
        ; If it's persistent, set it back to the arrow cursor
        setMouseShape(MouseArrow,FALSE)
    endIf
endMethod
```

isTableCorrupt method

System

Determines if the specified file is corrupt.

Syntax

```
isTableCorrupt ( const tableName String) Logical or
isTableCorrupt ( const tableName Sting, const errorName String) Logical
```

Description

isTableCorrupt assumes the current working directory, although it can be overridden by a passing alias. You do not have to supply a file extension, as the method will assume a Paradox Table. The argument *tableName* is the name of the file and the argument *errorName* will produce a table with the error list.

Example

The following example determines if a specified file is corrupt, and if so, displays an error message:

```
method run ( var eventInfo Event)
  var
    I Logical
  endvar

  if isTableCorrupt ("bugs.db") then
    msgInfo("Warning", "This table is corrupt")
  endif
endmethod
```

message procedure**System**

Displays a message composed of up to six strings in the status line.

Syntax

```
message ( const message String [ , const message String ] * )
```

Description

message displays a message composed of up to six strings in the status line.

Example

The following example writes a message to the status line:

```
; showMessage::pushButton
method pushButton(var eventInfo Event)
  var
    lastName, firstName String
  endVar
  lastName = "Corel"
  firstName = "Frank"
  message("Hello, my name is ", firstName, " ", lastName, ".")
endMethod
```

msgAbortRetryIgnore procedure**System**

Displays a dialog box containing a message and the Abort, Retry, and Ignore buttons.

Syntax

```
msgAbortRetryIgnore ( const caption String, const text String ) String
```

Description

msgAbortRetryIgnore displays a three-button dialog box, where *caption* specifies the text in the Title Bar and *text* specifies the message. The return value is a mixed upper and lowercase string, that corresponds to the button you click.

msgInfo procedure

Example

The following example uses the showAbortRetryIgnore button to warn you that an operation may take a long time and asks you 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() ; Display your choice.

; Display a message based on your choice.
switch
  case doThis = "Abort" : message("Aborting operation.")
  case doThis = "Retry" : message("Retrying operation.")
  case doThis = "Ignore" : message("Ignoring problem.")
endSwitch
endMethod
```

msgInfo procedure

System

Displays a one-button dialog box containing the information icon, a caption and message, and an OK button.

Syntax

```
msgInfo ( const caption String, const text String )
```

Description

msgInfo displays a one-button dialog box containing the information icon, a caption and message, and an OK button. *caption* is displayed in the Title Bar, and *text* is displayed in the box. Click OK or press ESC to close the dialog box. This procedure does not return a value.

Example

The following example uses the **msgInfo** method displays a message:

```
; showMsgInfo::pushButton
method pushButton(var eventInfo Event)
msgInfo("Trivia", "The capital of Oregon is Salem.")
endMethod
```

msgQuestion procedure

System

Displays a dialog box containing a caption and message, a question mark icon, and Yes and No buttons.

Syntax

```
msgQuestion ( const caption String, const text String ) String
```

Description

msgQuestion displays a dialog box containing a caption and message, a question mark icon, and Yes and No buttons. It displays *caption* in the Title Bar, and *text* in the box itself. This procedure returns your selection (Yes or No) in mixed upper and lowercase.

Example

The following example asks you whether to change the desktop title. If you choose Yes, the desktop title is changed and 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 desktop title.
        sleep(2000) ; Pause.
        thisApp.setTitle("Paradox for Windows") ; Restore it.
    case userChoice = "No" :
        message("Application title not changed.")
endSwitch
endMethod
```

msgRetryCancel procedure**System**

Displays a dialog box containing a caption, a message, and the Retry and Cancel buttons.

Syntax

```
msgRetryCancel ( const caption String, const text String ) String
```

Description

msgRetryCancel displays a dialog box containing a caption, a message, and the Retry and Cancel buttons. The argument *caption* specifies the text in the dialog box's Title Bar. *text* specifies the message displayed. **msgRetryCancel** returns your selection (Retry or Cancel). If you press ESC or select Close, returns Cancel. Values are returned in mixed upper and lowercase.

Example

The following example poses a question and confirms your response on the status line:

```
; 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
```

msgStop procedure**System**

Displays a dialog box containing a stop sign icon, a caption and message, and an OK button.

Syntax

```
msgStop ( const caption String, const text String )
```

msgYesNoCancel procedure

Description

msgStop displays a dialog box containing a stop sign icon, a caption and message, and an OK button. It displays caption in the Title Bar, and text and a Stop icon in the box itself. Click OK or press ESC to close the dialog box. This procedure does not return a value.

Example

The following example uses the **pushButton** method for **showMsgStop** to alert you to 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
```

msgYesNoCancel procedure

System

Displays a dialog box containing a caption, a message and the Yes, No, and Cancel buttons.

Syntax

```
msgYesNoCancel ( const caption String, const text String ) String
```

Description

msgYesNoCancel displays a dialog box containing a caption, a message and the Yes, No, and Cancel buttons. The argument *caption* specifies the text in the dialog box's Title Bar. *text* specifies the message displayed. **msgYesNoCancel** returns your selection (Yes, No or Cancel) in mixed upper and lowercase. If you press ESC or select Close, this procedure returns Cancel.

Example

The following example uses **msgYesNoCancel** to ask you whether to save the data before quitting, to discard the data, or to cancel the quit the operation:

```
; 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
```

pixelsToTwips procedure

System

Converts the screen coordinates from pixels to twips.

Syntax

```
pixelsToTwips ( const pixels Point ) Point
```

Description

pixelsToTwips converts the screen coordinates from pixels to twips.

Example

The following example uses the object variable `self` show the position of the button in twips and in pixels. This code displays the screen resolution in pixels opens 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
```

play procedure**System**

Plays a standalone script.

Syntax

```
play ( const scriptName String ) AnyType
```

Description

play executes **scriptName** to play a standalone script. To return a value from a script, call **formReturn** from within the script.

For more information, refer to the Script type.

Example

The following example plays a script called TESTSCR.SSL, which resides in the working directory:

```
; playAScript::pushButton
method pushButton(var eventInfo Event)
play("Testscr.ssl")
endMethod
```

printerGetInfo procedure**System**

Retrieves information about the printer installed on your system.

Syntax

```
printerGetInfo ( var printInfo PrinterInfo ) Logical
```

Description

printerGetInfo assigns printer information to *printInfo*, a record that you declare using a special ObjectPAL data type named PrinterInfo. The following table displays the structure of PrinterInfo:

printerGetOptions procedure

Field	Type	Description
DriverName	String	Name of the printer driver (e.g., PSCRIPT.DRV)
DeviceName	String	Name that identifies the printer type (e.g., Apple LaserWriter Plus)
PortName	String	Name of the printer port (e.g., LPT1)
DefaultPrinter	Logical	Determines whether the current printer is the default

This procedure returns True if successful; otherwise, it returns False.

Example

See the **printerSetOptions** example.

printerGetOptions procedure

System

Retrieves information about your system printer's settings.

Syntax

1. `printerGetOptions (var printOptions PrinterOptionInfo) Logical`
2. `printerGetOptions (var printerInfo DynArray[] AnyType) Logical`

Description

printerGetOptions assigns printer information to `printInfo`. *PrintInfo* is a variable you declare as an ObjectPAL record with a predefined structure called *PrinterOptionInfo*.

printerGetOptions assigns printer information to `printInfo`, a record you declare as an ObjectPAL data type *PrinterOptionInfo*.

Syntax 2 fills an array named *printerInfo* with supported print options.

This procedure returns True if successful; otherwise, it returns False.

Example

The following example sets the current printer settings and determines whether the printer is using a large format paper source:

```
method pushButton(var eventInfo Event)
var
    recUserOptions,
    recMyOptions    PrinterOptionInfo
endVar

; Get the current printer settings.
printerGetOptions(recUserOptions)
if recUserOptions.DefaultSource = prnLargeFmt then
    return
endif

; Specify new printer settings. prnLargeFmt is a PrintSources constant.
recMyOptions.DefaultSource = prnLargeFmt

if printerSetOptions(recMyOptions) then
    message("Printer setup complete.")
else
    errorShow()
```

```

    endIf
endMethod

```

PrinterOptionInfo record structure

System

Field	Type	Description
Orientation	LongInt	Paper orientation (portrait or landscape). Use a PrinterOrientation constant to test the value.
PaperSize	LongInt	Paper size. Use a PrinterSizes constant to test the value.
PaperWidth	LongInt	Custom paper width in twips (maximum of 64K twips). This value is converted internally to the tenths of a millimeter required by Windows.
PaperLength	LongInt	Custom paper length in twips (maximum of 64K twips). This value is converted internally to the tenths of a millimeter required by Windows.
Scale	LongInt	Scaling factor in percent. A scale value of 50 reduces the original to one-half its size. A value of 200 increases the original to twice its size. Scaling only applies to printers that support scaling for all functions, graphics, and fonts (e.g., Postscript printers and the Microsoft Windows Printing System).
Copies	LongInt	Number of copies for the printer to make. The Copies option works only with page printers (e.g., laser printers) where the full page can be held in printer memory. Some printer drivers support this feature on printers that cannot do full page printing. The Copies setting is equivalent to unchecking the Collate button in the Print File dialog box. Output is not collated. This operation is faster than repeatedly sending the full document to the printer, but requires hand sorting at completion.
DefaultSource	LongInt	Bin, tray, or feeder used by the default printer. Use a PrintSources constant to test the value.
PrintQuality	LongInt	Higher print qualities are used for final output, and lower print qualities for draft output. Lower quality prints differ significantly from the preview appearance of the document. Use a PrintQuality constant to test the value.
Color	LongInt	Sets color printers to color or monochrome printing. Monochrome printing is usually faster. Use a PrintColor constant to test the value.

printerSetCurrent procedure

Duplex

LongInt

Double-sided printing. Some printer drivers can support double-sided printing on otherwise single-sided printers by making two passes over the document. Use a PrintDuplex constant to test the value.

printerSetCurrent procedure

System

Sets the active printer on your system.

Syntax

```
printerSetCurrent ( printerInfo String ) Logical
```

Description

printerSetCurrent sets the active printer on your system. The argument *printerInfo* specifies the printer name, driver name, and printer port (separated by commas). For example, if the printer name is Postscript Printer, the driver is PSCRIPT.DRV, and the port is LPT1, the following code applies:

```
PostScript Printer,pscript,LPT1:
```

This procedure returns True if successful; otherwise, it returns False.

Example

The following example searches the available printers and looks for a specific driver in order to set the current printer:

```
method pushButton(var eventInfo Event)
  var
    arPrnNames, arPrinters array[]anytype
    stDrvName string
  endVar

  ;stDrvName is the name of the driver you want to use when setting the current
  printer
  stDrvName = "HP LaserJet 5MP"

  enumPrinters(arPrinters) ; Get a list of installed printers.
  arPrinters.view() ; View the above List

  ;search available printers looking for the correct one
  for i from 1 to arPrinters.size()
    stPrnInfo = arPrinters[i]
    stPrnInfo.breakApart(arPrnNames, ",")
    ;After breakapart array item 1 is the printer
    ;array 2 is the driver name

    if
      arPrnNames[2] = stDrvName
    then
      if
        printerSetCurrent(stPrnInfo)
      then
        msgInfo("Current Printer:", arPrnNames[1])
      else
        errorshow()
      endif
    endif
  return
endif
endfor
```

```

    msginfo("Current Printer:", "Could not find printer driver " + stDrvName)
endMethod

```

printerSetOptions procedure

System

Specifies settings for your system printer.

Syntax

1. `printerSetOptions (PrintOptions PrinterOptionInfo) Logical`
2. `printerSetOptions (var printerInfo DynArray[] AnyType [const override Logical]) Logical`

Description

printerSetOptions specifies settings for your system printer. *printerSettings* is a record of the special ObjectPAL data type `PrinterOptionInfo` that you must declare. You don't have to specify values for each field in a `PrinterOptionInfo` record. The printer substitutes its current setting for any value you don't specify.

Syntax 2 uses an array named *printerInfo* (obtained with **printerGetOptions**) to send the printer settings for only those options that the printer supports. The optional *override* argument tells **printerSetOptions** to override printer settings specified in the Form or Report level.

printerSetOptions returns True if successful; otherwise, it returns False. If you specify a value that doesn't apply to your printer, this method returns False.

Example

The following example prompts you to specify the number of copies of a report to print, sets up the printer, and prints the copies:

```

method pushButton(var eventInfo Event)
    var
        siNCopies    SmallInt
        stPrompt      String
        prnOptions    PrinterOptionInfo
        reOrders      Report
    endVar

    siNCopies = 0
    stPrompt = "Print how many copies?"

    siNCopies.view(stPrompt)
    if siNCopies < 0 then
        prnOptions.Copies = siNCopies
    else
        return
    endIf

; Use constant to specify lower paper tray.
    prnOptions.DefaultSource = prnLower

; Use constant to specify landscape (long) orientation.
    prnOptions.Orientation = prnLandscape

; Use constant to specify high quality print.
    prnOptions.PrintQuality = prnHigh

    if printerSetOptions(prnOptions) then
        reOrders.print("orders")
    else

```

projectViewerClose procedure

```
        errorShow("Could not set printer options.")
    endIf

endMethod
```

projectViewerClose procedure

System

Closes the Project Viewer window.

Syntax

```
projectViewerClose ( ) Logical
```

Description

projectViewerClose closes the Project Viewer window. This procedure returns True if successful; otherwise, it returns False.

Example

The following example calls **projectViewerIsOpen** to determine whether the Project Viewer window is open. If the Project viewer is open, this code closes it.

```
method open(var eventInfo Event)
    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:
        if projectViewerIsOpen() then
            projectViewerClose()
        endIf
    endIf
endMethod
```

projectViewerIsOpen procedure

System

Tells whether the Project Viewer window is open.

Syntax

```
projectViewerIsOpen ( ) Logical
```

Description

projectViewerIsOpen determines whether the Project Viewer window is open. This procedure returns True if the Project Viewer window is open; otherwise, it returns False.

Example

See the **projectViewerClose** example.

projectViewerOpen procedure

System

Opens the Project Viewer window.

Syntax

```
projectViewerOpen ( ) Logical
```

Description

projectViewerOpen opens the Project Viewer window. This procedure returns True if successful; otherwise, it returns False.

Example

The following example calls **projectViewerIsOpen** to determine whether the Project Viewer window is open. If the Project viewer is open, this code closes it.

```
method open(var eventInfo Event)
    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:
        if not projectViewerIsOpen() then
            projectViewerOpen()
        endIf
    endIf
endMethod
```

readEnvironmentString procedure**System**

Reads an item from the Paradox copy of the DOS environment.

Syntax

```
readEnvironmentString ( const key String ) String
```

Description

readEnvironmentString returns a string containing information about the DOS environment variable specified by key. When you launch Paradox it makes a copy of the DOS environment.

readEnvironmentString reads that copy and compiles information in a string. Changes made to DOS environment variables after Paradox is launched are not read by this procedure.

The DOS command SET assigns values to the environment variables. These values control the appearance and function of DOS and some batch files. Commonly used environment variables include PATH, PROMPT, and COMSPEC. For more information, see the SET command your DOS manuals, especially .

Example

The following example uses **readEnvironmentString** to retrieve the value of the PATH environment variable. The code then uses **writeEnvironmentString** to change it.

```
; 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 = getaliaspath(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
```

readProfileString procedure

```
pathArr.view("An array of paths")      ; view the results
endMethod
```

readProfileString procedure

System

Returns a value from a specified section of a file.

Syntax

```
readProfileString ( const fileName String, const section String, const key String )
String
```

Description

readProfileString returns a value from a specified section of a file. By default this procedure searches the WINDOWS directory. You can also use this method to read your WIN.INI file, so *fileName* would be WIN.INI.

Each section header in WIN.INI is bounded by square brackets on a separate line (e.g., [windows]). To specify a section, omit the brackets (e.g., use windows). In each section, a value marker is followed by an equal sign (e.g., Beep =). The equal sign is not required when you specify the value of key.

Example

The following example uses **readProfileString** to retrieve the setting for the Windows beep, and **writeProfileString** to change the setting:

```
; 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
```

resourceInfo procedure

System

Lists the system resources.

Syntax

```
resourceInfo ( var info DynArray[ ] AnyType )
```

Description

resourceInfo writes system resource data to *info*. *Info* is a dynamic array (DynArray) that you declare and pass as an argument.

The following table displays the information returned in *info*:

Index	Definition
DiskAvail	Available disk space on the current drive
DiskTotal	Total disk space on the current drive
FreeGdiResources	Percentage of free Windows GDI resources. This item is not supported in the 32-bit Windows environment.)
FreeSpace	Free Windows memory
FreeSystemResources	Percentage of free Windows system resources. This item is not supported in the 32-bit Windows environment.
FreeUserResources	Percentage of free Windows user resources. This item is not supported in the 32-bit Windows environment.
InternalVersion	Paradox internal Borland Database Engine (BDE) version
MemoryLoad	Percent of memory in use
MemPhysicalTotal	Total physical memory
MemPhysicalFree	Available physical memory
MemPageFileTotal	Total page/file memory
MemPageFileFree	Available page/file memory
MemVirtualTotal	Total virtual memory
MemVirtualFree	Available virtual memory

Example

The following example writes resource information to a dynamic array named *dyn* and then displays *dyn* in a View dialog box:

```

; btnResourceInfo::pushButton
method pushButton(var eventInfo Event)
var
    dynResources Dynarray[] String
endVar

    resourceInfo(dynResources)
    dynResources.view()
endmethod

```

runExpert procedure

System

Runs a registered Paradox expert or if the expert cannot be found, launches Install As You Go dialog box.

Syntax

```
runExpert ( const expertType String, const expertName String )
```

searchRegistry procedure

Description

runExpert runs a registered Paradox expert. If the expert you attempt to run is not found by Paradox, **runExpert** launches the Install As You Go dialog box. The **expertName** argument specifies which expert to run. The **expertType** parameter determines the type of experts to list. ObjectPAL provides **ExpertTypes** constants for this purpose.

Example

The following example runs **expertForm** if it is available. If it is not available, it opens the Install As You Go dialog box:

```
method run(var eventInfo Event)
  runExpert( "Document", "Form")
endmethod
```

searchRegistry procedure

System

Searches the registry for a specified value.

Syntax

```
searchRegistry ( const key String, const searchStr String, const rootKey LongInt,
const searchMode LongInt, const inMem TCursor ) Logical
```

Description

searchRegistry searches the registry string data types for the value in *searchStr*. Searches performed by **searchRegistry** are case insensitive and the results are placed in *inMem*, an in-memory TCursor. **searchRegistry** returns True if successful; otherwise, it returns False.

key is entered as a path similar to a file path. If **key** is not blank, the search begins at the specified path; otherwise, it starts from the **rootKey**. **searchStr** is the value of the object you want to locate. **searchRegistry** only searches strings, and not registry DWORD or Binary types. If **searchStr** is blank, **searchRegistry** returns an error. Set **rootKey** with the predefined RegistryKeyType Constants, or it can be set to zero. If **rootKey** is zero, then all rootKeys are searched.

searchMode specifies the registry objects you want to search in the registry. Registry objects include keys, value names, and data. The following table describes the *searchMode* flags:

searchMode	Registry objects searched
0	All
1	Keys
2	Value names
3	Data
4	Keys and value names
5	Keys and Data
6	Value names and Data

The *inMem* TCursor has three fields that are limited to A255. The values in these fields are truncated if the key returned is greater than 255 characters. **searchRegistry** returns a warning if the field limit is reached. The following table displays the structure of *inMem*:

Field	Type	Description
RegistryType	A12	Registry object type (Key, ValueName or Data)
RootKeyConst	A25	Predefined rootKey constant as a string
KeyPath	A255	Full path of the key
ValueName	A255	Full value name
Data	A255	Full Data

Example

The following example searches the registry for all keys containing the string Corel. The results are displayed in a TableView window.

```
; Search the registry for keys that have the string "Corel" in them, and write
; the results to a table
var
    tc TCursor
endVar
searchRegistry( "", "Corel", 0, 1, tc ) ; Search the registry
                                        ; for keys that have
                                        ; "Corel" in them

if NOT tc.isEmpty() then
    tc.instantiateView("keytab.db") ; write the results to a table
endif
tc.close()
```

The following example searches the entire registry for keys containing the word Pdoxwin. The results are displayed in a TableView window.

```
var
    tc Tcursor
    tv TableView
endvar

searchRegistry( "", "Pdoxwin", 0, 1, tc )
tc.instantiateView( ":priv:keysreg" )
tv.open( ":priv:keysreg" )
tv.wait()
tv.close()
```

sendKeys procedure

System

Sends one or more keystrokes to the active window.

Syntax

sendKeys (const *keyText* String [, const *wait* Logical]) Logical

Description

sendKeys sends one or more keystrokes to the active window as if they had been entered at the keyboard. The active window does not have to be Paradox. The argument *keyText* specifies the keystrokes to send. *wait* (optional) specifies whether to continue executing keystroke sequences in the message loop without waiting. **sendKeys** returns False if an error it from sending the keys. *errorCode* returns one of the following messages:

sendKeys procedure

Error code	Error message
peskMissingCloseBrace	Missing closing brace
peskInvalidKey	The key name is not correct
peskMissingCloseParen	Missing closing parentheses
peskInvalidCount	The repeat count is not correct
peskStringTooLong	The keys string is too long
peskCantInstallHook	Could not install Windows journal hook

Notes

- **Sleep()** should always be called after any intensive operation such as opening a form or another application to ensure sequential processing.
- **sendKeys** can only send keystrokes to Microsoft Windows applications. It cannot send the Print Screen (PRINT SCRNR) key to any application.

The keyText argument

Each key is represented by one or more lowercase characters. To represent the letter A, use "a" for keyText. To represent more than one character, string them together. To send the letters a, b, and c, use "abc" for keyText. The plus sign (+), caret (^), percent sign (%), tilde (~), and parentheses () have special meanings to sendKeys. To specify one of these characters, enclose it inside braces. To specify the plus sign, use {+}. To send brace characters, enclose each brace in braces: {{} and {}}.

To specify non-printing characters (such as ENTER or TAB) and keys that represent actions rather than characters, use the following codes:

Key	Codes
BACKSPACE	{backspace}, {bs}, {bksp}, {vk_back}
BREAK	{break}, {vk_break}
CAPS LOCK	{capslock}, {vk_captial}
CLEAR	{clear}, {vk_clear}
DEL	{delete}, {del}, {vk_delete}
Down Arrow	{down}, {vk_down}
END	{end}, {vk_end}
ENTER	{enter}, {return}, {vk_return} (the character ~)
ESC	{escape}, {esc}, {vk_escape}
HELP	{help}, {vk_help}
HOME	{home}, {vk_home}
INS	{insert}, {vk_insert}
Left Arrow	{left}, {vk_left}

NUM LOCK	{numlock}, {vk_numlock}
PAGE DOWN	{pgdn}, {vk_next}
PAGE UP	{pgup}, {vk_prior}
PRINT SCRN	{prtsc}, {vk_snapshot}
Right Arrow	{right}, {vk_right}
SCROLL LOCK	{scrolllock}, {vk_scroll}
SPACEBAR	{vk_space}
TAB	{tab}, {vk_tab}
Up Arrow	{up}, {vk_up}
F1	{F1}, {vk_F1}
F2	{F2}, {vk_F2}
F3	{F3}, {vk_F3}
F4	{F4}, {vk_F4}
F5	{F5}, {vk_F5}
F6	{F6}, {vk_F6}
F7	{F7}, {vk_F7}
F8	{F8}, {vk_F8}
F9	{F9}, {vk_F9}
F10	{F10}, {vk_F10}
F11	{F11}, {vk_F11}
F12	{F12}, {vk_F12}
F13	{F13}, {vk_F13}
F14	{F14}, {vk_F14}
F15	{F15}, {vk_F15}
F16	{F16}, {vk_F16}

The ~ character represents the ENTER key. For example, `sendKeys("abc~")` types the letters abc and the carriage return.

To specify keys combined with SHIFT, CTRL, and ALT, precede the regular key code with one or more of the following codes:

Key	Code
SHIFT	+

sendKeys procedure

CTRL	^
ALT	%

For example, use the following syntax to display the File menu list in Paradox: `sendKeys("%f")`.

The following code moves down 3 menu items: `sendKeys("{down 3}")`.

Pick the item using the following syntax: `sendKeys("~")`.

To combine these three steps into one: `sendKeys("%f{down 3}~")`

To specify that SHIFT, CTRL, and (or) ALT must be held down while one or more keys are pressed, enclose the key codes in parentheses. For example, if SHIFT is pressed while a and b are pressed, use "+(ab)". If SHIFT is pressed while a is pressed, and b is pressed without SHIFT, use "+ab".

To specify repeating keys, enclose a string and a number in braces {key number}. For example, {left 42} specifies you must press the left arrow key 42 times; and {h 9} means you must press h 9 times.

Special commands

The following are special commands you can include as part of the *keyText* argument:

{delay value}

delay sets the delay (in milliseconds) between keystrokes. {delay 1000} waits 1 second between keystrokes; this is approximate and may vary if SHIFT, Alt, or CTRL are set. If the actual time to execute the command is longer, you may see additional delays.

delay is mainly used to let dialog boxes display. Without it the keys are sent at full speed, and Windows processes the keys too quickly to paint the dialog box on the screen. **delay** remains in effect until another delay or a `sendKeys` statement executes; it does not affect action commands.

{action integervalue}

action sends an action to the object in the form that issued the **sendKeys** statement. It allows you to gain control while **sendKeys** executes, to inspect the state of forms or dialog boxes. *integervalue* is a value between 0 and 2047. Do not call any methods or procedures that wait for user input, and do not open a form or report.

{cmt comment}

cmt lets you insert comments. *comment* represents your remarks; all characters are allowed.

{beginexact} text {endexact}

sendKeys normally ignores carriage returns and line feeds, and assigns meanings to certain characters. To bypass this processing, enclose the text with {beginexact} and {endexact}. Once a {beginexact} is encountered, all text is processed exactly as is until the {endexact}.

If you call **sendKeys** while another **sendKeys** statement is executing, Paradox adds the new key sequence to the end of the event queue.

{menu integervalue}

This sends a menu command to the active object. *integervalue* represents a value from the menu command constants.

The wait argument

wait specifies whether to wait after keys are sent, or to continue ObjectPAL execution. The recommended setting is False. Windows sometimes stops responding to **sendKeys** if the **wait** parameter is True (e.g., when keys are sent to nested dialogs). Set **wait** to False when changing the working directory or the private directory.

Note

- **sendKeys** statements are not portable across language barriers.

Example

The following example uses the execute system procedure to run the Windows Notepad application and then **sendKeys** sends keystrokes to Notepad twice and saves the file as TWOLINES.TXT:

```
method pushButton(var eventInfo event)

    execute("notepad.exe")          ; run Notepad.
    sleep(1000)
    ; write a short note.

    sendKeys("this is the first line of a 2-line note.~")
    sendKeys("this is the second line of a 2-line note.")

    ; send alt+f, s to choose File, Save.

    sendKeys("%fs")

    ; send a filename to the dialog box, and
    ; send enter to save the file.

    sendKeys("twolines.txt~")

    ; send Alt+f4 to close Notepad.

    sendKeys("%{f4}")

endMethod
```

sendKeysActionID method**System**

Allows the **sendKeys** procedure to notify you when the **sendKeys** queue is empty.

Syntax

```
sendKeysActionID ( const id SmallInt )
```

Description

sendKeysActionID allows the **sendKeys** procedure to notify you when the **sendKeys** queue is empty. The argument *id* is a user-defined action constant whose value is between the **IdRanges** constants **UserAction** and **UserActionMax**. *id* is sent to the form's active object (or to the form itself if there is no active object) that issued the **sendKeys** method.

The code used to process **sendKeysActionID** is usually placed at the form level. If there is an active object, it receives the ID in its **action** method. The default, however, is to bubble the action ID to the form.

Example

The following example specifies the action ID value sent when the queue is empty. Suppose a form contains an unbound field and a button. The following code is attached to the form's Const window:

```
const
    kMyCustomAction = 1
endConst
```

The following code is attached to the form's built-in action method.

setDefaultPrinterStyleSheet procedure

```
method action(var eventInfo ActionEvent)
  if eventInfo.id() = UserAction + kMyCustomAction then
    message("sendKeys has finished sending")
  endIf
endMethod
```

The following code is attached to a button's built-in pushButton method.

```
method pushButton(var eventInfo Event)
  ; Send keys but do not wait.
  sendKeys("This is some text", FALSE)

  ; Set the action id to send when the queue is empty.
  sendKeysActionID(UserAction + kMyCustomAction)
endMethod
```

setDefaultPrinterStyleSheet procedure

System

Specifies a default printer style sheet.

Syntax

```
setDefaultPrinterStyleSheet ( const fileName String )
```

Description

setDefaultPrinterStyleSheet sets the Paradox style sheet, specified by *fileName*, as the default for documents designed for the printer. If *fileName* does not specify a full path, **setDefaultPrinterStyleSheet** searches the working directory.

Any UIObjects created in forms and reports while the style sheet is active are given the properties and methods of the corresponding prototype objects in the style sheet.

This procedure does not change the properties or methods of existing UIObjects and has no effect on UIObjects in forms and reports that use different style sheets.

Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **setDefaultScreenStyleSheet** to specify the name of the default screen style sheet. The screen style sheet is used whenever you create design documents that are designed for the screen.

Note

- Printer style sheet files have an .FP extension and screen style sheet files have and .FT the extension. Printer and screen style sheets are not interchangeable.

Example

The following example calls **getDefaultPrinterStyleSheet** to determine the current default style sheet. If the style sheet is not COREL.FT, the code calls **setDefaultPrinterStyleSheet** to set it. The code then calls **getDefaultPrinterStyleSheet** again to make sure it was reset successfully.

setDefaultPrinterStyleSheet requires double backslashes in the path, but **getDefaultPrinterStyleSheet** returns single backslashes.

```
method pushButton(var eventInfo Event)
  ; Get and set the system style sheet.
  if getDefaultPrinterStyleSheet() "c:\\Corel\\Paradox\\Corel.fp" then
    setDefaultPrinterStyleSheet("c:\\Corel\\Paradox\\Corel.fp")
  if getDefaultPrinterStyleSheet() "c:\\Corel\\Paradox\\Corel.fp" then
    msgStop("Problem", "Could not set the style sheet.")
  endIf
endIf
endMethod
```


setDefaultScreenStyleSheet procedure**System**

Specifies a default screen style sheet.

Syntax

```
setDefaultScreenStyleSheet ( const fileName String )
```

Description

setDefaultScreenStyleSheet sets the Paradox style sheet specified by *fileName* as the default for documents designed for the screen. If *fileName* does not specify a full path, **setDefaultScreenStyleSheet** searches the working directory.

Any UIObjects created in forms and reports while the style sheet is active are given the properties and methods of the corresponding prototype objects in the style sheet.

This procedure does not change the properties or methods of existing UIObjects and has no effect on UIObjects in forms and reports that use different style sheets.

Use **getStyleSheet** and **setStyleSheet** to work with style sheets for specific forms and reports.

Use **setDefaultScreenStyleSheet** to specify the name of the default screen style sheet. The screen style sheet is used whenever you create design documents that are designed for the screen.

Note

- Printer style sheet files have an .FP extension and screen style sheet files have and .FT the extension. Printer and screen style sheets are not interchangeable.

Example

The following example calls **getDefaultScreenStyleSheet** to determine the current system style sheet. If it is not COREL.FT, **setDefaultScreenStyleSheet** sets it. The code then makes sure it was set successfully.

```
method pushButton(var eventInfo Event)
; Get and set the system style sheet.
if getDefaultScreenStyleSheet() "c:\\Corel\\Paradox\\Corel.ft" then
setDefaultScreenStyleSheet("c:\\Corel\\Paradox\\Corel.ft")
if getDefaultScreenStyleSheet() "c:\\Corel\\Paradox\\Corel.ft" then
msgStop("Problem", "Could not set the style sheet.")
endif
endif
endMethod
```

setDesktopPreference procedure**System**

Sets a desktop preference.

Syntax

```
setDesktopPreference ( const section AnyType, const name AnyType, const value AnyType
) Logical
```

Description

setDesktopPreference sets the desktop preference specified by the section and name arguments. The value argument corresponds to one of the DesktopPreferenceTypes Constants.

Example

The following example sets the title name preference, retrieves the name, and displays it:

```
method pushButton(var eventInfo Event)
setDesktopPreference( PrefProjectSection, prefTitleName,"Paradox pour Windows" )
```

setMouseScreenPosition procedure

```
x = getDesktopPreference( PrefProjectSection, prefTitleName )  
  
x.view()  
endmethod
```

setMouseScreenPosition procedure

System

Displays the pointer at a specified position.

Syntax

1. setMouseScreenPosition (const *mousePosition* Point)
2. setMouseScreenPosition (const *x* LongInt, const *y* LongInt)

Description

setMouseScreenPosition displays the pointer at the specified position. In Syntax 1, the pointer is displayed at the point specified in **mousePosition**. In Syntax 2 the pointer is displayed at the coordinates specified in twips by *x* and *y*.

Use Point type methods such as *x* and *y* to retrieve more information.

Example

See the **getMouseScreenPosition** example.

setMouseShape procedure

System

Sets the shape of the pointer.

Syntax

```
setMouseShape ( const mouseShapeId LongInt [,const persist Logical] ) LongInt
```

Description

setMouseShape sets the shape of the pointer. The argument *mouseShapeId* specifies the shape of the pointer. ObjectPAL provides MouseShapes constants for this purpose.

If *persist* is true then the pointer will be persistent (will not change shape) to objects that implicitly change the shape of the mouse (e.g., button objects and field objects). *persist* will not affect where the ObjectPAL developer has explicitly changed the shape of the mouse. For example, in a **mouseEnter** method of an object, **setMouseShape** will override mouse persistence. **persist** does not affect ActiveX or Native Windows Controls.

Example

In the following example, a form has two buttons: *btnNonPersistent* and *btnPersistent*. The **pushButton** method of each button uses **setMouseShape** to set the mouse shape of the cursor; the first with persistence set to False, the second with persistence set to True. The second button, *btnPersistent* also contains a **mouseEnter** method which will use **isMousePersistent** to evaluate the persistency of the pointer and return it to its original state.

When the first button is pressed, the pointer changes. However, when the pointer moves off the button, the pointer returns to its original setting. When the second button is pressed, the pointer changes and remains that way until the pointer moves back over the second button. This triggers the **mouseEnter** method of the second button and return the pointer back to its original state.

The following code is attached to the **pushButton** method for *btnNonPersistent*:

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

```

; Set the shape to international symbol for No - non-persistent
setMouseShape(MouseNo,FALSE)
endMethod

```

The following code is attached to the **pushButton** method for *btnPersistent*:

```

; btnPersistent::pushButton
method pushButton(var eventInfo Event)
; Set the shape to international symbol for No - persistent
setMouseShape(MouseNo,TRUE)
endMethod

```

The following code is attached to the **mouseEnter** method for *btnPersistent*:

```

; btnPersistent::mouseEnter
method mouseEnter(var eventInfo MouseEvent)
if isMousePersistent() then
; If its persistent, set it back to the arrow cursor
setMouseShape(MouseArrow,FALSE)
endif
endMethod

```

setMouseShapeFromFile method

System

Specifies the shape of the pointer.

Syntax

```
setMouseShapeFromFile ( const fileName String [,const persist Logical] ) LongInt
```

Description

setMouseShapeFromFile specifies the shape of the pointer based on data contained in *fileName*. *fileName* is a *.CUR or *.ANI file that supports paths and aliases. If *fileName* does not exist, a warning is generated. **setMouseShapeFromFile** returns a LongInt handle to the mouse shape.

If **persist** is True then the pointer is persistent (will not change shape) to objects that implicitly change the shape of the mouse (e.g., button objects and field objects). **persist** does not affect where the ObjectPAL developer has explicitly changed the shape of the mouse. For example, in a **mouseEnter** method of an object, **setMouseShape** overrides mouse persistence. **persist** does not affect ActiveX or Native Windows Controls.

Example

In the following example, a form has two buttons: *btnNonPersistent* and *btnPersistent*. The **pushButton** method of each button uses **setMouseShapeFromFile** to set the mouse shape of the cursor to an animated cursor provided with Windows 95, Windows 98 and Windows NT; the first with persistence set to false, the second with persistence set to true. The second button, *btnPersistent* also contains a **mouseEnter** method which will use *isMousePersistent* to evaluate the persistency of the pointer and return it to its original state.

When the first button is pressed, the pointer changes. However, when the pointer moves off the button, the pointer returns to its original setting. When the second button is pressed, the pointer changes and remains that way until the pointer is moved back over the second button. This triggers the **mouseEnter** method of the second button and returns the pointer back to its original state. Each **pushButton** method will determine which operating system its running under to determine where to find the animated cursor file.

The following code is attached to the **pushButton** method for *btnNonPersistent* (assuming a Windows 95 or Windows NT environment):

setRegistryValue method

```
; btnNonPersistent::pushButton
method pushButton(var eventInfo Event)
var
    sysDyn      DynArray[] AnyType
    mouseHandle LongInt
endVar
sysInfo(sysDyn)
if sysDyn["WindowsPlatform"] = "WIN95" then
    ; if Windows 95
    mouseHandle = setMouseShapeFromFile( windowsDir() +
        "\\Cursors\\Hourglass.ANI", FALSE)
else
    ; if Windows NT
    mouseHandle = setMouseShapeFromFile( windowsSystemDir() +
        "\\Hourglass.ANI", FALSE)
endIf
endMethod
```

The following code is attached to the **pushButton** method for *btnPersistent* (assuming a Windows 95 or Windows NT environment):

```
; btnPersistent::pushButton
method pushButton(var eventInfo Event)
var
    sysDyn      DynArray[] AnyType
    mouseHandle LongInt
endVar
sysInfo(sysDyn)
if sysDyn["WindowsPlatform"] = "WIN95" then
    ; if Windows 95
    mouseHandle = setMouseShapeFromFile( windowsDir() +
        "\\Cursors\\Hourglass.ANI", TRUE)
else
    ; if Windows NT
    mouseHandle = setMouseShapeFromFile( windowsSystemDir() +
        "\\Hourglass.ANI", TRUE)
endIf
endMethod
```

The following code is attached to the **mouseEnter** method for *btnPersistent* (assuming a Windows 95 or Windows NT environment):

```
; btnPersistent::mouseEnter
method mouseEnterpushButton(var eventInfo MouseEvent)
    if isMousePersistent() then
        ; If its persistent, set it back to the arrow cursor
        setMouseShap(MouseArrow,FALSE)
    endIf
endMethod
```

setRegistryValue method

System

Sets a value in the registry.

Syntax

```
setRegistryValue ( const key String, const value String, const data AnyType, const
rootKey LongInt ) Logical
```

Description

setRegistryValue writes data to a specified value of a registry key. If the key or value do not exist, then they will be created. If data is empty then only **key** is created. If value is empty, then key and data are created.

key is a path similar to a file path. However, wildcards are not expanded in the key. key cannot contain a single backslash and cannot be empty. Its size is limited to 65,534 bytes.

The value is a string that is limited to 65,534 bytes. value can contain backslashes and can be empty.

setRegistryValue returns True if successful; otherwise, it returns False.

data accepts the following types:

ObjectPAL Type	Registry type	Size limitation
Currency	String	32k
Date	String	32k
DateTime	String	32k
Logical	String	32k
LongInt	DWORD	4 bytes
Memo	String	32k
Number	String	32k
Point	String	32k
SmallInt	DWORD	4 bytes
String	String	32k
Time	String	32k

rootKey is analogous to a directory drive. Set rootKey with the predefined RegistryKeyType Constants.

Example

The following example sets the current ObjectPAL level in the registry:

```
var
    strLevel    String
endvar

; create key, value and data in regCurrentUser
setRegistryValue( "Software\\Corel\\Myapp\\Settings", "ObjectValue", "An object",
regKeyCurrentUser )
```

setUserLevel procedure**System**

Sets your ObjectPAL level (Beginner or Advanced). Beginner restricts the methods displayed for each object in the Integrated Development Environment (IDE) to those a new ObjectPAL user would likely need; Advanced displays all methods.

sleep procedure

Syntax

```
setUserLevel ( const level String )
```

Description

setUserLevel sets your ObjectPAL level (Beginner or Advanced). Use **getUserLevel** to return the current setting.

Notes

- The ObjectPAL level setting does not affect how code executes; it only affects what is displayed in the user interface.
- The advanced setting is highly recommended.

Example

Use **getUserLevel** to determine if the ObjectPAL user level is set to Beginner. If the ObjectPAL level is set to Beginner, **setUserLevel** sets it to Advanced. If the ObjectPAL user level is already set to Advanced, the code sends a message stating this to the Status Bar.

```
;setToAdvanced::pushButton
method pushButton(var eventInfo Event)
    if getUserLevel() = "Beginner" then
        setUserLevel("Advanced")
        message("ObjectPAL level is now set to Advanced")
    else
        message("ObjectPAL level was already set to Advanced")
    endIf
endmethod
```

sleep procedure

System

The following example produces a delay of a specified duration.

Syntax

```
sleep ( [ const numberOfMilliseconds LongInt ] )
```

Description

sleep disables the executing form for the number of milliseconds specified in **numberOfMilliseconds**. **sleep** does not disable the desktop or stop timer events. When the form is disabled, it cannot receive keystrokes, mouse events or focus.

Notes

- When **sleep** is called with no argument, it does not disable the form. Instead, it causes the current method to yield to Windows to let a single pending message be processed.
- **Sleep()** should always be called after any intensive operation such as opening a form or another application to ensure sequential processing.

Example

The following example displays a message in the status line and 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

```

sound procedure

System

Creates a sound of specified frequency and duration.

Syntax

```
sound ( const freqHertz, const durationMilliSecs LongInt )
```

Description

sound creates a sound of the frequency specified by *freqHertz* (in Hertz) for a time *durationMilliSecs* (in milliseconds) . Frequency values can range from 1 to 50,000 Hertz. The sound is played through the computer's internal speaker, and not the system sound card.

Note

- The **sound** procedure only works as described in Windows NT. In Windows 95 and Windows 98, the default windows sound is played if there is a sound card installed on the machine. If no sound card is installed, then the standard system beep is played and the parameters are disregarded.

Example

The following example uses the **pushButton** method for *makeMusic* to declare constants for frequency values in a scale. These notes 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

```

startWebBrowser procedure

```
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
```

startWebBrowser procedure

System

Launches the default or specified web browser with the specified URL.

Syntax

```
startWebBrowser ( const URL string ) or
startWebBrowser ( const URL string, const alternateBrowser string )
```

Description

URL is the URL of the web page to be loaded (for example, "http://www.corel.com") and alternateBrowser is the path of an alternative web browser executable if you want to use one different than your default (for example, "c:\program files\netscape\communicator\netscape.exe").

Example

The following example uses the startWebBrowser procedure to open the default web browser to the defined URL.

```
method run ( var eventInfo Event)
  StartWebBrowser ("www.corel.com")
endmethod
```

sysInfo procedure

System

Creates a dynamic array of information about the system running Paradox.

Syntax

```
sysInfo ( var info DynArray[ ] AnyType )
```


Description

sysInfo creates a dynamic array of information about the system running Paradox. Declare a dynamic array named `info` before calling `sysInfo`. `info` contains indexes for system attributes and their values. The following table describes the structure of `info`:

System Attribute Index	Definition
<code>AnsiCodePage</code>	The ANSI (Windows) code page loaded by Windows
<code>AreMouseButtonsSwapped</code>	Functions of the left and right mouse buttons are reversed
<code>CodePage</code>	The code page currently loaded by Windows
<code>CPU</code>	Processor type
<code>Edition</code>	Paradox edition (e.g., Standard)
<code>EngineDate</code>	Creation date of database engine
<code>EngineLanguageID</code>	The language used for Borland Database Engine (BDE) messages and QUERY BY EXAMPLE (QBE) keywords, shown in the list of language identifiers
<code>EngineVersion</code>	Version number of database engine
<code>FullHeight</code>	Vertical working area in a maximized window (in pixels)
<code>FullWidth</code>	Horizontal working area in a maximized window (in pixels)
<code>IconHeight</code>	Height of icons (in pixels)
<code>IconWidth</code>	Width of icons (in pixels)
<code>KeyboardFNKeys</code>	Number of function keys
<code>KeyboardLayoutID</code>	The layout name for the currently loaded keyboard (usually a language ID)
<code>KeyboardSubType</code>	An OEM-dependent value
<code>KeyboardType</code>	Keyboard type and manufacturer
<code>LanguageDriver</code>	Default language drivers for Paradox tables
<code>LocalShare</code>	Reports whether Local Share is active
<code>Memory</code>	Available memory in bytes, including swap file (if present)
<code>Mouse</code>	The number of mice attached to the system
<code>NetDir</code>	The path to PDOXUSRS.NET
<code>NetProtocol</code>	Network protocol
<code>NetShare</code>	Reports whether Net Share is active
<code>NetType</code>	Network type
<code>ParadoxSystemDir</code>	The path of the Paradox folder
<code>ScreenHeight</code>	Total height of screen (in pixels)

Language identifiers

ScreenWidth	Total width of screen (in pixels)
StartupDir	The full path (including the drive ID letter) to your start-up folder (the folder from which Paradox was launched)
SystemDefaultLCID	The system default locale ID (a 32-bit value which is the combination of a language ID and a sort ID)
UserDefaultLCID	The user default locale ID
UserName	Network user name
WindowsBuild#	The internal build number
WindowsDir	Path to the WINDOWS directory (folder)
WindowsPlatform	Win95, NT, or WIN32s
WindowsSystemDir	Path to the WINDOWS\SYSTEM directory (folder)
WindowsText	Arbitrary information
WindowsVersion	Windows version number

Example

The following example writes system information to a dynamic array named *userSys* and then displays *userSys* in a View dialog box:

```
; 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
```

Language identifiers

System

Language identifiers consists of the primary language ID and the sub_language ID.
The following codes are included in the primary language IDs:

Code	Language	Code	Language
0x0401	Arabic	0x0415	Polish
0x0402	Bulgarian	0x0416	Brazilian Portuguese
0x0403	Catalan	0x0417	Rhaeto-Romanic
0x0404	Traditional Chinese	0x0418	Romanian
0x0405	Czech	0x0419	Russian
0x0406	Danish	0x041A	Croato-Serbian (Latin)
0x0407	German	0x041B	Slovak

0x0408	Greek	0x041C	Albanian
0x0409	U.S. English	0x041D	Swedish
0x040A	Castilian Spanish	0x041E	Thai
0x040B	Finnish	0x041F	Turkish
0x040C	French	0x0420	Urdu
0x040D	Hebrew	0x0421	Bahasa
0x040E	Hungarian	0x0804	Simplified Chinese
0x040F	Icelandic	0x0807	Swiss German
0x0410	Italian	0x0809	U.K. English
0x0411	Japanese	0x080A	Mexican Spanish
0x0412	Korean	0x080C	Belgian French
0x0413	Dutch	0x0C0C	Canadian French
0x0414	Norwegian - Bokml	0x100C	Swiss French
0x0810	Swiss Italian	0x0816	Portuguese
0x0813	Belgian Dutch	0x081A	Serbo-Croatian(Cyrillic)
0x0814	Norwegian - Nynorsk		

tracerClear procedure

System

Clears the Tracer window.

Syntax

```
tracerClear ( )
```

Description

tracerClear clears the Tracer window. You can open the Tracer window with the **tracerOn** procedure at run time, or by clicking View, Tracer in the ObjectPAL Editor.

Example

The following example clears the Tracer window. Assume that the Tracer window is open and contains information.

```
; wipeTracer::pushButton
method pushButton(var eventInfo Event)
  tracerClear()           ; clear the Tracer window
endMethod
```

tracerHide procedure

System

Hides the Tracer window.

Syntax

```
tracerHide ( )
```

tracerOff procedure

Description

tracerHide hides the Tracer window. This procedure makes the Tracer window invisible but does not clear or close it. To view the Tracer again, use **tracerShow**.

Example

The following example hides the Tracer window, pauses and then displays it again. Assume that the Tracer window is 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
```

tracerOff procedure

System

Turns off code tracing.

Syntax

```
tracerOff ( )
```

Description

tracerOff stops writing code traces to the Tracer window but does not hide the Tracer window. To hide the Tracer window use **tracerHide**. You can resume tracing code with the **tracerOn** procedure. By default, tracing is turned on when the Tracer window is opened.

Example

The following example turns off code tracing:

```
; stopTracer::pushButton
method pushButton(var eventInfo Event)
  tracerOff()           ; turns off code tracing
endMethod
```

tracerOn procedure

System

Activates code tracing.

Syntax

```
tracerOn ( )
```

Description

tracerOn activates code tracing. This procedure resumes writing code traces to the Tracer window.

Example

The following example reactivates code tracing:

```
; startTracer::pushButton
method pushButton(var eventInfo Event)
  tracerOn()           ; reactivates the Tracer window
endMethod
```

tracerSave procedure**System**

Saves the contents of the Tracer window to a file.

Syntax

```
tracerSave ( const fileName String )
```

Description

tracerSave saves the contents of the Tracer window to the file specified by *fileName*.

Example

The following 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
```

tracerShow procedure**System**

Makes the Tracer window visible.

Syntax

```
tracerShow ( )
```

Description

tracerShow makes the Tracer window visible. You can make the Tracer window invisible using the **tracerHide** procedure.

Example

See the **tracerHide** example.

tracerToTop procedure**System**

Positions the Tracer window on top of all other windows on the desktop.

Syntax

```
tracerToTop ( )
```

Description

tracerToTop places the Tracer window on top of all other windows on the desktop.

Example

See the **tracerWrite** example.

tracerWrite procedure**System**

Writes a message to the Tracer window.

Syntax

```
tracerWrite ( const message String [ , const message String ] * )
```

Description

tracerWrite writes a message to the Tracer window.

twipsToPixels procedure

Example

The following example logs a message to the Tracer window and places the Tracer window on top of all other windows on the desktop:

```
; logTracerMsg::pushButton
method pushButton(var eventInfo Event)
tracerOn()
tracerWrite("Tracer hit by " + String(self.Name) +
            " at " + String(time()))           ; log a message
tracerToTop()                                ; make the Tracer window the top-layer window
endMethod
```

Note

- This example assumes that **TracerOn()** has already been called.

twipsToPixels procedure

System

Converts screen coordinates from twips to pixels.

Syntax

```
twipsToPixels ( const twips Point ) Point
```

Description

twipsToPixels converts the screen coordinates specified in twips from twips to pixels.

Example

See the **pixelsToTwips** example.

version procedure

System

Returns the Paradox version and build number.

Syntax

```
version ( ) String
```

Description

version returns the Paradox version and build number. If you have more than one version installed, version returns the version number and build of the active application.

Example

The following example uses the **pushButton** method for *showVersion* to show which version and build of Paradox is active:

```
; showVersion::pushButton
method pushButton(var eventInfo Event)
    msgInfo("FYI", "You are running version and build " + version() + ".")
endMethod
```

winGetMessageID procedure

System

Returns the ID of a Windows message.

Syntax

```
winGetMessageID ( const msgName String ) SmallInt
```

Description

winGetMessageID returns the integer value of the Windows message represented by the string specified in *msgName*. Messages may include WM_CLOSE (sent as a signal that a window or application should terminate), and WM_ACTIVATE (sent when a window is activated or deactivated).

winGetMessageID returns 0 if *msgName* is not recognized as a Windows message. For more information, see your Windows programming documentation.

Note

- **winGetMessageID** should only be used by Windows programmers who are familiar with Windows messages.

Example

The following example displays the integer value of the Windows message WM_LBUTTONDOWN:

```
method pushButton(var eventInfo event)
  var
    smMsgID    SmallInt
    stMsgName   String
  endVar

  stMsgName = "WM_LBUTTONDOWN"
  smMsgID = winGetMessageID(stMsgName)
  smMsgID.view(stMsgName) ; Displays 513 in Win32.
  ; The value may be different in other versions of Windows.
endMethod
```

winPostMessage procedure**System**

Posts a message to Windows.

Syntax

```
winPostMessage ( const hWnd LongInt, const msg LongInt, const wParam LongInt, const
lParam LongInt ) Logical
```

Description

winPostMessage posts a message to Windows. Unlike **winSendMessage**, which dispatches its message immediately, **winPostMessage** method adds its message to the end of the Windows message queue. Messages in the queue are dispatched in the order than they appear. Windows determines which arguments are valid to winPostMessage For more information, see your Windows programming documentation.

Note

- **winPostMessage** should only be used by Windows programmers who are familiar with Windows messages.

Example

See the **winSendMessage** example.

winSendMessage procedure**System**

Sends a message to Windows.

Syntax

```
winSendMessage ( const hWnd LongInt, const msg LongInt, const wParam LongInt, const
lParam LongInt ) LongInt
```

writeEnvironmentString procedure

Description

winSendMessage sends a message to Windows. Windows determines which arguments are valid to **winSendMessage**. For more information, see your Windows programming documentation.

Note

- **winPostMessage** should only be used by Windows programmers who are familiar with Windows messages.

Example

The following example opens Notepad and calls **enumWindowNames** to create a table of data about the windows currently open on your system. The code then searches the table for information about Notepad, and gets the handle for that window. Next, calls **winGetMessageID** to retrieve the integer value of the command represented by the string "WM_CLOSE." Finally, the code calls **winSendMessage** with the window handle and command value as arguments. The message is dispatched to Windows, and Notepad is closed. To add the message to the end of the Windows message queue, call **winPostMessage** instead of **winSendMessage**.

```
method pushButton(var eventInfo Event)
var
    tcOpenWin    TCursor
    tbOpenWin    Table
    stTbName     String
    siWinHandle,
    siWinMsgID   SmallInt
endVar

stTbName = ":PRIV:openWin"

execute("Notepad.exe", No, ExeShowNormal)      ; Run Notepad.
sleep(1000)                                     ; Pause so you can see what happens.

enumWindowNames(stTbName)                      ; List open windows.

tcOpenWin.open(stTbName)
; Locate the Notepad window in the list of names.
if tcOpenWin.locatePattern("ClassName", "Notepad") then
    ; Get the Windows handle for the Notepad window.
    siWinHandle = tcOpenWin."Handle"
    ; Get the Windows message ID for WM_CLOSE to close the window.
    siWinMsgID = winGetMessageID("WM_CLOSE")
    ; Send the specified message to the specified window.
    winSendMessage(siWinHandle, siWinMsgID, 0, 0)
else
    errorShow()
endif
endmethod
```

writeEnvironmentString procedure

System

Sets a variable in the Paradox copy of the DOS environment.

Syntax

```
writeEnvironmentString ( const key String, const value String ) Logical
```


Description

writeEnvironmentString sets a variable in the Paradox copy of the DOS environment. When Paradox launches, a copy of the DOS environment is made. **writeEnvironmentString** writes to that copy but changes are not written to the DOS environment.

You can use the SET command to assign environment variables. These assigned values control the appearance and function of DOS and some batch files. Some common environment variables include PATH, PROMPT, and COMSPEC. For more information, the SET command in your DOS manuals.

Example

See the **readEnvironmentString** example.

writeProfileString procedure**System**

Writes system information to a file.

Syntax

```
writeProfileString ( const fileName String, const section String, const key String,
const value String ) Logical
```

Description

writeProfileString writes system information to a specified file on your system. If you specify a filename without a path, this method searches for the file in the WINDOWS directory (folder).

Typically, you use this method to modify your WIN.INI file. In this case, *fileName* would be WIN.INI. Sections are defined by square brackets and reside on a separate line in the WIN.INI file. To specify a section, simply type the string or section name (e.g., to specify the [windows] section, type "windows").

In each section, a value marker is followed by an equal sign (e.g., Beep =). The equal sign is not required when you specify the value of key.

Example

See the **readProfileString** example.

Table type

A Table variable describes a table. It differs from a TCursor which is a pointer to a table's data, and from a table frame or a TableView, which are objects that display the data.

You can use Table variables to add, copy, create, and index tables, to perform column calculations in columns, retrieve information about a table's structure, and more. Some table operations require Paradox to create temporary tables in the private directory.

The **create**, **index**, and **sort** structures are basic language elements (not methods or procedures) that operate on Table variables. Table variables cannot be used to edit records — you must use a TCursor or table frame (UIObject) to modify a record in a table.

Methods for the Table type

Table

add	enumFieldNames	nKeyFields
attach	enumFieldNamesInIndex	nRecords
cAverage	enumFieldStruct	protect
cCount	enumIndexStruct	reIndex
cMax	enumRefIntStruct	reIndexAll
cMin	enumSecStruct	rename
cNpv	familyRights	setExclusive
compact	fieldName	setGenFilter
copy	fieldNo	setIndex
create	fieldType	setRange
createIndex	getGenFilter	setReadOnly
cSamStd	getRange	showDeleted
cSamVar	index	sort
cStd	isAssigned	subtract
cSum	isEmpty	tableRights
cVar	isEncrypted	type
delete	isShared	unAttach
dropGenFilter	isTable	unlock
dropIndex	lock	unProtect
empty	nFields	usesIndexes

add method/procedure

Table

Adds data from 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

add adds data from a table to a target table, which can be specified using a String (*destTableName* in Syntax 1) or a Table variable (*destTableVar* in Syntax 2). If the target table does not exist, this method creates it. The source table and the target table can be any types that have compatible field structures.

When set to True, *append* adds records at the end of a non-indexed target table, or at the appropriate place in an indexed target table. When set to True, *update* compares records in both tables, and where key values match, replaces the data in the target table. When both are set to 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*. By default, both arguments are True.

```
myTable.add(yourTable, False, True) ; specifies update
myTable.add(yourTable)               ; specifies update and append by default
```

Key violations (including validity check violations) are listed in KEYVIOL.DB in the private directory. If KEYVIOL.DB already exists, **add** overwrites it. If KEYVIOL.DB does not exist, this method creates it.

When tables are keyed, **add** uses the keyed fields to determine which records to update and which to append. If the target table is not keyed and update is set to True, **add** fails. If the target table is not keyed, the structure of the entire record in the source table must match the record structure in the target table.

DOS

If you are a DOS PAL programmer, you can use the following procedure to operate on tables by specifying the table name, rather than using a variable.

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
```

Example

The following example uses the **pushButton** method for *updateCust* to run a query from an existing file and add 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"

newCust.readFromFile("newCust.qbe")

if newCust.executeQBE() then           ; if the query succeeds
    ansTbl.attach(":PRIV:Answer.db")

    ; attempt to add Answer.db records to Customer.db
    if isTable(destTbl) then
        if NOT ansTbl.add(destTbl) then
            errorShow()
        endif
```

attach method

```
    else
        msgStop("Error", "Can't find " + destTbl + ".")
    endIf
else
    errorShow("Query failed.")
endIf

endMethod
```

attach method

Table

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

attach associates a Table variable with the table specified in *tableName*. Optional arguments *tableType* and *db* specify a table type (Paradox or dBASE) and a database. If you don't specify *tableType*, ObjectPAL determines the table type from the table name's file extension. If you don't specify *db*, ObjectPAL works in the default database.

This method fails if the value of *tableName* is not valid (e.g., the table name doesn't match the table type, or conflicts with the database name). This method returns True if successful; otherwise, it returns False.

Notes

- **attach** does not verify that *tableName* exists, or is a table. Use the **isTable** method to verify a table's 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.

Example

In the following 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.

```
; 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

```

cAverage method/procedure

Table

Returns the average of values in a column of fields.

Syntax

1. cAverage (const *fieldName* String) Number
2. cAverage (const *fieldNum* SmallInt) Number

Description

cAverage returns the average of values in the column of fields specified by *fieldName* or *fieldNum*. If the column contains empty fields, **cAverage** uses the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period **cAverage** attempts to place a write lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cAverage (const *tableName* String, const *fieldName* String) Number
2. cAverage (const *tableName* String, const *fieldNum* SmallInt) Number

Example

The following 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

```

cCount method/procedure

Table

Returns the number of nonblank values in a table column.

Syntax

1. cCount (const *fieldName* String) LongInt
2. cCount (const *fieldNum* SmallInt) LongInt

Description

cCount returns the number of values in the column specified by *fieldName* or *fieldNum*. **cCount** works for all field types. If the column contains numeric values **cCount** 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 nonblank values in the column of fields.

This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period **cCount** attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmers, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cCount (const *tableName* String, const *fieldName* String) Number
2. cCount (const *tableName* String, const *fieldNum* SmallInt) Number

Example

In the following example, the **pushButton** method for *lineItemInfo* uses **cAverage** and **cCount** to perform calculations on the Qty field in LINEITEM.DB. The code attempts to place a write lock on the table so that changes cannot be made to the table between the calls to **cAverage** and **cCount**. If the lock cannot be placed, the operation is aborted.

```
; lineItemInfo::pushButton
method pushButton(var eventInfo Event)
var
  lineTbl Table
  avgQty Number
  numItems LongInt
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",
            String(avgQty, "\nbased on ", numItems, " items. "))
  else
    errorShow("Can't lock Lineitem table.")
  endIf
else
  errorShow("Can't attach to Lineitem table.")
endIf

endMethod
```

cMax method/procedure**Table**

Returns the maximum value of a table's column.

Syntax

1. cMax (const *fieldName* String) Number
2. cMax (const *fieldNum* SmallInt) Number

Description

cMax returns the maximum value in the column of fields specified by *fieldName* or *fieldNum*. **cMax** respects the limits of restricted views set by **setRange** or **setGenFilter**. **cMax** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a write lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. **cMax** (const *tableName* String, const *fieldName* String) Number
2. **cMax** (const *tableName* String, const *fieldNum* SmallInt) Number

Example

The following example displays the maximum value 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
```

cMin method/procedure**Table**

Returns the minimum value of a table's column.

Syntax

1. **cMin** (const *fieldName* String) Number
2. **cMin** (const *fieldNum* SmallInt) Number

Description

cMin returns the minimum value in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cMin** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. **cMin** (const *tableName* String, const *fieldName* String) Number
2. **cMin** (const *tableName* String, const *fieldNum* SmallInt) Number

cNpv method/procedure

Example

The following example displays the minimum value 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
```

cNpv method/procedure

Table

Returns the net present value of a column, based on a discount or interest rate.

Syntax

1. cNpv (const *fieldName* String, const *discRate* AnyType) Number
2. cNpv (const *fieldNum* SmallInt, const *discRate* AnyType) Number

Description

cNpv returns the net present value of the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cNpv** handles blank values as specified in the **blankAsZero** setting for the session.

The net present value calculation is based on *discRate*, expressed as a decimal (e.g., 0.12 for 12 percent). **cNpv** calculates net present values using the following formula:

$$\text{cNpv} = \sum_{p=1 \text{ to } n} \text{Vp} / (1 + r)^p$$

(n = number of periods, Vp = cash flow in pth period, and r = rate per period)

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

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

Example

The following example defines a Table variable for the *GoodFund* table and calculates the net present value for the Expected Return field. 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

```

compact method

Table

Removes deleted records from a table.

Syntax

```
compact ( [ const regIndex Logical ] ) Logical
```

Description

compact removes deleted records from a table.

Deleted records are not immediately removed from a dBASE table. Instead, they are flagged as deleted and kept in the table. The optional argument *regIndex* specifies whether to regenerate or update the indexes associated with the table. When *regIndex* is set to True, this method regenerates all indexes associated with the table. This includes indexes specified by **usesIndexes**, and the .MDX index (whose name matches the table name). If *regIndex* is set to False, indexes are not regenerated. By default, *regIndex* is set to True.

If you delete records from a Paradox table, they cannot be retrieved. However, the table file and associated index files contain dead space where the record was originally stored. If you use **compact** with a Paradox table, all indexes are regenerated and dead space is removed.

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 set to True. In this example, *regIndex* is set to False and **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()

```

copy method/procedure

```
        ; removes deleted records and regenerates all indexes  
endMethod
```

copy method/procedure

Table

Copies a table.

Syntax

1. copy (const *destTable* String) Logical
2. copy (const *destTable* Table) Logical

Description

copy copies the records from a source table to a target table specified in *destTable*. The data from the source table completely replaces the data in target table. The source and target tables can be different table types. If the target table is open, the method fails.

Throughout the retry period, this method attempts to place a write lock on the source table, and a full (exclusive) lock on the target table. If either lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. copy (const *sourceTable* String, const *destTable* String) Logical
2. copy (const *sourceTable* String, const *destTable* Table) Logical

Example

In the following example, the **pushButton** method for *backupCust* copies the *Customer* table to *CustBak*. If *CustBak* already exists in the current directory, this code asks 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 CustBak.db  
; Does not copy .VAL file if all it contains is RI information.  
endMethod
```

create keyword

Table

Creates a table.

Syntax

```
create tableName [ as tableType ] [ database db ]
  [ [ like likeObject ]
    [ with fieldName : type [ , fieldName : type ] * ]
    [ where fieldID is newname [ , fieldID is newname ] * ]
    [ without fieldID [ , fieldID ] * ]
    [ struct fieldStructTable ]
    [ indexStruct indexStructTable ]
    [ refIntStruct refIntStructTable ]
    [ secStruct secStructTable ]
    [ languageDriver driverName ]
    [ versionLevel versionNumber ]
  ] *
  [ key fieldID [ , fieldID ] * ]
endCreate
```

Description

create creates a table specified by *tableName*. Unless an **as** clause explicitly specifies a table type (see below), **create** uses the *tableName* extension to infer a table type (.DB is a Paradox table and .DBF is a dBASE table.) For example, given Orders.dbf for *tableName*, **create** creates a dBASE table. If *tableName* does not include an extension, **create** creates a Paradox table.

If *tableName* exists, **create** attempts to place a full lock on it throughout the retry period. If the lock cannot be placed, **create** fails.

The following clauses specify table attributes. They are optional, and can appear in any order within the **create** structure. The clauses are executed in the order they appear in the structure.

The **as** *tableType* clause specifies the table format:

```
AS "Paradox"
```

If **as** is omitted, **create** creates a Paradox table by default (unless the table resides on a SQL server. See the discussion of the **database** clause, below).

The **database** *db* clause specifies a Database variable (opened before creating the new table) that determines where the table resides. If the database is on an SQL server, the table is of a type appropriate for the server. By default, the table is created in the working directory:

```
DATABASE megaData
```

The **like** *likeObject* clause specifies an open TCursor, table name, or Table variable from which you can borrow field names, field types, the language driver, and the version level. 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):

```
LIKE "Sales.dbf"           ; table name as a string
LIKE ordersTC              ; a TCursor variable pointing to ORDERS.DB
LIKE ordersTB              ; a Table variable pointing to ORDERS.DB
```

The **with** *fieldName* : *type* clause adds one or more fields to the table structure:

```
with "Last name" : "A20", "First name" : "A15", "Quantity" : "N"
```

You can specify the field type for *fieldName* in *type*. Valid values for *type* vary depending on the type of table you are creating. Paradox tables use specific field names. Tables created on servers other than Paradox require field name translations.

The following tables list valid field specifications for Paradox and dBASE tables:

create keyword

Paradox tables	3.5 and earlier	4.5	5.0	7
Alpha	Annn	Annn	Annn	Annn
Number	N	N	N	N
Money	\$	\$	\$	\$
Date	D	D	D	D
Short	S	S	S	S
Memo	(none)	Mnnn	Mnnn	Mnnn
Formatted Memo	(none)	(none)	Fnnn	Fnnn
Binary	(none)	Bnnn	Bnnn	Bnnn
Graphic	(none)	(none)	Gnnn	Gnnn
OLE	(none)	(none)	Onnn	Onnn
Logical	(none)	(none)	L	L
Long Integer	(none)	(none)	I	I
Time	(none)	(none)	T	T
Timestamp	(none)	(none)	@	@
BCD	(none)	(none)	#	#
Autoincrement	(none)	(none)	+	+
Bytes	(none)	(none)	Y	Y

dBASE tables	III+	IV	V
Character	Cnnn	Cnnn	Cnnn
Number	Nnnn	Nnnn	Nnnn
Date	D	D	D
Logical	L	L	L
Memo	M	M	M
Float	(none)	Fnnn.d	Fnnn.d
OLE	(none)	(none)	O
Binary	(none)	(none)	B

The **where** *fieldID* is "*newName*" clause changes the name of one or more fields specified by the name or number *fieldID* to *newName*:

where "Last name" IS "Customer last name", 2 IS "Customer first name"

The **without** *fieldID* clause removes one or more fields (specified by name or number) from the structure. Example:

```
without 4, "Country code"
```

The **struct** clause specifies in *fieldStructTable* an open TCursor, table name, or Table variable from which you can borrow the field-level structure. Unlike the **like** clause, **struct** borrows all validity check and primary key information. Use **enumFieldStruct** to generate *fieldStructTable* (or create it manually) before executing **create**:

```
struct "CustFlds.db"
```

The **indexStruct** clause specifies in *indexStructTable* an open TCursor, table name, or Table variable from which you can borrow secondary index information. Use **enumIndexStruct** to generate *indexStructTable* (or create it manually) before executing **create**:

```
indexStruct "CustIndx.db"
```

The **refIntStruct** clause specifies an open TCursor, table name, or Table variable from which you can borrow referential integrity information. Use **enumRefIntStruct** to generate *refIntStructTable* (or create it manually) before executing **create**:

```
refIntStruct "Cust_Ref.db"
```

The **secStruct** clause specifies in *secStructTable* an open TCursor, table name, or Table variable from which you can borrow security information. Use **enumSecStruct** to generate *secStructTable* (or create your own) before executing **create**:

```
secStruct "Cust_Sec.db"
```

When you use **secStruct**, Paradox automatically protects the table with the master password *secret*. For information about master passwords, see About password security in the Paradox online Help.

The **languageDriver** clause specifies in *driverName* the internal name of a language driver to use with the table. A language driver determines the table's sort order and available character set. For a list of language drivers, see language drivers for Paradox tables, or language drivers for dBASE tables.

The **versionLevel** clause specifies in *versionNumber* what level of table to create. Valid values for *versionNumber* are listed in the following table.

Table type	Version number
Paradox	3 specifies a level 3 table corresponding to that created for Paradox 3.5 and earlier (Paradox Engine version 2)
	4 specifies a level 4 table corresponding to Paradox for Windows 4.5 and earlier and Paradox for DOS 4.0 and 4.5 (Paradox Engine version 3)
	5 specifies a level 5 table corresponding to Paradox for Windows 5.0
	7 specifies a level 7 table corresponding to Paradox 7
dBASE	3 specifies a dBASE III table
	4 specifies a dBASE IV table
	5 specifies a dBASE for Windows table

The **key** *fieldID* clause specifies one or more key fields. You must specify key fields in order from left to right:

```
key "Last name", "First name"
```

create keyword

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 are meaningless unless preceded by a **like** clause.

Note

- Because **create** is not a method, dot notation is inappropriate. Instead, use = to assign the **create** structure to a Table variable.
- If the versionLevel parameter is not set the created table will default to version 4.

Example 1

The following example creates a Paradox table named PARTS.DB. The table has three fields (Part number, Part name, and Quantity) and 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
```

Example 2

The following examples show two ways to create a dBASE table named 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 following example uses the **struct** option to borrow field-level information (including primary keys and validity checks) for use in a new table. For more information, see **enumFieldStruct**.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
    custTbl, newCustTbl Table
    custTC TCursor
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

    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
```

Language drivers for Paradox tables

The following table displays the language drivers that you can use for Paradox tables, and the code page for each driver. Use the internal name to specify *driverName*.

Note

- Internal language driver names are case-sensitive.

Driver name	Internal	Language/DOS Code Page
Paradox 'ascii'	ASCII	English (US)/437
Paradox 'hebrew'	HEBREW	Hebrew
Paradox 'intl'	INTL	International/437
Paradox 'intl850'	INTL850	International/850

create keyword

Paradox 'nordan'	NORDAN	Danish-Norwegian
Paradox 'turk'	TURK	Turkish
Paradox ANSI 'turk'	ANTURK	Turkish
Paradox ANSI China	ANCHINA	Chinese
Paradox ANSI Cyrillic	ANCYRR	Russian
Paradox ANSI Czech	ANCZECH	Czech
Paradox ANSI Greek	ANGREEKI	Greek
Paradox ANSI HEBREW	ANHEBREW	ANSI Hebrew
Paradox ANSI Hun DC	ANHUNDC	Hungarian
Paradox ANSI Intl	ANSIINTL	ANSI International
Paradox ANSI Intl850	ANSI1850	ANSI International/850
Paradox ANSI Korea	ANKOREA	Korean
Paradox ANSI Nordan4	ANSINOR4	ANSI Danish-Norwegian/4
Paradox ANSI Polish	ANPOLISH	Polish
Paradox ANSI Slovene	ANSISLOV	Yugoslavia
Paradox ANSI Spanish	ANSISPAN	ANSI Spanish
Paradox ANSI Swedfin	ANSISWFN	ANSI Swedish-Finnish
Paradox ANSI Thai	ANTHAI	ANSI Thai
Paradox China 437	CHINA	Chinese/437
Paradox Cyrr 866	CYRR	Russian/866
Paradox Czech 852	CZECH	Czech/852
Paradox Czech 867	CSKAMEN	Czech/867
Paradox ESP 437	SPANISH	Spanish/437
Paradox Greek GR437	GRCP437	Greek/437
Paradox Hun 852 DC	HUN852DC	Hungarian/852
Paradox ISL 861	ICELAND	Iceland/861
Paradox Korea 949	KOREA	Korean/949
Paradox NORDAN	NORDAN	Danish-Norwegian/865
Paradox NORDAN40	NORDAN40	Danish-Norwegian/865
Paradox Polish 852	POLISH	Polish/852

Paradox Slovene 852	SLOVENE	Yugoslavia/852
Paradox SWEDFIN	SWEDFIN	Swedish-Finnish/437
Paradox Thai 437	THAI	Thai/437

Language drivers for dBASE tables

The following table displays the language drivers that you can use for dBASE tables. Use the internal name to specify *driverName*.

Note

- Internal language driver names are case-sensitive.

Driver	Internal name	Language
dBASE CHN pc437	DB437CN0	Chinese
dBASE CSY cp852	DB852CZ0	Czech
dBASE CSY cp867	DB867CZ0	Czech
dBASE DAN cp865	DB865DA0	Danish
dBASE DEU cp437	DB437DE0	German
dBASE DEU cp850	DB850DE0	German
dBASE ELL GR437	DB437GR0	Greek
dBASE ENG cp437	DB437UK0	English (U.K)
dBASE ENG cp850	DB850UK0	English (U.K)
dBASE ENU cp437	DB437US0	English (U.S.)
dBASE ENU cp850	DB850US0	English (U.S.)
dBASE ESP cp437	DB437ES1	Spanish
dBASE ESP cp850	DB850ES0	Spanish
dBASE FIN cp437	DB437FI0	Finnish
dBASE FRA cp437	DB437FR0	French
dBASE FRA cp850	DB850FR0	French
dBASE FRC cp850	DB850CF0	French (Can.)
dBASE FRC cp863	DB863CF1	French (Can.)
dBASE HUN cp852	DB852HDC	Hungarian
dBASE ITA cp437	DB437IT0	Italian
dBASE ITA cp850	DB850IT0	Italian
dBASE KOR cp949	DB949K00	Korean

create keyword

dBASE NLD cp437	DB437NLO	Dutch
dBASE NLD cp850	DB850NLO	Dutch
dBASE NOR cp437	DB437N00	Norwegian
dBASE NOR cp865	DB865N00	Norwegian
dBASE PLK pc852	DB852P00	Polish
dBASE PTB cp850	DB850PT0	Portuguese (Bra.)
dBASE PTG cp860	DB860PT0	Portuguese
dBASE RUS cp866	DB866RU0	Russian
dBASE SLO cp852	DB852SLO	Yugoslavian
dBASE SVE cp437	DB437SV0	Swedish
dBASE SVE cp850	DB850SV0	Swedish
dBASE TRK cp857	DB857TR0	Turkish
dBASE TWN cp437	DB437TW0	Taiwanese

Field translations for tables

The following table displays the field names used in tables that are created on dBASE, Oracle, Sybase, InterBase and Informix servers:

Paradox	dBASE	Interbase	Oracle	Sybase	Informix
Alpha	Character	Varying	Character	VarChar	Character
Number	Float{20.4}	Double	Number	Float	Float
Money	Float{20.4}	Double	Number	Money	Money{16.2}
Date	Date	Date	Date	DateTime	Date
Short	Number{6.0}	Short	Number	SmallInt	SmallInt
Memo	Memo	Blob/l	Long	Text	Text
Binary	Memo	Blob	LongRaw	Image	Byte
Formatted Memo	Memo	Blob	LongRaw	Image	Byte
OLE	Memo	Blob	LongRaw	Image	Byte
Graphic	Memo	Blob	LongRaw	Image	Byte
Long	Number{11.0}	Long	Number	Int	Integer
Time	Character{8}	Character{8}	Character{8}	Character{8}	Character{8}
DateTime	Character{8}	Date	Date	DateTime	DateTime

Bool	Bool	Character{1}	Character{1}	Bit	Character
AutoInc	Number{11.0}	Long	Number	Int	Integer
Bytes	Bytes	Blob	LongRaw	Image	Byte
BCD	N/A	N/A	N/A	N/A	N/A

createIndex method**Table**

Creates an index for a table.

Syntax

```
1. createIndex ( const attrib DynArray[ ] AnyType, const fieldNames Array[ ] String )
   Logical
2. createIndex ( const attrib DynArray[ ] AnyType, const fieldNums Array[ ] SmallInt )
   Logical
```

Description

createIndex creates an index using attributes specified in a DynArray named *attrib* and the field names (or numbers) specified in an Array named *fieldNames* (or *fieldNums*). This method is provided as an alternative to the **index** structure. It is especially useful when you don't know the index structure beforehand (e.g., when the information is supplied by the user).

Each key of the DynArray must be a string. You do not have to include all the keys to use **createIndex**. Any key you omit is assigned the corresponding default value.

The following table displays the key strings and their corresponding values:

String value	Description
MAINTAINED	If True, the index is incrementally maintained. That is, after a table is changed, only that portion of the index affected by the change is updated. If False, Paradox does not maintain the index automatically. Maintained indexes typically result in better performance. Default = False (Paradox tables only).
PRIMARY	If True, the index is a primary index. If False, it's a secondary index. Default = False (Paradox tables only).
CASEINSENSITIVE	If True, the index ignores differences in case. If False, it considers case. Default = False (Paradox tables only).
DESCENDING	If True, the index is sorted in descending order, from highest values to lowest. If False, it is sorted in ascending order. Default = False.
UNIQUE	If True, records with duplicate values in key fields are ignored. If False, duplicates are included and available.
IndexName	A name used to identify this index. No default value, unless you're creating a secondary, case-sensitive index on a single field, in which case the default value is the field name. For dBASE tables, the index name must be a valid DOS filename. If you do not specify an extension, .NDX is added automatically.
TagName	The name of the index tag associated with the index specified in indexName (dBASE tables only).

createIndex method

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example 1

The following example builds a maintained secondary index for a Paradox table named CUSTOMER.DB. If the *Customer* table cannot be found or locked, **createIndex** aborts the operation.

```
method pushButton(var eventInfo Event)
var
    stTbName      String
    tbCust        Table
    arFieldNames  Array[3] String
    dyAttrib      DynArray[]AnyType
endVar

stTbName = "Customer.db"

arFieldNames[1] = "Customer No"
arFieldNames[2] = "Name"
arFieldNames[3] = "Street"

dyAttrib["PRIMARY"]    = False
dyAttrib["MAINTAINED"] = True
dyAttrib["IndexName"]  = "NumberNameStreet"

if isTable(stTbName) then
    tbCust.attach(stTbName)
    if not tbCust.lock("FULL") then
        errorShow()
        return
    endIf

    if not tbCust.createIndex(dyAttrib, arFieldNames) then
        errorShow()
    endIf

; This createIndex statement has the same effect
; as the following INDEX structure:
{
    INDEX tbCust                ; Create index for Customer.db.
        MAINTAINED
        ON "Customer No", "Name", "Street"
    ENDINDEX
}

else
    errorShow()
endIf

endMethod
```

Example 2

The following example adds a unique index tag named StatProv to the production index for a dBASE table named CUSTOMER.DBF:

```
method pushButton(var eventInfo Event)
var
    tbCust        Table
    arFieldNames  Array[1] String
    dyAttrib      DynArray[]AnyType
endVar
```

```

arFieldNames[1] = "STATE_PROV"

dyAttrib["UNIQUE"]      = True
dyAttrib["MAINTAINED"] = True

; A dBASE index name must be a valid DOS filename.
; If an extension is omitted, .NDX is appended automatically.

dyAttrib["IndexName"] = "Customer.Mdx"
dyAttrib["TagName"]   = "StatProv"

if isTable("Customer.dbf") then
    tbCust.attach("Customer.dbf")

    if not tbCust.createIndex(dyAttrib, arFieldNames) then
        errorShow()
    endif

; This createIndex statement has the same effect
; as the following INDEX structure:
{
    INDEX tbCust                                ; Create index for Customer.dbf.
      UNIQUE
    ON "STATE_PROV"                            ; Index on this field.
    TAG "StatProv" OF "Customer.dbf"           ; Name the tag "StatProv".
ENDINDEX
}

else
    errorShow()
endif

endMethod

```

cSamStd method/procedure

Table

Returns the sample standard deviation of a table's column.

Syntax

1. cSamStd (const *fieldName* String) Number
2. cSamStd (const *fieldNum* SmallInt) Number

Description

cSamStd returns the sample standard deviation for 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. **cSamStd** handles blank values as specified in the **blankAsZero** setting for the session.

The sample standard deviation calculation is based on the sample variance and uses the following formula:

$$\text{sqrt}((\text{sampVar}) * (n/(n - 1)))$$

(*sampVar* = cSamVar(*tableName*, *fieldName*) and *n* = cCount(*tableName*, *fieldName*))

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

The population standard deviation is calculated using the **cStd** method.

cSamVar method/procedure

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cSamStd (const *tableName* String, const *fieldName* String) Number
2. cSamStd (const *tableName* String, const *fieldNum* SmallInt) Number

Example

The following example calculates the sample standard deviation of test scores for the Winter quarter. This code is attached to the **pushButton** method for *showSamStd*:

```
; showSamStd:pushButton
method pushButton(var eventInfo Event)
    const
        kTbName = "winter"
    endConst

    var
        tbWinter    Table
        nuSamStd     Number
    endVar

    tbWinter.attach(kTbName)
    nuSamStd = tbWinter.cSamStd("TestScore")
    nuSamStd.view()
endMethod
```

cSamVar method/procedure

Table

Returns the sample variance of a table's column.

Syntax

1. cSamVar (const *fieldName* String) Number
2. cSamVar (const *fieldNum* SmallInt) Number

Description

cSamVar returns the sample variance for the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cSamVar** handles blank values as specified in the **blankAsZero** setting for the session.

The sample variance is calculated using the formula:

$$cVar(tableName, fieldName) * (n / (n - 1))$$
$$(n = cCount(tableName, fieldName))$$

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cSamVar (const *tableName* String, const *fieldName* String) Number
2. cSamVar (const *tableName* String, const *fieldNum* SmallInt) Number

Example

The following example calculates the sample variance of two 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
```

cStd method/procedure**Table**

Returns the standard deviation of the values in a column.

Syntax

1. cStd (const *fieldName* String) Number
2. cStd (const *fieldNum* SmallInt) Number

Description

cStd returns the population standard deviation of the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. This method handles blank values as specified in the **blankAsZero** setting for the session. Population standard deviation calculations are based on the variance. For more information, see **cVar**.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cStd (const *tableName* String, const *fieldName* String) Number
2. cStd (const *tableName* String, const *fieldNum* SmallInt) Number

Example

In the following 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")
```

cSum method/procedure

```
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
```

cSum method/procedure

Table

Returns the sum of the values in of a table's column.

Syntax

1. cSum (const *fieldName* String) Number
2. cSum (const *fieldNum* SmallInt) Number

Description

cSum returns the sum of the values in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cSum** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. cSum (const *tableName* String, const *fieldName* String) Number
2. cSum (const *tableName* String, const *fieldNum* SmallInt) Number

Example

In the following 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
```

cVar method/procedure

Table

Returns the variance of a field in a table.

Syntax

1. `cVar (const fieldName String) Number`
2. `cVar (const fieldNum SmallInt) Number`

Description

cVar returns the population variance of the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. `cVar (const tableName String, const fieldName String) Number`
2. `cVar (const tableName String, const fieldNum SmallInt) Number`

Example

In the following 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
```

delete method/procedure**Table**

Deletes a table.

Syntax

```
delete ( ) Logical
```

Description

delete deletes a table without asking for confirmation. Compare this method to **empty**, which removes data from a table but does not delete it.

If the table is open or is locked, **delete** fails.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

dropGenFilter method

Syntax

```
delete ( const tableName String ) Logical
```

Example

The following example deletes ANSWER.DB from the private directory:

```
; 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
```

dropGenFilter method

Table

Removes the filter criteria associated with a Table variable.

Syntax

```
dropGenFilter ( ) Logical
```

Description

dropGenFilter removes the filter criteria associated with a Table variable. Any indexes and ranges remain in effect in the unfiltered table.

Example

In the following example, a form contains a button named *btnCACustomers*. The **pushButton** method for *btnCACustomers* attaches a Table variable to the *Customer* table, sets filter criteria, and stores the value in the number variable *nSubTotal*. **dropGenFilter** removes the filter and the total number of records is stored in a number variable named *nTotal*. Finally, a message information box displays the number of customers in California compared to the total number of customers.

```
;btnCACustomers :: pushButton
method pushButton(var eventInfo Event)
var
    tbl          Table
    dyn           DynArray[] AnyType
    nTotal,
    nSubTotal    Number
endVar

tbl.attach("CUSTOMER.DB")

dyn["State/Prov"] = "CA"
tbl.setGenFilter(dyn)
nSubTotal = tbl.cCount("State/Prov") ;Get customers in CA.
```

```
tbl.dropGenFilter()
nTotal = tbl.nRecords()           ;Get all customers.

msgInfo("Customer Analysis", string(nSubtotal) + " out of " + string(nTotal) + "
reside in California.")
endMethod
```

dropIndex method

Table

Deletes a specified index file or tag.

Syntax

1. (Paradox tables) **dropIndex** (const *indexName* String) Logical
2. (dBASE tables) **dropIndex** (const *indexName* String
[, const *tagName* String]) Logical

Description

dropIndex deletes a specified index file or tag.

In a Paradox table, *indexName* specifies a secondary index. If you specify an empty string in *indexName*, the primary index is removed.

In a dBASE table, *indexName* specifies an .NDX file. You can also use *indexName* and *tagName* to specify an .MDX file and an index tag.

You must call **setExclusive** before calling **dropIndex** to obtain exclusive rights to the table.

dropIndex fails if the index you're trying to delete is in use, or if the table is open.

For more information about indexes, see About keys and indexes in tables in the Paradox online Help.

Example

In the following example, the **pushButton** method for *thisButton* deletes the CustName tag from an .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

    ; Get exclusive access to the table.
    salesTbl.setExclusive(Yes)
    ; 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
endif

else
    msgStop("Stop!", "Could not find Sales.dbf table.")
endif

endMethod
```

empty method/procedure**Table**

Removes all records from a table.

Syntax

```
empty ( ) Logical
```

Description

empty removes all records from a table without asking for confirmation. This operation cannot be undone. This method returns True if it succeeds; otherwise, it returns False.

empty removes information from the table, but does not delete the table itself. Compare this method to **delete**, which does delete the table.

empty first tries to gain exclusive rights to the table. If it can't, it tries to place a write lock on the table.

If **empty** gains exclusive rights, it deletes all records in the table at once. If a write lock is placed on the table, **empty** must delete each record individually.

If **empty** gains exclusive rights to a dBASE table, all records are deleted and the table is compacted. If a write lock is placed on the table, 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.)

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
empty ( const tableName String ) Logical
```

Example

The following example prompts the user for confirmation before deleting all records from the *Scratch* table:

```
; tblEmpty::pushButton
method pushButton(var eventInfo Event)
var
    tblName String
    tblVar Table
endVar
tblName = "Scratch.db"

tblVar.attach(tblName)
if isTable(tblName) then
    if msgQuestion("Empty?", "Empty " + tblName + " ?") = "Yes" then

        if tblVar.empty() then
            message("All " + tblName + " records have been deleted.")
        else
            errorShow()
        endif
    endif
else
    errorShow()
endif
endMethod
```

enumFieldNames method**Table**

Fills an array with the table's field names.

Syntax

```
enumFieldNames ( var fieldArray Array[ ] String ) Logical
```

Description

enumFieldNames fills an array named *fieldArray* with a table's field names. You must declare *fieldArray* as a resizable array before calling this method. If *fieldArray* already exists, **enumFieldNames** overwrites it without asking for confirmation.

Example

In the following example, the **pushButton** method for the *btnEnumFields* button stores field names in a resizable array and uses **view** to display the contents of the array:

```
; btnEnumFields::pushButton
method pushButton(var eventInfo Event)
var
    tbl Table
    arFieldNames Array[] AnyType
endVar

tbl.attach("Sales.dbf")
if tbl.isTable() then
    tbl.enumFieldNames(arFieldNames)
    arFieldNames.view()
else
    errorShow()
endif

endMethod
```

enumFieldNamesInIndex method**Table**

Fills an array with a table index's field names.

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

enumFieldNamesInIndex fills an array named *fieldArray* with the names of the fields in a table's index, as specified in *indexName*. You must declare *fieldArray* as a resizable array before calling this method. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

In a dBASE table, the argument *tagName* is required to specify an index tag within an .MDX file.

By default, *indexName* corresponds to the index currently being used.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

In the following example, the **pushButton** method for the *showIndexFlds* button stores field names in a resizable array and uses **view** to display the array's contents:

enumFieldStruct method

```
; 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
```

enumFieldStruct method

Table

Lists a table's field structure.

Syntax

- 1. enumFieldStruct (const *tableName* String) Logical
- 2. enumFieldStruct (*inMem* TCursor) Logical

Description

enumFieldStruct lists the field structure of a Table variable. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates a Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **struct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field	Type	Description
Field Name	A31	Specifies the name of field
Type	A31	Specifies the data type of field
Size	S	Specifies the size of field
Dec	S	Specifies the number of decimal places, or 0 if field type doesn't support decimal places
Key	AI	Specifies whether the field is a key (* = key field, blank = not key field)
_ Required Value	AI	Specifies whether the field is required (T = required, N (or blank) = Not required)

<u>Min Value</u>	A255	Specifies the field's minimum value
<u>Max Value</u>	A255	Specifies the field's maximum value
<u>Default Value</u>	A255	Specifies the field's default value
<u>Picture Value</u>	A175	Specifies the field's picture
<u>Table Lookup</u>	A255	Specifies the name of lookup table (including the full path if the lookup table is not in :WORK:)
<u>Table Lookup Type</u>	AI	Specifies the type of lookup table 0 (or blank) = no lookup table, 1 = Current field + private 2 = All corresponding + no help 3 = Just current field + help and field 4 = All corresponding + help
<u>Invariant Field ID</u>	S	Specifies the field's ordinal position in table (first field = 1, second field = 2, etc.)

Once *tableName* is created, you can modify values in the table and use it with the **struct** option in the **create** command.

Example

The following example assumes that you want a new table named *NewCust* that is similar to the *Customer* table. It also assumes that you want all of the fields in *NewCust* to be required fields. The following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans *CustFlds* and modifies the field definitions so that each record describes a required field. *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
endVar

custTbl.attach("Customer.db")
if custTbl.isTable() then

    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"
```

enumIndexStruct method

```

        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
```

enumIndexStruct method

Table

Lists the structure of a table's secondary indexes.

Syntax

- 1. enumIndexStruct (const *tableName* String) Logical
- 2. enumIndexStruct (*inMem* TCursor) Logical

Description

enumIndexStruct lists the structure of a table's secondary indexes. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory. The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field	Type	Description
infoHeader	AI	Specifies whether this record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field
szName	A255	Specifies the index name, including path
szTagName	A31	Specifies the tag name, no path (dBASE only)
szFormat	A31	Specifies the optional index type, e.g., BTREE, HASH
bPrimary	AI	Specifies whether the index is primary
bUnique	AI	Specifies whether the index is unique
bDescending	AI	Specifies whether the index is descending
bMaintained	AI	Specifies whether the index is maintained

bCaseInsensitive	AI	Specifies whether the index is case-sensitive
bSubset	AI	Specifies whether the index is a subset index (dBASE only)
bExpldx	AI	Specifies whether the index is an expression index (dBASE only)
iKeyExpType	N	Specifies the key type of index expression (dBASE only)
szKeyExp	A220	Specifies the key expression for expression index (dBASE only)
szKeyCond	A220	Specifies the subset condition for subset index (dBASE only)
FieldNo	N	Specifies the ordinal position of key field in table
FieldName	A31	Specifies the name of key field
bDescendingField	AI	Specifies whether the field is indexed in descending order
iIndexId	N	Specifies the ID of the index (generated by BDE) Used restructure to specify the addition, modification or deletion of an index

tableName also includes information for indexes that are used if the dBASE table is open. To specify which indexes to associate to a Table variable, use the **usesIndexes** method and call **enumIndexStruct** to create a table that list those indexes.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example assumes that you want a new table named *NewCust* that is similar to the *Customer* table. It also assumes that you don't want to borrow referential integrity or security information. 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
    custTC TCursor
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.
; Borrow secondary indexes from CUSTINDX.DB.
newCustTbl = CREATE "NewCust.db"
                STRUCT "CustFlds.db"
                INDEXSTRUCT "CustIndx.db"
                ENDCREATE
```

enumRefIntStruct method

```
else
  msgStop("Error", "Can't find Customer table.")
endIf

endMethod
```

enumRefIntStruct method

Table

Lists a table’s referential integrity information.

Syntax

- 1. enumRefIntStruct (const *tableName* String) Logical
- 2. enumRefIntStruct (*inMem* TCursor) Logical

Description

enumRefIntStruct lists referential integrity information for a Table variable. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* is open, this method fails. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow that table’s field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field name	Type	Description
infoHeader	AI	Specifies whether the record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field
RefName	A31	Specifies the name to identify this referential integrity constraint
OtherTable	A255	Specifies the name (including path) of the other table in the referential integrity relationship
Slave	AI	Specifies whether the table is slave, not master (i.e., the table is dependent)
Modify	AI	Specifies the update rule (Y = Cascade, blank = Prohibit)
Delete	AI	Specifies the delete rule (blank = Prohibit). Paradox does not support cascading deletes for Paradox or dBASE tables.
FieldNo	N	Specifies the ordinal position of the field in this table involved in a referential integrity relationship
aiThisTabField	A31	Specifies the name of the field in this table involved in a referential integrity relationship

Other FieldNo	N	Specifies the ordinal position of the field in the other table involved in a referential integrity relationship
aiOthTabField	A31	Specifies the name of the field in the other table involved in a referential integrity relationship

Example

The following example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. The code supplies *CustRef* to the **refIntStruct** clause in a **create** statement. When using the referential integrity structure from another table, you must use the secondary index structure.

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
    var
        tbl1, tbl2 Table
    endVar

    tbl1.attach("Customer.db")
    tbl1.enumRefIntStruct("CustRef.db")
    tbl1.enumFieldStruct("CustFlds.db")
    tbl1.enumIndexStruct("CustIdx.db")

    try
        tbl2 = CREATE "NewCust.db"
                        struct "CustFlds.db"
                        refIntStruct "CustRef.db"
                        indexStruct "CustIdx.db"
                    ENDCREATE

    onFail
        errorShow()
    endTry

endMethod
```

enumSecStruct method

Table

Lists a table's security information.

Syntax

1. enumSecStruct (const *tableName* String) Logical
2. enumSecStruct (*inMem* TCursor) Logical

Description

enumSecStruct lists the security information (access rights) of a Table variable. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* is open, this method fails. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **secStruct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

enumSecStruct method

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field name	Type	Description
infoHeader	AI	Specifies whether the record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field
iSecNum	N	Specifies the number to identify security description (first description = 1)
eprvTable	N	Specifies the table privilege value
eprvTableSym	A10	Specifies the table privilege name
iFamRights	N	Specifies the family rights value
iFamRightsSym	A10	Specifies the family rights name
szPassword	A31	Specifies the password
fldNum	N	Specifies the ordinal position of field in table
aprvFld	N	Specifies the field privilege value
aprvFldSym	A10	Specifies the field privilege name

Example

The following example creates a new table based on the security information that is associated with the *Secrets* table. The code uses **enumSecStruct** to write security information to the *SecInfo* table which is then used to create the *MySecrts* table.

```
; getSecrets::pushButton
method pushButton(var eventInfo Event)
var
    tb1, tb2 Table
endVar

tb1.attach("Secrets.db")
tb1.enumSecStruct("SecInfo.db")

tb2 = CREATE "MySecrts.db"
      LIKE "Secrets.db"
      SECSTRUCT "SecInfo.db"
      ENDCREATE

endMethod
```

Privilege values and names for enumSecStruct

The following table lists numeric values and symbolic names for table and field privileges.

Value	Name	Description
0	None	Specifies no privileges
1	ReadOnly	Specifies a read-only field or table

3	Modify	Specifies a read and modify field or table
7	Insert	Specifies insert + all of the above privileges (table only)
15	InsDel	Specifies delete + all of the above privileges (table only)
31	Full	Specifies full rights (table only)
255	Unknown	Specifies privileges unknown

Family rights values and names for enumSecStruct

The following table lists numeric values and symbolic names for family rights.

Value	Name	Description
0	NoFamRights	Specifies no family rights
1	FormRights	Specifies the right to change forms only
2	RptRights	Specifies the right to change reports only
4	ValRights	Specifies the right to change val checks only
8	SetRights	Specifies the right to change image settings
15	AllFamRights	Specifies all of the above

familyRights method

Table

Tests a user's ability to create or modify objects in a table's family.

Syntax

```
familyRights ( const rights String) Logical
```

Description

familyRights determines whether you can create or modify objects in a table's family. This method returns True if you have rights to the type of object specified in *rights*; otherwise, it returns False. *rights* is a single-letter string — F (form), R (report), S (image settings), or V (validity checks) — that indicates the object type to which you may have rights. This method preserves the functionality required by Paradox 3.5 tables but does not apply to tables created in versions of Paradox after 3.5.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
familyRights( const tableName String, rights AnyType ) Logical
```

Example

The following example determines whether you have F rights to CUSTOMER.DB.

```
; showFRights::pushButton
method pushButton(var eventInfo Event)
var
    custTB Table
endVar
```

fieldName method/procedure

```
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
```

fieldName method/procedure

Table

Returns the name of a table's field, given a field number.

Syntax

```
fieldName ( const fieldNum SmallInt ) String
```

Description

fieldName 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.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
fieldName ( const tableName String, const fieldNum SmallInt ) String
```

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
```

fieldNo method/procedure

Table

Returns the position of a field in a table.

Syntax

```
fieldNo ( const fieldName String ) SmallInt
```

Description

fieldNo returns the position of the field specified by *fieldName*, or 0 if *fieldName* is not found. Fields are numbered from left to right, beginning with 1.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
fieldNo ( const tableName String, const fieldName String ) SmallInt
```

Example

The following example displays the field number of the Date field 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
```

fieldType method/procedure**Table**

Returns the data type of a field in a table.

Syntax

1. fieldType (const *fieldName* String) String
2. fieldType (const *fieldNum* SmallInt) String

Description

fieldType returns the data type of a field. If the specified field is not found, this method returns "unknown." The following tables list the possible return values for Paradox and dBASE tables:

Paradox Field Type	Return Value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY

fieldType method/procedure

Bytes	BYTES
Date	DATE
Formatted Memo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
Long Integer	LONG
Memo	MEMO
Money	MONEY
Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
Timestamp	TIMESTAMP

dBASE Field Type	Return Value
BINARY	BINARY
CHARACTER	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. fieldType (const *tableName* String, const *fieldName* String) String
2. fieldType (const *tableName* String, const *fieldNum* SmallInt) String

Example

The following example uses a dynamic array to store the data type of each field in the *BioLife* table and 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)
  endFor
  ; Now show the contents of the DynArray.
  fldTypes.view(tblName + " field types")
else
  msgStop("Sorry", "Can't find " + tblName + " table.")
endIf
endMethod

```

getGenFilter method

Table

Retrieves the filter criteria that is associated with a Table variable.

Syntax

1. `getGenFilter (criteria DynArray[] AnyType) Logical`
2. `getGenFilter (criteria Array[] AnyType [, fieldName Array[] AnyType]) Logical`
3. `getGenFilter (criteria String) Logical`

Description

getGenFilter retrieves the filter criteria that is associated with a Table variable. This method assigns values to a dynamic array (DynArray) variable in Syntax 1, or to two Array variables that you declare and include as arguments in Syntax 2.

In Syntax 1, the DynArray *criteria* lists fields and filtering conditions as follows: the index is the field name or number (depending on how it was set), and the item is the corresponding filter expression.

In Syntax 2, the Array *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on).

If the arrays used in Syntax 2 are resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as possible, beginning with criteria field 1. If there are more array items than fields, the remaining items are empty. If there are more fields than items, this method fills the array.

In Syntax 3, filter criteria is assigned to a String variable *criteria* that you must declare and pass as an argument.

Example 1

In the following example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to fill a DynArray named *dyn* with a table's filter criteria. The code then determines whether the current criteria filters the State/Prov field with a value of CA, and resets the filter if necessary.

```

;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)

```

getGenFilter method

```
var
    custTb      Table
    dyn         DynArray[] AnyType
    keysAr      Array[] AnyType
    stFilterFld,
    stCriteria   String
endVar

stFilterFld = "State/Prov"
stCriteria  = "CA"
custTb.attach("Customer")

custTb.getGenFilter(dyn) ; Get filter information.

dyn.getKeys(keysAr)
if keysAr.contains(stFilterFld) then
    if dyn[stFilterFld] = stCriteria then
        return ; Filter is set correctly.
    endIf
else
    dyn.empty() ; Set filter criteria correctly.
    dyn[stFilterFld] = stCriteria
    custTb.setGenFilter(dyn)
endIf
endMethod
```

Example 2

In the following example, a form contains a custom method named *cmGetOrders*. This custom method is used by a button named *btnViewOrders* to set a filter and return the number of records in the filter. The following code is attached to the form:

```
;Form :: cmGetOrders
method cmGetOrders(var tbl Table) Number
var
    dynCurrent DynArray[] AnyType
    dynNew      DynArray[] AnyType
endVar

dynNew["Ship Via"] = "UPS" ;Set filter criteria.
dynNew["Total Invoice"] = " 10000"
tbl.getGenFilter(dynCurrent) ;Get the current criteria.

if dynCurrent dynNew then ;If current criteria is not
    tbl.setGenFilter(dynNew) ;the same as new criteria,
endIf ;then set new criteria.

return(tbl.cCount("Order No")) ;Return number of orders.
endMethod
```

The following code is attached to the button. It associates a Table variable with a table and calls the custom method attached to the form to operate on the data.

```
;btnViewOrders :: pushButton
method pushButton(var eventInfo Event)
var
    tbl Table
endVar

tbl.attach("ORD_JUN.DB")
view(cmGetOrders(tbl), "UPS orders over $10,000 in June")
```

```
tbl.attach("ORD_JUL.DB")
view(cmGetOrders(tbl), "UPS orders over $10,000 in July")
endMethod
```

getRange method

Table

Retrieves the values that specify a range for a Table variable.

Syntax

```
getRange ( var rangeVals Array[ ] String ) Logical
```

Description

getRange retrieves the values that specify a range for a Table variable. This method assigns values to an Array variable that you declare and include as an argument. The following table displays the array values and the corresponding range criteria:

Number of array items	Range specification
No items (empty array)	Specifies no range criteria is associated with the Table variable
One item	Specifies a value for an exact match on the first field of the index
Two items	Specifies a range for the first field of the index
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size n, specifies exact matches on the first n - 2 fields of the index. The last two array items specify a range for the n - 1 field of the index

If the array is resizable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

Example

In the following example, **getRange** is used on a Table variable *tbl* to test if the current range criteria is the same as the new range criteria. If it is not, then the new range is set using **setRange**.

```
;btnSetRange :: pushButton
method pushButton(var eventInfo Event)
  var
    arGet      Array[2] Anytype
    arSet      Array[2] Anytype
  endVar

  arSet[1] = "A"
  arSet[2] = "B"

  ;The following assumes a Table variable
  ;is declared and used elsewhere.

  tbl.getRange(arGet)      ;Get the current range.
  if arGet arSet then      ;Compare current range with new.
    tbl.setRange(arSet)    ;Show records starting with A.
  endIf
endMethod
```

index keyword

Table

Creates an index on a the specified fields of a table.

Syntax

```
1. index
   [ maintained ] tableDesc on fieldID
   endIndex
2. index tableDesc
   [ maintained ]           (Paradox)
   [ primary ]             (Paradox)
   [ caseInsensitive ]     (Paradox)
   [ descending ]         (Paradox and dBASE)
   [ unique ]              (dBASE)
   on
   { fieldDesc [ , fieldDesc ] [ to indexName ]
   |
   { keyExp
   to ndxFileName|tag tagName [ of mdxFileName ]
   |
   for condition } }
   endIndex
```

Description

index generates a primary or secondary index on the specified fields of a table. Paradox uses the index to accelerate queries and searches that access those fields.

For Paradox tables, the keywords **maintained**, **primary**, and **caseInsensitive** are available. The **primary** keyword specifies a primary index (key), which is required to create any secondary indexes. If the table has a primary index and you create another one, the new index replaces the original. A primary index must be declared on one or more consecutive fields, beginning with the first field in the table. Memo fields, formatted memo fields, OLE fields, and Graphic fields cannot be indexed.

Secondary indexes can be either maintained (created using the **maintained** keyword) or non-maintained. Paradox updates a maintained index as records are added, deleted, or changed. A non-maintained index is only updated when in use. If you use the **maintained** keyword for Paradox tables and specify a non-keyed table to index, **index** fails. For dBASE tables, all opened index files are automatically maintained.

The **caseInsensitive** keyword causes an index to ignore case. A primary index must be case-sensitive. For Paradox tables, a case-sensitive, maintained index on a single field must have the same name as that field. A case-*insensitive*, maintained index on a single field must *not* have the same name as that field.

The **on** clause specifies which fields to index and two forms: one for Paradox tables, and one for dBASE tables.

For Paradox tables, use

on fieldDesc [, fieldDesc] to indexName

(*fieldDesc* specifies one or more field names or field numbers, and *indexName* specifies the index file.

Other methods use this name to refer to the index.)

For dBASE tables, use

keyExp to ndxFileName|tag tagName [of mdxFileName]

(which lets you choose between an .NDX file or a tag in an .MDX file. If *mdxFileName* is omitted, the default .MDX filename is the same as the table. A dBASE table can only be indexed on one field or expression)

In multi-user applications, **index** places a full lock on the table while it is being indexed. If the table has already been locked by another user or application, the command is retried throughout the retry period. If the lock cannot be obtained by the end of the period, **index** fails. You can use the **lock** method to determine whether you can lock the table *before* you use the **index** command.

It can be convenient to develop your applications without worrying about indexes and introduce them where appropriate to speed up queries and searches.

The index command fails if

- 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 is inappropriate. Instead, you create an index structure to specify how to index the table.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example 1

The following example builds a primary index for a Paradox table named CUSTOMER.DB. If the Customer table can not be found, or cannot be locked, this code aborts the **index** operation. If the table is indexed, the code enumerates indexed fields to an array and displays the array's contents 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
```

Example 2

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)
```

isAssigned method

```
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
```

isAssigned method

Table

Reports whether a Table variable has an assigned value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned 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

- Even if **isAssigned** returns True, the table may not exist. For example, the following code displays True in a dialog box:

```
var tb Table endVar
tb.attach("zxcv.qw")          ; attach to some nonsense filename
msgInfo("Assigned?", tb.isAssigned()) ; displays True
```

Example

The following example determines whether the 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 Table
endVar
```

The following code is attached to the **pushButton** method for the *thisButton* button. 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
```

isEmpty method/procedure

Table

Reports whether a table contains any records.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns True if there are no records in a table; otherwise, it returns False.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isEmpty ( const tableName String ) Logical
```

Example

In the following example, the **pushButton** method for the *rptRecNo* button displays the number of records in the *Orders* table. If *Orders* is empty, this code informs the user:

```
; 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
```

isEncrypted method/procedure

Table

Reports whether a table is password-protected.

isShared method/procedure

Syntax

```
isEncrypted ( [ const tableName String ] ) Logical
```

Description

isEncrypted returns True if a table is password-protected; otherwise, it returns False. A TCursor can't be opened on an encrypted table until the password is presented interactively or using the Session type method **addPassword**. To determine whether a user has access rights to the table use **tableRights**.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isEncrypted ( const tableName String ) Logical
```

Example

The following example uses **isEncrypted** to determine whether the *Secrets* table is password-protected ; if it is, the user must enter a password.

```
method pushButton(var eventInfo Event)
  const
    kTbName = "Secrets"
  endConst

  var
    tbSecret Table
    tvSecret TableView
  endvar

  tbSecret.attach(kTbName)

  ; If the table is encrypted, prompt the
  ; user for the password.

  if tbSecret.isEncrypted() then
    menuAction(MenuFileTablePasswords)
  endIf

  if not tvSecret.open(kTbName) then
    errorShow("Could not open " + kTbName)
  endIf

endMethod
```

isShared method/procedure

Table

Reports whether a table is currently shared with another user on the network.

Syntax

```
isShared ( ) Logical
```

Description

isShared 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 with another session.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isShared ( const tableName String ) Logical
```

Example

In the following 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 that **setExclusive** has on tables in a multi-user 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("Multi-user Status", tblName + " is not shared.")
endif

endMethod
```

isTable method/procedure**Table**

Reports whether a table exists in a database.

Syntax

```
isTable ( ) Logical
```

Description

isTable returns True if the specified Table variable represents a table that can be opened; otherwise, it returns False.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
isTable ( const tableName String ) Logical
```

Example

The following example uses **isTable** to determine whether 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 and displays the contents of the array in a dialog box:

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

lock method

```
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
```

lock method

Table

Locks a specified table.

Syntax

```
lock ( const lockType String ) Logical
```

Description

lock locks a specified table. The *lockType* argument is one of the following String values, listed in order of decreasing strength and increasing concurrency:

String value	Description
Full	The current session has exclusive access to the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If successful, **lock** returns True; otherwise, it returns False.

Example

The following example attaches a Table variable to *Customer*, places an exclusive lock on the table and uses **reIndex** to rebuild the *Phone_Zip* index. When the index is rebuilt, this code unlocks *Customer* so other network users can gain access to the table:

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tblVar Table
  pdoxTbl String
endVar
pdoxTbl = "Customer.db"

if isTable(pdoxTbl) then
  tblVar.attach(pdoxTbl)
  if tblVar.lock("Full") then      ; Try to lock the table.
    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

```

nFields method/procedure

Table

Returns the number of fields in a table.

Syntax

```
nFields ( ) LongInt
```

Description

nFields returns the number of fields in a table.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
nFields ( const tableName String ) LongInt
```

Example

In the following 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

```

nKeyFields method/procedure

Table

Returns the number of fields in the primary index for a table.

Syntax

```
nKeyFields ( ) LongInt
```

Description

nKeyFields returns the number of fields in the primary index for a table. Use **getIndexName** (TCursor type) to retrieve the index's name.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

nRecords method/procedure

Syntax

```
nKeyFields ( const tableName String ) LongInt
```

Example

The following example returns the number of primary key fields in a Paradox table (ORDERS.DB). This code also returns the number of primary key fields in the LastName tag of the SCORES.MDX index for a dBASE table (SCORES.DBF):

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    pdoxTbl, dBaseTbl Table
    nkf LongInt
endVar

pdoxBtl.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("Scores", "LastName")
nkf = dBaseTbl.nKeyFields() ; key fields in LastName tag
msgInfo("Scores.dbf", "LastName tag has "
        + String(nkf) + " key fields.")

endMethod
```

nRecords method/procedure

Table

Returns the number of records in a table.

Syntax

```
nRecords ( ) LongInt
```

Description

nRecords returns the number of records in the table associated with a Table variable.

If you call **nRecords** after setting a filter, the return value does not represent the number of records in the filtered set. To retrieve that information, use **cCount**. If you call **nRecords** after setting a range, the return value represents the number of records in the set defined by the range.

nRecords counts deleted records in dBASE tables if **showDeleted** is turned on. If showDeleted is turned off, deleted records are not counted.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
nRecords ( const tableName String ) LongInt
```

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"

if isTable(tblName) then
    tblVar.attach(tblName)
    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

```

protect method/procedure

Table

Assigns an owner password to a table.

Syntax

```
protect ( const password String ) Logical
```

Description

protect assigns an owner password to a table. The password cannot exceed 31 characters. A password-protected table cannot be accessed without presenting the password specified in *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, and the **removePassword** method to withdraw the password. *password* is case-sensitive (e.g., 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.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
protect ( const tableName String, const password String ) Logical
```

Example

In the following 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

```

reIndex method

```
secretData.protect("Get007") ; Password-protect table with "Get007"
endIf

endMethod
```

reIndex method

Table

Rebuilds an index or index tag that is not automatically maintained.

Syntax

1. (Paradox tables) **reIndex** (const *indexName* String) Logical
2. (dBASE tables) **reIndex** (const *indexName* String [const *tagName* String]) Logical

Description

reIndex rebuilds an index or index tag that is not automatically maintained. In a Paradox table, use *indexName* to specify an index. In a dBASE table, use *indexName* to specify an .NDX file, or *indexName* and *tagName* to specify an index tag in an .MDX file. **reIndex** requires exclusive access to the table.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example attaches a Table variable to a Paradox table named Customer, places an exclusive lock on the table and uses **reIndex** to rebuild the *Phone_Zip* index:

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
    tblVar Table
    pdoxTbl String
endVar
pdoxBtl = "Customer.db"

tblVar.attach(pdoxBtl)
if tblVar.lock("Exclusive") then ; Try to lock the table.
    tblVar.reIndex("Phone_Zip") ; Rebuild Phone_Zip index.
    tblVar.unlock("Exclusive") ; Unlock the table.
else
    msgStop("Sorry", "Can't lock " + pdoxBtl + " table.")
endIf

endMethod
```

reIndexAll method

Table

Rebuilds all index files associated with a table.

Syntax

reIndexAll () Logical

Description

reIndexAll rebuilds all index files associated with a table. This method requires exclusive rights to rebuild a maintained index and a write lock to rebuild a non-maintained index.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

In the following example, the **pushButton** method for a button attempts to place an exclusive lock on the *Customer* table. If **lock** is successful, this code rebuilds all indexes for the *Customer* table and 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("Exclusive") then      ; attempt to lock Customer.db
    tblVar.reIndexAll()                ; rebuild all Customer.db indexes
    tblVar.unlock("Exclusive")         ; unlock the table
else
    msgStop("Sorry", "Can't lock " + pdoxTbl + " table.")
endif

endMethod
```

rename method/procedure**Table**

Renames a table.

Syntax

```
rename ( const destTableName String ) Logical
```

Description

rename changes a table's name to the name specified by *destTableName*. If the table named by *destTableName* already exists, an error results.

Throughout the retry period, this method attempts to place a full lock on the table. If the lock cannot be placed, an error results.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
rename ( const tableName String, const destTableName String ) Logical
```

Example

The following example renames CUSTOMER.DB to OLDCUST. If *OldCust* already exists, this code allows you to abort the operation:

```
; renameCust::pushButton
method pushButton(var eventInfo Event)
var
    tblVar Table
    oldName, newName String
endVar

oldName = "Customer.db"
newName = "OldCust.db"
```

restructure method

```
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
```

restructure method

Table

Syntax

```
restructure ( const createSpec DynArray[ ] AnyType ) Logical
```

Description

restructure allows you to modify a table's structure under program control. You can add, delete, or modify the fields in your table, create indexes, change referential integrity relationships, and so on. **restructure** also allows you to perform other operations that are available when restructuring a table (e.g., packing the table).

restructure uses a dynamic array named *createSpec*, which contains the information on the changes to make to the table.

To specify the kind of change to be made, the field, index, referential integrity, and security structure tables, if specified, must include an ID field. This ID field is named slightly differently in each structure file, however, all end with id. Use this ID field to specify the type of operation to perform. The RestructureOperations constants *RestructureModify*, *RestructureAdd*, and *RestructureDrop* are provided for each operation.

restructure returns True if successful, and False if the restructure operation fails, or if a Keyviol or Problems table is generated. It also fails if it cannot obtain a full lock on the table.

The following clauses specify table attributes in *createSpec* and are optional. They can appear in any order within the dynamic array:

saveAs specifies a new name for the restructured table, leaving the original table unchanged. By default, the restructured table is saved with the same name, which overwrites the original table.

Keyviol specifies the table to which any records causing a key violation are saved.

Problems specifies the table to which any problem records are saved. If data is lost during the restructure, the problem records are placed in the problems table, and the operation that caused the problem is not performed on those records.

fieldStruct specifies the name of the table from which you can borrow field structure information. Use **enumFieldStruct** to generate the field structure table before executing **restructure**. The *_Invariant Field Id* field of the field structure table contains the original field number of the table to be restructured. Use the *_Invariant Field Id* field to specify the change to be made to the table. To add a field, insert a new record in the field structure table describing the new field and place *RestructureAdd* in the *_Invariant Field Id* field. To delete an existing field, remove the record from the field structure table.

indexStruct specifies the name of the table from which you can borrow index structure information. Use **enumIndexStruct** to generate the index structure table (or create it manually). Use the *iIndexId* field of the index structure table to specify the change to be made to the table. To modify an index, use *RestructureModify* in the *iIndexId* field. You can modify an index name by dropping (*RestructureDrop*), and adding (*RestructureAdd*) another record with the changed name.

refIntStruct specifies the name of the table from which you can borrow referential integrity structure information. Use **enumRefIntStruct** to generate the referential integrity structure table (or create it manually). Use the *iRefId* field of the referential integrity structure table to specify the change to be made to the table. Use *RestructureModify* to modify existing values, *RestructureAdd* to add, or *RestructureDrop* to delete.

secStruct specifies the table from which you can borrow security structure information. Use **enumSecStruct** to generate the security structure table name (or create it manually). Use the *iSecId* field in the security structure table to specify the change to be made to the table. Use *RestructureModify* to modify existing values, *RestructureAdd* to add, or *RestructureDrop* to delete.

pack specifies whether to pack the table. Valid values are True or False. For more information, see **compact** (Table type).

versionLevel specifies the table version level. See **create** for a listing of version numbers for Paradox and dBASE tables.

languageDriver specifies the language driver name. For a list of language drivers for Paradox tables, see Language drivers for Paradox tables. For dBASE tables, see Language drivers for dBASE tables.

blockSize specifies the size of data blocks used to store information in the table, in kilobytes. (A kilobyte is 1,024 bytes.) Valid block sizes depend on the file format of the table. For Paradox versions 4.5 or earlier, 1K through 4K are valid. For versions 5.0 and later, 1K through 4K, 8K, 16K, and 32K are valid.

warnings specifies whether warnings encountered during the restructure operation are displayed. Valid values are True or False. If **warnings** is set to True, warnings can be placed on the error stack for examination.

If **errorTrapOnWarnings** is set to True, the first warning generated terminates the restructure operation.

Example

The following example appears in a script window and modifies the Customer table by changing a field name. This code uses **enumFieldStruct** to create the field structure information, updates the information, copies the updated information to a dynamic array and passes the dynamic array to the restructure method:

```
method run(var eventInfo Event)
var
    tbl          Table
    tcFlds       TCursor
    dynNewStru   DynArray[] Anytype
endvar

tbl.attach( "Customer.db" )
tbl.enumFieldStruct( "field_struct.db" )

tcFlds.open("field_struct.db" )
tcFlds.edit()

scan tcFlds :
    if tcFlds."Field Name" = "Name" then
        tcFlds."Field Name" = "Company Name"
```

setExclusive method

```
        quitLoop
    endif
endscan
tcFlds.endEdit()
tcFlds.close()

dynNewStru["FIELDSTRUCT"] = "field_struct.db"
tbl.restructure( dynNewStru )
endmethod
```

setExclusive method

Table

Specifies whether to grant the user exclusive rights to a table when it is opened.

Syntax

```
setExclusive ( [ const yesNo Logical ] )
```

Description

setExclusive specifies whether to open a table with shared or exclusive rights. This method does not place locks on the table — an exclusive lock is placed on the table only when it is opened. Exclusive locks are more powerful than full locks.

By default, tables are opened in shared mode. Optional argument *yesNo* specifies whether to set exclusive rights. A value of Yes requests exclusive rights so that no other user can read or write to the table; a value of No allows the table to be opened in shared mode. By default, *yesNo* is set to Yes.

Example

The following example demonstrates how **setExclusive** affects access rights to a table. This code defines a Table variable for the *Customer* table and calls **setExclusive** so *Customer* is opened exclusively. Then, a TCursor is opened for *Customer*. If the TCursor is successfully opened, it has exclusive rights to the table and **lockStatus** is called 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
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

```

setGenFilter method

Table

Specifies conditions for including records in a TCursor opened on a Table variable.

Syntax

1. setGenFilter (*criteria* DynArray[] AnyType) Logical
2. setGenFilter (*criteria* Array[] AnyType [, *fieldId* Array[] AnyType]) Logical

Description

setGenFilter specifies conditions for including records in a TCursor opened on a Table variable. Records that meet the specified conditions are included in the TCursor. Records that don't meet the criteria are filtered out, creating a restricted view of the table. **setGenFilter** must be executed before opening a table with a TCursor.

In Syntax 1, a dynamic array (DynArray) named *criteria* specifies fields and filtering conditions. The index is the field name or number, and the item is the filter expression.

The following code specifies criteria based on the values of three fields:

```

criteriaDA[1]      = "Widget"           ; The value of the first field
                                     ; in the table is Widget.

criteriaDA["Size"] = " 4"               ; The value of the field named
                                     ; Size is greater than 4.

criteriaDA["Cost"] = "= 10.95, 22.50" ; The value of the field named
                                     ; Cost is greater than or
                                     ; equal to 10.95 and less
                                     ; than 22.50.

```

If the DynArray is empty, all existing filter criteria are removed.

In Syntax 2, an Array named *criteria* specifies filtering conditions, and an optional Array named *fieldId* specifies field names and numbers. If you omit *fieldId*, conditions are applied to fields in the order they appear in the *criteria* array (the first condition applies to the first field, the second condition applies to the second field, and so on). The following example specifies the same criteria as the example for

Syntax 1:

```

criteriaAR[1] = "Widget"
criteriaAR[2] = " 4"
criteriaAR[3] = "= 10.95, 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"

```

If the Array is empty, all existing filter criteria are removed.

Filtering on special characters

If you are filtering on special characters, you must precede the number or literal value that can be interpreted as an operator (like `/`, `\`, `—`, `+`, `=`, etc.) with a backslash(`\`). In `setgenfilter()`, the filter criteria is put into a string and parsed to pick out numbers and operators for calculations. If the number or operator in the filter needs to be interpreted literally, it needs to be preceded by a backslash(`\`). For example to filter a table with the following records:

```
1st Base
1st Love
2nd Base
3rd Base
```

and retrieve only those that start with "1st," the filter would look like the following:

```
filter = "\\1st.."
```

One backslash for the number and another to indicate the first backslash is not an escape sequence.

Example 1

In the following example, the built-in **run** method for a script attaches a Table variable to the *Customer* table and sets filter criteria on the *State* field to equal CA:

```
;Script :: run
method run(var eventInfo Event)
    var
        tb          Table
        dyn          DynArray[] AnyType
    endVar

    dyn["State/Prov"] = "CA"

    tb.attach("CUSTOMER.DB")
    tb.setGenFilter(dyn)

endMethod
```

Example 2

In the following example, a form contains a button named *btnBalanceStatus*. The **pushButton** method for *btnBalanceStatus* attaches a Table variable to the *Orders* table and sets filter criteria that displays only those records with a positive balance. **cCount** then retrieves the number of records, **cAverage** retrieves the average balance due, and **cSum** retrieves the total balance due. Finally, a dialog box displays the values.

```
;btnBalanceStatus
method pushButton(var eventInfo Event)
    var
        tbl          Table
        dyn          DynArray[] AnyType
        s1,
        s2,
        s3           String
    endVar

    tbl.attach("ORDERS")
    Dyn["Balance Due"] = " 0"
    tbl.setGenFilter(Dyn)

    s1 = string(tbl.cCount("Balance Due"))
    s2 = string(tbl.cAverage("Balance Due"))
    s3 = string(tbl.cSum("Balance Due"))
```

```

    msgInfo("Outstanding balances", "There are " + s1 + " orders with an average
balance due of " + s2 + ", totaling " + s3 + ".")
endMethod

```

setIndex method

Table

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

setIndex specifies an index to use when a table is opened.

In a Paradox table, use *indexName* to specify an index. In a dBASE table, you can use *indexName* to specify an .NDX file, or *indexName* and *tagName* to specify an index tag in an .MDX file.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example assumes that the Paradox Customer table has a secondary index named CityState. The following code specifies CityState with **setIndex** to set up for a call to **setRange**. When the filter is set for *Customer*, this code loads a DynArray with information from the filtered table then displays the DynArray's contents 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.setRange("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
endIf

```

setRange method

```
else
  msgStop("Error", "Can't find Customer.db")
endIf
endMethod
```

setRange method

Table

Specifies a range of records to associate with a Table variable. This method enhances the functionality of **setFilter**, which it replaces in this version. Code that calls **setFilter** continues to execute as before.

Syntax

```
1. setRange ( [ const exactMatchVal AnyType ] * [ , const minVal AnyType, const maxVal AnyType ] ) Logical
2. setRange ( rangeVals Array[ ] AnyType ) Logical
```

Description

setRange specifies conditions for associating a contiguous range of records with a Table variable. Records that meet the conditions are included when the table is opened. **setRange** compares the criteria you specify with values in the corresponding fields of a table's index. If the table is not indexed, this method fails. If you call **setRange** without any arguments, the range criteria is reset to include the entire table.

Syntax 1 allows you to set a range based on the value of the first field of the index by specifying values in *minVal* and *maxVal*. For example, the following statement examines values in the first field of the index of each record:

```
tableVar.setRange(14, 88)
```

If a value is less than 14 or greater than 88, that record is filtered out. To specify an exact match on the first field of the index, assign the same value to *minVal* and *maxVal*. For example, the following statement filters out all values except 55:

```
tableVar.setRange(55, 55)
```

To set a range based on the values of more than one field, specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on Corel and Paradox (assuming they are the first fields in the index), and values ranging from 100 to 500 (inclusive) for the third field:

```
tableVar.setRange("Corel", "Paradox", 100, 500)
```

Syntax 2 allows you to pass an array of values to specify the range criteria, as listed in the following table.

Number of Array Items	Range Specification
No items (empty array)	Resets range criteria to include the entire table
One item	Specifies a value for an exact match on the first field of the index
Two items	Specifies a range for the first field of the index
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size n, specify exact matches on the first n-2 fields of the index. The last two array items specify a range for the n-1 field of the index.

Example 1

The following example assumes that *Lineitem*'s key field is Order No. and that 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 and uses **cSum** to calculate the total for order number 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.setRange(1005, 1005)

    ; now display the total for Order No. 1005
msgInfo("Total for Order 1005", tblVar.cSum("Total"))

endMethod
```

Example 2

The following example calls **setRange** with a criteria array that contains more than three items. The following code sets a range to include orders from a person with a specific first name, middle initial, and last name, and an order quantity ranging from 100 to 500 items. The code then counts the number of records in this range and displays the value in a dialog box. This example assumes that the *PartsOrd* table is indexed on the *FirstName*, *MiddleInitial*, *LastName*, and *Qty* fields:

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
var
    tbPartsOrd Table
    arRangeInfo Array[5] AnyType
    nuCount Number
endVar

arRangeInfo[1] = "Frank" ; FirstName (exact match)
arRangeInfo[2] = "P." ; MiddleInitial (exact match)
arRangeInfo[3] = "Corel" ; LastName (exact match)
arRangeInfo[4] = 100 ; Minimum qty value
arRangeInfo[5] = 500 ; Maximum qty value

if tbPartsOrd.attach("PartsOrd") then
    tbPartsOrd.setRange(arRangeInfo)
    nuCount = tbPartsOrd.cCount(1)
    nuCount.view("Number of big orders by Frank P. Corel:")
else
    errorShow("Can't open the table.")
endif
endMethod
```

setReadOnly method**Table**

Specifies whether to grant the user read-only rights to a table when it is opened.

showDeleted method

Syntax

```
setReadOnly ( [ const yesNo Logical ] )
```

Description

setReadOnly specifies whether to grant 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 No allows full rights to the table. By default, *yesNo* is set to Yes.

Example

The following example attaches a Table variable to the Orders table, calls **setReadOnly** to limit rights and opens a TCursor for Orders:

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

errorTrapOnWarnings()
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
tc.edit()

endMethod
```

showDeleted method

Table

Specifies whether to display deleted records in a dBASE table.

Syntax

```
showDeleted ( [ const yesNo Logical ] ) Logical
```

Description

showDeleted specifies whether to display deleted records in a dBASE table. Records deleted from a dBASE table aren't immediately removed. Instead, they are flagged for deletion and removed later.

showDeleted is relevant only for dBASE tables.

Optional argument *yesNo* specifies whether to display deleted records (a value of Yes) or hide deleted records (a value of No). By default, *yesNo* is set to Yes. If you don't call this method before using the Table variable associated with the table, deleted records are not displayed.

Example

In the following example, the **pushButton** method attached to the *showDeletedRecs* button displays a Table variable's deleted records:

```
; 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

```

sort keyword

Table

Sorts a table.

Syntax

```
sort sourceTable [ on fieldNameList [ D ] ] [ to destTable ]    endSort
```

Description

sort sorts the table specified in *sourceTable*.

sourceTable can be a Table, TCursor, or String type. *destTable* can be a Table or String type. However, you can't sort a TCursor onto itself.

If you include the optional **on** clause, the table is sorted on the first field specified in *fieldNameList*. Each subsequent field settles 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 sort result is written to the table specified by *destTable*. If that table already exists, it is overwritten without asking for confirmation. If you omit the **to** clause, the sorted records are returned to *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 the tables being sorted if the result is written to the same table. Otherwise, a write lock is required for the source table and a full lock for the target table.

sort is not a method, so dot notation is inappropriate. Instead, you create a structure to specify how to sort the table.

Example

The following example sorts *Customer* on the Last Name and First Name fields, and displays the results in the *CustSort* table:

```

; sortCustTable::pushButton
method pushButton(var eventInfo Event)
var
    custTbl Table
    tv TableView
endVar

custTbl.attach("Customer.db")

sort custTbl
    on "Country" D, "Name" D    ; sort in descending order
    to "CustSort.db"
endSort

tv.open("CustSort.db")        ; open the sorted table

endMethod

```

subtract method/procedure**Table**

Subtracts the records in one table from another table.

Syntax

1. subtract (const *destTableName* String) Logical
2. subtract (const *destTableName* Table) Logical

Description

subtract determines whether records that reside in the source table also reside in *destTableName*. If matching records are found, **subtract** deletes them from *destTableName* without asking for confirmation.

If *destTableName* is keyed, **subtract** deletes the records with keys that match the values of key fields in the source table. If *destTableName* is not keyed, **subtract** deletes the records that match any record in the source table. Whether tables are keyed or not, this method considers only fields that *could* be keyed (based on data type, not position). For example, numeric fields are considered, but formatted memos are not. This method requires read/write access to both tables.

If the target table is not keyed, this operation can be time-consuming.

Throughout the retry period, this method attempts to place a full lock on both tables. If locks cannot be placed, an error results.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

1. subtract (const *sourceTableName* String, const *destTableName* String) Logical
2. subtract (const *sourceTableName* String, const *destTableName* Table) Logical

Example

The following example subtracts records found in the *Inserted* table (in the private directory) from the Customer table:

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
    instTbl, CustTbl Table
    fs FileSystem
    tblName String
endVar
tblName = privDir() + "\\Inserted.db"

instTbl.attach(tblName)
if instTbl.isTable() then
    instTbl.subtract(custTbl)    ; remove from custTbl matching records in instTbl
else
    msgInfo("Sorry", "Can't find " + tblName + " table.")
endif

endMethod
```

tableRights method/procedure**Table**

Specifies whether the user has the right to perform table operations.

Syntax

```
tableRights ( const rights String ) Logical
```

Description

tableRights specifies whether the user has the right to perform table operations. The following table describes *rights*:

Value	Description
ReadOnly	Specifies the right to read from the table without making changes
Modify	Specifies the right to enter or change data
Insert	Specifies the right to add new records
InsDel	Specifies the right to add and delete records
Full or All	Specifies the right to perform all of the above operations

This method returns True if the user has the specified rights; otherwise, it returns False.

DOS

If you are a DOS PAL programmer, you can use this procedure to operate on tables by specifying the table name, rather than using a variable.

Syntax

```
tableRights ( const tableName String, const rights String )
```

Example

The following 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
```

type method**Table**

Returns a table's type.

Syntax

```
type ( ) String
```

unAttach method

Description

type returns the string value PARADOX or DBASE to specify the table's type.

Example

The following example removes deleted records from the *Orders* table if **type** returns DBASE. If **type** returns Paradox, a message is displayed:

```
; 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
```

unAttach method

Table

Ends the association between a Table variable and a table description.

Syntax

```
unAttach ( ) Logical
```

Description

unAttach ends the association (created using **attach** or **create**) between a Table variable and a table description. 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 the following example, a Table variable is used to summarize sales information from two different tables. When 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

switch
    case q2 q1 : msg = "Sales are down."
```

```

        case q2 = q1 : msg = "Sales are flat."
        case q2  q1 : msg = "Sales are up."
    endSwitch

    msgInfo("Sales", msg)

endMethod

```

unlock method

Table

Unlocks a specified table.

Syntax

```
unlock ( const lockType String ) Logical
```

Description

unlock removes locks that are explicitly placed on the table associated with a Table variable. *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency:

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

unlock removes locks that have been explicitly placed by a particular user or application using **lock**. **unlock** has no effect on locks placed automatically by Paradox. To ensure maximum concurrent availability of tables unlock a table that has been explicitly locked as soon as the lock is no longer needed. If you lock a table twice, you must unlock it twice. You can use **lockStatus** (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. If you try to unlock a table that isn't locked or cannot be unlocked, **unlock** returns False .

If successful, this method returns True; otherwise, it returns False.

Example

In the following example, the **pushButton** method for *updateCust* runs a query from an existing file and 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 is placed, this code adds *Answer* records and uses **unlock** to unlock *Customer*:

```

; updateCust::pushButton
method pushButton(var eventInfo Event)
var
    newCust Query
    ansTbl Table
    destTbl String
endVar
destTbl = "Customer.db"

newCust.readFromFile("getCust.qbe")

if newCust.executeQBE() then                ; If the query succeeds,

```

```

ansTbl.attach(":PRIV:Answer.db")
if destTbl.lock("Write") then          ; try to write lock the 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

```

unProtect method/procedure

Table

Permanently removes an owner password from a table.

Syntax

1. (Procedure) unProtect (const *tableName* String [, const *Password* String])
2. (Method) unProtect ([const *password* String])

Description

unProtect permanently removes an owner password from a table. A protected table is encrypted and cannot be accessed without presenting the password that is specified in *password*. If you have already issued the master password for a table, *password* is not necessary.

Example

The following 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

```

usesIndexes method

Table

Specifies index files to use and maintain with a dBASE table.

Syntax

```
usesIndexes ( const indexFileName String [ , const indexFileName String ] * Logical
```

Description

usesIndexes specifies one or more index files (.NDX and .MDX) to maintain while you use a dBASE table. This method specifies index files to open when the table is opened. This method is not used to open production files (e.g., the .MDX file with the same name as the table) for a dBASE table. These files are opened automatically.

If any of the specified index files do not exist, this method fails.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example calls **usesIndexes** to specify two different indexes in the *Orders* table and 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
```

Using ranges and filters

Table

Although ranges and filters allow you to select a subset of the records in a Table variable, a TCursor, or a UIObject, they operate differently.

A range is based on the fields in an index. When you apply a range to a table, a subset of records that are contiguous and consecutive is created. For this reason, a range gives faster performance than a filter.

A filter offers greater flexibility when selecting fields and specifying criteria. A filter is based on any table field and can use expressions to specify criteria. For example, a filter can select records in which the Quantity field has values of 125, 200, and 350. A range could only specify values ranging from 125 to 350.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

TableView type

A TableView object displays a table's data 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 is a programmatic construct that points to the data in a table.

If you declare a TableView variable and open a TableView object to that variable, a handle to the TableView window is created. You can refer to the handle in your code to manipulate the TableView object.

action method

TableView methods are a subset of the Form type methods and control the Table window's size, position, and appearance. Although you can start and end Edit mode for a table view, you cannot use ObjectPAL to directly edit the data in a table view. You can use ObjectPAL to manipulate TableView properties in the following areas:

- the TableView object as a whole (e.g., background color, grid style, number of records, and the value of the active record)
- the field-level data in the table (e.g., font, color, and display format — (TVData))
- the TableView heading (e.g., font, color, and alignment — (TVHeading))

The TableView type includes several derived methods from the Form type.

Methods for the TableView type

Form	←	TableView
bringToTop		action
getPosition		close
getTitle		moveToRecord
hide		open
isAssigned		wait
isMaximized		
isMinimized		
isVisible		
maximize		
minimize		
setPosition		
setTitle		
show		
windowHandle		

action method

TableView

Performs an action command.

Syntax

```
action ( const actionID SmallInt ) Logical
```

Description

action performs the action specified by the constant *actionId*. *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

You can also use **action** to send a user-defined action constant to a built-in **action** method. User-defined action constants are integers that don't interfere with any of ObjectPAL's constants.

User-defined constants can be used to signal other parts of an application. For example, assume that the Const window for a form declares a constant named *myAction*. You can use the **id** method to verify the value of each incoming ActionEvent. If the value is equal to *myAction*, you can respond to that action accordingly. By default, Paradox passes the action to the **action** method.

The **action** method is distinct from the built-in **action** method for a TableView or for any form or UIObject. The built-in **action** method for an object responds to an action event; this method causes an ActionEvent.

Example

The following example opens a TableView for the *Orders* table, moves the cursor to the end of the table, starts Edit mode, and inserts a new 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
```

close method

TableView

Closes a table window.

Syntax

```
close ( )
```

Description

close closes a table window. **close** performs the same function as the Close command in the Control menu.

Example

In the following example, a form's **open** method opens a TableView object for the *Customer* table to a global variable named *custTV*. When the form closes, its **close** method 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
```

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

isAssigned method

```
; thisForm::Var
Var
  custTV TableView ; global to form, the TableView object is opened by
                    ; form's open method
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
    custTV.open("Customer") ; open the Customer table view
endif
endMethod
```

isAssigned method

TableView

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just Table View.

Example

The following 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 is attached 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
```

moveToRecord method

TableView

Moves to a specific record in a table.

Syntax

```
moveToRecord ( const tc TCursor ) Logical
```

Description

moveToRecord moves to the record pointed to by a TCursor named *tc*. Use the **RecNo** property to accelerate performance in dBASE tables.

Example

The following example uses a TCursor to search for a customer named Jones and calls **moveToRecord** to display that record. The following code is attached to a button's built-in **pushButton** method:

```
method pushButton (var eventInfo Event)
var
    custTC TCursor
    custTV TableView
endVar

custTC.open ("customer.db")
custTV.open ("customer.db")

if custTC.locate ("Name", "Ocean Paradise") then
    custTV.moveToRecord (custTC)
else
    msgInfo("Search failed", "Couldn't find Ocean Paradise.")
endif

endMethod
```

open method**TableView**

Opens a table window.

Syntax

```
1. open ( const tvName String [ , const windowStyle LongInt ] ) Logical
2. open ( const tvName String, const windowStyle LongInt, const x SmallInt, const y
SmallInt, const w SmallInt, const h SmallInt ) Logical
```

Description

open opens the table specified by *tvName* in a table window. Optional arguments specify (in twips) the location of the upper-left corner of the form (*x* and *y*), the form's width and height (*w* and *h*), and the form's style (*windowStyle*). The *windowStyle* argument is required for Syntax 2. To specify a size and position for the form, use a window style constant (WinStyleDefault).

Example

In the following example, the **pushButton** method for a button named *openWaitOrders* opens the *Orders* table:

```
; 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
```

wait method**TableView**

Suspends a method's execution.

Syntax

```
wait ( )
```

Description

wait suspends a method's execution. Execution resumes when the TableView object is closed. When a TableView object has been called by **wait**, the method suspends execution until the TableView object is closed using the **close** method.

Example

See the **open** example.

Tcursor type

A TCursor is a pointer to data that is contained in a table. Using TCursors, you can manipulate a table's data without displaying the table. When you edit records in a TCursor, the underlying table is changed. Locks on the table affect the TCursor. A TCursor can point to an entire table or to a subset of the records in a table (e.g., those specified by a restricted view, detail set, filter, or range).

For more information about related objects, see the Table, TableView, and UIObject types.

Some table operations require Paradox to create temporary tables in the private directory.

Methods for the TCursor type

add	fieldSize	reIndexAll
atFirst	fieldType	seqNo
atLast	fieldUnits2	setBatchOff
attach	fieldValue	setBatchOn
attachToKeyViol	forceRefresh	setFlyAwayControl
bot	insertRecord	getGenFilter
cancelEdit	instantiateView	getIndexName
cAverage	isAssigned	getLanguageDriver
cCount	isEdit	getLanguageDriverDesc
close	isEmpty	getRange
cMax	isEncrypted	home
cMin	isRecordDeleted	initRecord
cNpv	isShared	insertAfterRecord
compact	isShowDeletedOn	insertBeforeRecord
copy	isValid	setFieldValue
copyFromArray	isView	setGenFilter
copyRecord	locate	setRange
copyToArray	locateNext	showDeleted
cSamStd	locateNextPattern	skip
cSamVar	locatePattern	subtract
cStd	locatePrior	switchIndex
empty	locatePriorPattern	tableName
end	lock	moveToRecNo
endEdit	lockRecord	nextRecord
enumFieldNames	lockStatus	nFields
enumFieldNamesInIndex	moveToRecord	nKeyFields
enumFieldStruct	cSum	nRecords
enumIndexStruct	currRecord	open
enumLocks	cVar	postRecord
enumRefIntStruct	deleteRecord	priorRecord
enumSecStruct	didFlyAway	qLocate
enumTableProperties	dropGenFilter	tableRights
eot	dropIndex	type

add method

familyRights	edit	unDeleteRecord
fieldName	recNo	unlock
fieldNo	recordStatus	unlockRecord
fieldRights	reIndex	updateRecord

add method

TCursor

Adds records from one table to another table.

Syntax

```
1. add ( const destTable String [ , const append Logical [ , const update Logical ] ] ) Logical
2. add ( const destTable Table [ , const append Logical [ , const update Logical ] ] ) Logical
3. add ( const destTable TCursor [ , const append Logical [ , const update Logical ] ] ) Logical
```

Description

add adds the records pointed to by a TCursor to the target table specified in *destTable*. If the destination does not exist, this method creates it. The source table and the target table can be the same type or different types; in any case, the tables must have compatible field structures.

When set to True, *append* adds records at the end of a non-indexed target table, or at the appropriate place in an indexed target table. When set to True, *update* compares records in both tables, and where key values match, replaces the data in the target table. When both are set to True, records with matching active indices (key value being the default active index) are updated, and others are appended. These arguments are optional, but if you specify *update*, you must also specify *append*. By default, both arguments are True.

```
myTCursor.add("yourTable", False, True) ; specifies update
myTCursor.add("yourTable") ; specifies update and append by default
```

Key violations (including validity check violations) are listed in KEYVIOL.DB in the private directory. If KEYVIOL.DB already exists, **add** overwrites it. If KEYVIOL.DB does not exist, this method creates it.

When tables are keyed, **add** uses the keyed fields to determine which records to update and which to append. If the target table is not keyed and update is set to True, **add** fails. **add** respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period this method tries to place write locks on the source and target tables. If either lock cannot be placed, the method fails.

Example

The following example assumes that the *OldCust* and *NewCust* tables exist in the active directory. The following code associates a TCursor with each of the tables, adds *NewCust* records to *OldCust* and adds all records to a table named *MyCust*. If *MyCust* does not exist in the active directory, **add** creates it. This code is attached to a button's **pushButton** method:

```
; getMyCust::pushButton
method pushButton(var eventInfo Event)
var
    TC1, TC2 TCursor
endVar

if TC1.open("oldCust.db") and
    TC2.open("newCust.db") then ; if both TCursors can be associated
```

```

TC2.add(TC1, True)           ; append oldCust records to newCust
                             ; records — now TC1 has
                             ; records from both tables
TC1.add("myCust.db", True)   ; add TC1 to myCust table

TC1.close()                 ; close both TCursors
TC2.close()

else
  msgStop("Stop!", "Could not open one or more tables.")
endif

endMethod

```

aliasName method

TCursor

Returns a TCursor's alias.

Syntax

```
aliasName ( ) String
```

Description

aliasName returns a string containing a TCursor's alias. Only TCursors that were opened using an alias returns an alias name. If the TCursor was not opened using an alias, **aliasName** returns an empty string.

Example

The following example uses **aliasName** to determine the OPEN MODE property value for the open TCursor:

```

method pushButton(var eventInfo Event)
var
  actualPropVal,
  expectedPropVal,
  propertyName,
  tableName      String
  tc              TCursor
endVar

; initialize variables
propertyName = "OPEN MODE"           ; SQL alias property name
expectedPropVal = "READ/WRITE"       ; SQL alias property value
tableName = ":Interbase4:Customer"    ; SQL table name (includes
                                     ; the SQL alias name)

if tc.open( tableName ) then
  ; Get the current property value by specifying the alias name
  ; using tc.aliasName() and compare with expected value
  actualPropVal = getAliasProperty( tc.aliasName(), propertyName )

  if actualPropVal = expectedPropVal then
    msgInfo("SQL Table Access Mode", actualPropVal)
  else
    ; try to set to the desired property by specifying the alias
    ; name using tc.aliasName() and notify the user
    setAliasProperty( tc.aliasName(), propertyName, expectedPropVal )
    msgInfo("SQL Table Access Mode Updated", actualPropVal)
  endif
endif
endIf
endMethod

```

atFirst method**TCursor**

Reports whether the TCursor is pointing to the table's first record.

Syntax

```
atFirst ( ) Logical
```

Description

atFirst returns True if the TCursor is pointing to the table's first record; otherwise, it returns False.

Example

The following example assumes that 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 whether the TCursor points to the first record. If **atFirst** returns False, this code moves the TCursor 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 it
    orders.moveToRecord(tc) ; move highlight to first record
else
    msgStop("Currently on record " + String(tc.recNo()),
            "You're already at the top of the list!")
endif
endMethod
```

atLast method**TCursor**

Reports whether the TCursor is pointing to the table's last record.

Syntax

```
atLast ( ) Logical
```

Description

atLast returns True if the TCursor is pointing to the table's last record; otherwise, it returns False.

Example

The following example assumes that 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 whether the TCursor points to the last record. If **atLast** returns False, this code moves the TCursor 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

```

attach method

TCursor

Associates a TCursor with a specified table.

Syntax

1. `attach (const object UIObject) Logical`
2. `attach (const srcTCursor TCursor) Logical`
3. `attach (const tv TableView) Logical`
4. `attach (const srcHandle LongInt) Logical`

Description

attach associates a TCursor with a specified table. The data (including filters, indexes, and edit mode) comes from the underlying table. The TCursor retrieves data from committed records only (and not from records being edited or inserted).

Syntax 1 associates a TCursor with the table displayed in a UIObject named *object*.

Syntax 2 associates the TCursor with the table represented by another TCursor, named *srcTCursor*.

Syntax 3 associates the TCursor with the TableView object named *tv*.

Syntax 4 associates the TCursor with the opened cursor handle named *srcHandle*. The TCursor's data comes from the underlying cursor, pointed to by *srcHandle*, which is typically from an external DLL call. **attach** clones the cursor for use in ObjectPAL. The external DLL is responsible for opening and closing the cursor. Explicitly close the TCursor using a **TCursor.close()** in Paradox before closing the cursor in the external DLL.

attach returns True if successful; otherwise, it returns False and adds the following warning to the error stack:

"You have tried to access a document that is not open."

Example 1

The following example assumes that 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*. The form also contains a button named *findDetails* to allow your users to search the entire *LineItem* table. The **pushButton** method for *findDetails* searches for orders that include the current part number.

The following code is attached to the Var window for the *findDetails* button:

```

; findDetails::Var
Var
    lineTC TCursor ; instance of LINEITEM for searching
endVar

```

The following code 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*:

attach method

```
; findDetails::pushButton
method pushButton(var eventInfo Event)
var
    stockNum,
    orderNum    Number
    orderTC     TCursor
endVar
; Get Stock No from current LineItem record.
stockNum = LINEITEM.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)      ; 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
```

The following code is attached to the **close** method for *findDetails*:

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close() ; Close the TCursor to LineItem.
endMethod
```

Example 2

The following example is contained in a Script window. PDXTEST.DLL contains the openTable(), moveTo(), and closeTable() methods. This code opens a cursor by calling the DLL's openTable() method. A returned handle *hcur*, an ObjectPAL TCursor attaches to the DLL's cursor, and displays the TCursor's active record number. The DLL's moveTo() method is then used to change the cursor's position to record 3. **attach** is called to resynchronize ObjectPAL's TCursor with the DLL's cursor. ObjectPAL's TCursor and the DLL's cursor are then closed.

```
; describe the methods from PDXTEST.DLL that will be called
Uses PDXTEST
    openTable ( tableName CPTR) CLONG [stdcall]
    moveTo    ( pos CLONG ) [stdcall]
    closeTable() [stdcall]
endUses
method run(var eventInfo Event)
var
    tc    tcursor
    hCur LongInt
endvar

; Returns a cursor to the table
hCur = openTable( "aspace.db" )

; Attach to the open cursor, and get the record position.
; (When attaching to the open cursor, Paradox creates a clone of hCur.)

tc.attach( hCur )
view(tc.recNo())
; Move to the 3rd record by calling the moveTo method of the DLL.
```

```

; (The DLL's cursor's record position is changed, not ObjectPAL's
; TCursor.)
moveTo( 3 )

; Reattach the cursor to sync to the new cursor position.
tc.attach( hCur )
view(tc.recNo())

; Close the ObjectPAL cloned cursor before closing the handle
; in the DLL

tc.close()

; Close the handles in the DLL. The DLL is responsible for closing
; all handles opened by the DLL.
closeTable()

endMethod

```

attachToKeyViol method

TCursor

Attaches a TCursor to the original record when a key violation occurs.

Syntax

```
attachToKeyViol ( const oldTC TCursor ) Logical
```

Description

attachToKeyViol attaches a TCursor to the original record after a key violation occurs. Specify the TCursor that points to the record that caused the key violation (the new, unposted record).

This method allows you to compare conflicting records before replacing or discarding changes to an existing record. *oldTC* must already be pointing to the new (yet unposted) record.

Example

The following example uses **attachToKeyViol** a key violation occurs. The code declares two TCursors: *keyViolTC* and *originalRecTC*. The code opens *keyViolTC* for the *Orders* table and deliberately inserts a record whose key value conflicts with another record. The code then forces a key violation by posting the new record to the table. 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 code calls **updateRecord**:

```

; 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)

```

bot method

```
; 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
```

bot method

TCursor

Determines whether a command attempts to move past the table's first record.

Syntax

bot () Logical

Description

bot returns True if a command attempts to move past the table's first record; otherwise, it returns False. **bot** is reset by the next move operation.

Example

The following example moves a TCursor backwards through a table and displays a message. This code is attached to a button's **pushButton** method:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    myTable TCursor
endVar
myTable.open("sites.db")
myTable.end()                ; moves to end of table
while myTable.bot() = False   ; loop until we hit the top
    myTable.priorRecord()     ; move backwards through table
endWhile
msgInfo("The Top", "You're at the beginning.")
msgInfo("At the top?", myTable.bot()) ; displays True
myTable.nextRecord()
msgInfo("At the top?", myTable.bot()) ; displays False
endMethod
```

cancelEdit method**TCursor**

Ends Edit mode without saving changes to the active record.

Syntax

```
cancelEdit ( ) Logical
```

Description

cancelEdit ends Edit mode without saving changes to the active record. Use **cancelEdit** before committing or unlocking the record. Once you move the TCursor, changes to the record are committed.

Example

The following example is attached to the **pushButton** method for the *changeKey* button. This code associates a TCursor with the *Customer* table and attempts to change a value in a keyed field. If the record cannot be posted (e.g., because of a key violation) an error message is displayed and **cancelEdit** is called 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")
if tc.locate("Customer No", 1231) then
    tc.edit()
    tc."Customer No" = 1221    ; attempt to change key value
    if not tc.endEdit() then    ; if endEdit fails
        errorShow("Can't complete the operation.")
        tc.cancelEdit()        ; restore record and leave edit mode
        message("Record left intact.")
    else
        message("Key value changed.")
    endIf
else
    errorShow("Can't find Customer 1231")
endIf

endMethod
```

cAverage method**TCursor**

Returns the average of values in a column of fields.

Syntax

1. cAverage (const *fieldName* String) Number
2. cAverage (const *fieldNum* SmallInt) Number

Description

cAverage returns the average of values in the column of fields specified by *fieldName* or *fieldNum*. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cAverage** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a write lock on the table. If a lock cannot be placed, the method fails.

cCount method

Example

The following 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 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
```

cCount method

TCursor

Returns the number of values in a column of fields.

Syntax

1. cCount (const *fieldName* String) LongInt
2. cCount (const *fieldNum* SmallInt) LongInt

Description

cCount returns the number of values in a column of fields 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 and contains blank fields, **cCount** returns the number of nonblank values in the column of fields.

This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

cCount is especially useful for returning the number of entries used by another column function.

Example

The following example opens a TCursor for a table and 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 Number
    numRecs LongInt
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
```

close method**TCursor**

Closes a table.

Syntax

```
close ( ) Logical
```

Description

close closes a TCursor, and makes the TCursor variable unassigned. If the active record cannot be committed, **close** closes the TCursor and discards any changes to the record.

Example

The following example opens a TCursor for a table, displays information found in the table's last record and closes the TCursor. This code displays a message indicating whether the TCursor variable remains assigned when the TCursor is closed, and 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
```

cMax method**TCursor**

Returns the maximum value of a column of fields.

Syntax

1. cMax (const *fieldName* String) Number
2. cMax (const *fieldNum* SmallInt) Number

Description

cMax returns the maximum value in the column of fields specified by *fieldName* or *fieldNum*. If a field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

Example

The following example assumes that a form has a button named *getMaxBalance*, and a table frame that is bound to the *Orders* table. The **pushButton** method for *getMaxBalance* associates the table frame with a TCursor and locates the highest balance due in the *Orders* table:

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

cMin method

```
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"))
; synchronize the table frame to the TCursor
ORDERS.moveToRecord(ordTC)

endMethod
```

cMin method

TCursor

Returns the minimum value in a column of fields.

Syntax

1. cMin (const *fieldName* String) Number
2. cMin (const *fieldNum* SmallInt) Number

Description

cMin returns the minimum value in the column of fields specified by *fieldName* or *fieldNum*. If the field is numeric, this method handles blank values as specified in the **blankAsZero** setting for the session. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

Example

The following example calculates the 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
```

The following example associates a TCursor with the *GoodFund* table, then calculates the net present value for the *Expected Return* field. 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

```

cNpv method

TCursor

Returns the net present value of a column, based on a discount or interest rate.

Syntax

1. cNpv (const *fieldName* String, const *discRate* Number) Number
2. cNpv (const *fieldNum* SmallInt, const *discRate* Number) Number

Description

cNpv returns the net present value of the entries in a column of fields. The net present value calculation is based on the interest or discount rate specified by *discRate*. *discRate* is a decimal number (e.g., 12 percent is expressed as .12). This method handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails. This method respects the limits of restricted views set by **setRange** or **setGenFilter**.

This method calculates net present value using the following formula:

$$\text{cNpv} = \sum_{p=1 \text{ to } n} Vp / (1+i)^p$$

(where n = number of periods, Vp = cash flow in p th 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. 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

```

compact method

TCursor

Removes deleted records from a dBASE table.

Syntax

```
compact ( [ const regIndex Logical ] ) Logical
```

copy method

Description

compact removes deleted records from a dBASE table. Deleted records are not immediately removed from a dBASE table. Instead, they are flagged as deleted and kept in the table. The optional argument *regIndex* specifies whether to regenerate or update the indexes associated with the table. When *regIndex* is set to True, this method regenerates all indexes associated with the TCursor and frees any unused space in the indexes. If *regIndex* is set to False, indexes are not regenerated. By default, *regIndex* is set to True.

This method fails if any locks have been placed on the table, or if the table is open. If the table has maintained indexes, this method requires exclusive access; otherwise it requires a write lock.

This method returns True if successful; otherwise, it returns False.

The **compact** method defined for the TCursor type does not work with Paradox tables. To pack a Paradox table, use the **compact** method defined for the Table type.

Example

The following example removes deleted records from a dBASE table named OLDDATA.DBF. This code is attached the **pushButton** method for the *purgeTable* button:

```
; purgeTable::pushButton
method pushButton(var eventInfo Event)
var
tb Table
tc TCursor
endVar
tb.attach("OldData.dbf")
    tb.setExclusive()                ; Get exclusive rights to the table.

    tc.open(tb)                    ; Associate TCursor with OldData table.

    if tc.compact() then            ; Remove all deleted records.
tc.close()
message("Old records purged.")
else
errorShow()
endIf
endMethod
```

copy method

TCursor

Copies a table.

Syntax

1. copy (const *tableName* String) Logical
2. copy (const *tableName* Table) Logical

Description

copy copies a table to the target table named *tableName*. If *tableName* does not exist, **copy** creates it. If *tableName* already exists, **copy** overwrites it without asking for confirmation.

Throughout the retry period, this method attempts to place a write lock on the source table and a full lock on the target table. This method fails if either lock cannot be placed, or if the target table is open.

This method does not respect the limits of restricted views.

Example

The following 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
        ; If .VAL file contains only RI info, it is not copied.
        sourceTC.copy(destTb)
    endIf
endIf

endMethod
```

copyFromArray method**TCursor**

Copies data from an array to the fields of the active record.

Syntax

1. `copyFromArray (const ar Array[] AnyType) Logical`
2. `copyFromArray (const ar DynArray[] AnyType) Logical`

Description

copyFromArray copies the elements of an array or a dynamic array (*DynArray*) to the record pointed to by a *TCursor*. The *TCursor* must be in Edit mode.

Syntax 1 uses an array named *ar*. 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.

Syntax 2 uses a *DynArray* named *ar*, where each index is a field name or a field number, and the corresponding item is the field value.

This method fails if an attempt is made to copy an unassigned array element or if the structures do not match. If there are more elements in the array than fields in the record, the extra elements are ignored. Use **insertRecord** to insert a blank record before using **copyFromArray** to copy a new record into an empty table.

Example

The following example assumes that *CUSTNAME.DB* has three fields: Last Name, A20; First name, A20; and Telephone, A12. This code associates a *TCursor* with the *CustName* table, creates an array with three elements, inserts a new record in the table and uses **copyFromArray** to copy data from the array to the new record:

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

copyRecord method

```
    tc TCursor
    aa Array[3] AnyType
endVar
aa[1] = "Corel"
aa[2] = "Frank"
aa[3] = "555-1212"
if tc.open("CustName.db") then ; open table
    tc.edit() ; copyFromArray works only 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
```

copyRecord method

TCursor

Copies a record from one TCursor into another TCursor.

Syntax

```
copyRecord ( const sourceTC TCursor ) Logical
```

Description

copyRecord copies the record pointed to by one TCursor into the record pointed to by another TCursor. The following code copies a record from the *sourceTC* TCursor into the *destinationTC* TCursor:

```
destinationTC.copyRecord(sourceTC)
```

The TCursor specified by *sourceTC* does not have to be in Edit mode; however, the destination TCursor must be in Edit mode. This method fails if any field in the source record cannot be converted to the data type of the corresponding field in the destination record. This method returns True if it succeeds; otherwise, it returns False.

Note

- You cannot use **copyRecord** to copy a record into an empty table. To copy a new record into an empty table, use **insertRecord**.

Example

The following example uses a TCursor to scan the *Orders* table for sales posted within the last 10 days and copies them to the *NewOrders* table in the active directory. This code is attached to the **pushButton** method for the *getNewOrders* button:

```
; getNewOrders::pushButton
method pushButton(var eventInfo Event)
var
    ordTC,
    newOrdTC    TCursor
    tvNewOrds   TableView
endVar

ordTC.open("Orders.db")
newOrdTC.open("NewOrders.db")
newOrdTC.edit() ; copyRecord only works in Edit mode.

; Scan Orders.db table for records posted in the last ten days.
scan ordTC for ordTC."Sale Date" = (today() - 10) and
    ordTC."Sale Date" today() :
```

```

newOrdTC.insertRecord()      ; Insert a new record in NewOrdrs.db.
newOrdTC.copyRecord(ordTC)   ; Copy from Orders.db into NewOrdrs.db.
endScan
newOrdTC.endEdit()           ; End Edit mode for TCursor.

tvNewOrdrs.open("NewOrdrs.db") ; Display the table.
endMethod

```

copyToArray method

TCursor

Copies a record's fields to an array.

Syntax

1. copyToArray (var *ar* Array[] AnyType) Logical
2. copyToArray (var *ar* DynArray[] AnyType) Logical

Description

copyToArray copies a record's fields to the elements of an array specified by *ar*. You must declare the array as an AnyType type, or another type that matches each field in the table.

In Syntax 1, where *ar* is a fixed or resizable 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 resizable, it expands to hold the number of fields in the record. If the array is fixed, it holds as many fields as possible, and discards the rest.

If Syntax 2, where *ar* is a dynamic array (DynArray), index values correspond to the field names and DynArray values correspond to field values.

ar [fieldName] = fieldValue

The record number field and any display-only or calculated fields that appear in a table's Form window are not copied to the array.

Example

The following example assumes that a form has a table frame named CUSTOMER that is bound to CUSTOMER.DB. When the user attempts to delete a CUSTOMER record, this code uses **copyToArray** and **copyFromArray** to copy the record to an archive table (CUSTARC.DB). If CUSTARC.DB cannot be opened, this code informs the user and does not delete the record. The following code is attached to the built-in **action** method:

```

; CUSTOMER::action
method action(var eventInfo ActionEvent)
var
    tcOrig, tcArc TCursor
    arcRec Array[] AnyType
endVar

if eventInfo.id() = DataDeleteRecord then ; when user 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
                tcArc.edit()                 ; copyFromArray requires Edit
                tcArc.insertAfterRecord()    ; create a new record
                tcArc.copyFromArray(arcRec)   ; copy arcRec to new record
            else
                msgBox("Error", "Cannot open CustArc.db")
            endIf
        endIf
    endIf
endMethod

```

createIndex method

```
        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
```

createIndex method

TCursor

Creates an index for a table.

Syntax

```
1. createIndex ( const attrib DynArray[ ] AnyType, const fieldNames Array[ ] String )
Logical
2. createIndex ( const attrib DynArray[ ] AnyType, const fieldNums Array[ ] SmallInt )
Logical
```

Description

createIndex creates an index using attributes specified in a dynamic array (DynArray) named *attrib* and the field names (or numbers) specified in an Array named *fieldNames* (or *fieldNums*). This method is provided as an alternative to the **index** structure. It is especially useful when you don't know the index structure beforehand (e.g., when the information is supplied by the user).

Each key of the DynArray must be a string. You do not have to include all the keys to use **createIndex**. Any key you omit is assigned the corresponding default value.

The following table displays the key strings and their corresponding values:

String value	Description
MAINTAINED	If True, the index is incrementally maintained. That is, after a table is changed, only that portion of the index affected by the change is updated. If False, Paradox does not maintain the index automatically. Maintained indexes typically result in better performance. Default = False (Paradox tables only).
PRIMARY	If True, the index is a primary index. If False, it's a secondary index. Default = False (Paradox tables only).
CASEINSENSITIVE	If True, the index ignores differences in case. If False, it considers case. Default = False (Paradox tables only).
DESCENDING	If True, the index is sorted in descending order, from highest values to lowest. If False, it is sorted in ascending order. Default = False.
UNIQUE	If True, records with duplicate values in key fields are ignored. If False, duplicates are included and available.

IndexName	A name used to identify this index. No default value, unless you're creating a secondary, case-sensitive index on a single field, in which case the default value is the field name. For dBASE tables, the index name must be a valid DOS filename. If you do not specify an extension, .NDX is added automatically.
TagName	The name of the index tag associated with the index specified in indexName (dBASE tables only).

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Note

- For createIndex to work, the TCursor must be attached to a table on which setExclusive has been called.

Example 1

The following example builds a maintained secondary index for a Paradox table named CUSTOMER.DB. If the *Customer* table cannot be found or locked, this code aborts the operation:

```
method pushButton(var eventInfo Event)
var
    tbCust          Table
    stTbName         String
    tcCust          TCursor
    arFieldNames     Array[3] String
    dyAttrib         DynArray[]AnyType
endVar

stTbName           = "Customer.db"

arFieldNames[1] = "Customer No"
arFieldNames[2] = "Name"
arFieldNames[3] = "Street"

dyAttrib["PRIMARY"] = False
dyAttrib["MAINTAINED"] = True
dyAttrib["IndexName"] = "NumberNameStreet"

if isTable(stTbName) then
    tbCust.attach(stTbName)
    tbCust.setExclusive()

    if tcCust.open(tbCust) = FALSE then
        msgStop("Stop!", "Can't lock " + stTbName + " table.")
        return
    endif

    if not tcCust.createIndex(dyAttrib, arFieldNames) then
        errorShow()
    endif

; This createIndex statement has the same effect
; as the following INDEX structure:

{
INDEX "Customer.db"
MAINTAINED
ON "Customer No", "Name", "Street"
```

createIndex method

```
        TO "NumberNameStree"
    ENDINDEX

}

else
    msgStop("Stop!", "Can't find " + stTbName + " table.")
endif

endMethod
```

Example 2

The following example builds a unique index named CITYSTAT.NDX for the dBASE table named CUSTOMER.DBF:

```
; cityStateIndex::pushButton
method pushButton(var eventInfo Event)
var
    tbCust      Table
    stTbName     String
    tcCust      TCursor
    arFieldNames Array[1] String
    dyAttrib     DynArray[]AnyType
endVar

stTbName        = "Cust.dbf"

arFieldNames[1] = "CITY"

dyAttrib["UNIQUE"]      = True
dyAttrib["MAINTAINED"] = True

; A dBASE index name must be a valid DOS filename.
; If an extension is omitted, .NDX is appended automatically.

dyAttrib["IndexName"] = "City"

if isTable(stTbName) then
    tbCust.attach(stTbName)
    tbCust.setExclusive()
    if tcCust.open(tbCust) = False then
        msgStop("Stop!", "Can't lock " + stTbName + " table.")
        return
    endif

    tcCust.createIndex(dyAttrib, arFieldNames)
; This createIndex statement has the same effect
; as the following INDEX structure:
{
    INDEX "Cust.dbf"
        UNIQUE
        ON "CITY", "STATE_PROV"
        TO "CityStat"
    ENDINDEX
}

else
    msgStop("Stop!", "Can't find " + stTbName + " table.")

endif

endMethod
```


cSamStd method**TCursor**

Returns the sample standard deviation of a table's column.

Syntax

1. cSamStd (const *fieldName* String) Number
2. cSamStd (const *fieldNum* SmallInt) Number

Description

cSamStd returns the sample standard deviation for 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. **cSamStd** handles blank values as specified in the **blankAsZero** setting for the session.

The sample standard deviation calculation is based on the sample variance and uses the following formula:

$$\text{sqrt}(\text{TCursor.cVar}(\text{FieldName}) * (n/(n-1)))$$

where

(variance = TCursor.cVar(fieldName) and n = TCursor.cCount(fieldName))

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

The population standard deviation is calculated using the **setRange**.

Example

The following example calculates the sample standard deviation of two 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
```

cSamVar method**TCursor**

Returns the sample variance in a column of fields.

Syntax

1. cSamVar (const *fieldName* String) Number
2. cSamVar (const *fieldNum* SmallInt) Number

Description

cSamVar returns the sample variance of the values in a column of fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. This method handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

The sample variance is calculated using this formula:

$$TCursor.cVar(fieldName) * (n/(n-1))$$

$$(n = TCursor.cCount(fieldName))$$
Example

The following example calculates 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
    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
```

cStd method**TCursor**

Returns the standard deviation of the values in a column.

Syntax

1. cStd (const *fieldName* String) Number
2. cStd (const *fieldNum* SmallInt) Number

Description

cStd returns the population standard deviation of the values in a column of fields. The calculation is based on the variance. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. This method handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

Example

In the following example, the **pushButton** method for *thisButton* calculates the population standard deviation for two separate fields. The results are displayed 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

```

cSum method

TCursor

Returns the sum of the values in a column of fields.

Syntax

1. cSum (const *fieldName* String) Number
2. cSum (const *fieldNum* SmallInt) Number

Description

cSum returns the sum of the values in a column of fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. This method handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

Example

In the following example, the **pushButton** method for *sumOrders* calculates 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

```

currRecord method

TCursor

Reads the active record into the record buffer.

cVar method

Syntax

```
currRecord ( ) Logical
```

Description

currRecord reads the values in the active record of the underlying table into the record buffer.

currRecord cancels any unposted changes to the TCursor. This method ensures that you're using the most recent version of the record on a network.

Example

The following example is part of a system that processes concert ticket orders. This code determines which artist the customer wants to see and how many seats the customer needs.

```
; updateSeats::pushButton
method pushButton(var eventInfo Event)
  var
    tcConcert      TCursor
    siSeatsNeeded,
    siCustSeats     SmallInt
    stArtist        String
  endVar

  ; Call a custom method to find out which artist
  ; the customer wants to see.
  stArtist = getArtistName()
  tcConcert.open("concerts")
  tcConcert.locate("Artist", stArtist)

  if tcConcert.SoldOut = True then
    msgStop("Sorry", "Sold out")
    return
  else

    ; Call a custom method to find out how many seats
    ; the customer needs (this may take awhile).
    siCustSeats = getCustSeats()

    ; Meanwhile, other customers may have ordered seats for this
    ; concert, so read current values into the record buffer.
    tcConcert.currRecord()

    if tcConcert.Seats = siCustSeats then
      processOrder() ; Call a custom method to process the order.
    else
      notEnoughSeats() ; Call a custom method.
    endIf
  endIf
endMethod
```

cVar method

TCursor

Returns the variance of the values in a column of numeric fields.

Syntax

1. cVar (const *fieldName* String) Number
2. cVar (const *fieldNum* SmallInt) Number

Description

cVar returns the population variance of values in a column of numeric fields. This method respects the limits of restricted views set by **setRange** or **setGenFilter**. **cVar** handles blank values as specified in the **blankAsZero** setting for the session.

Throughout the retry period, this method attempts to place a read lock on the table. If a lock cannot be placed, the method fails.

Example

In the following example, the **pushButton** method for *thisButton* calculates the population variance deviation for two fields. The results are displayed 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
```

deletesRecord method

TCursor

Deletes the record pointed to by a TCursor.

Syntax

```
deleteRecord ( ) Logical
```

Description

deleteRecord deletes the record pointed to by a TCursor without asking for confirmation. Deleted Paradox records cannot be retrieve, but deleted dBASE records can. The table must be in Edit mode.

If the specified record is contained in a dBASE table and is locked or has already been deleted by another user, this method fails.

Example

In the following example, the **pushButton** method for the *checkIOU* button determines whether a debt has been paid. If the record has been marked as paid, 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 active record
        message(searchName + " deleted")
    endIf
endIf
```

didFlyAway method

```
    else
        sendBill()                ; run custom procedure
    endIf
else
    msgStop("Stop", "Couldn't find " + searchName)
endIf
endMethod
```

didFlyAway method

TCursor

Reports whether a key value change moved the active record to a different position in the table.

Syntax

```
didFlyAway ( ) Logical
```

Description

didFlyAway 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 only accurate if the **setFlyAwayControl** method has been set to True; otherwise, **didFlyAway** returns always False.

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) ; Enable flyaway tracking.

if tc.locate("Customer No", 120) then
    tc.edit()

    ; Change the key value so that the record
    ; changes relative position.
    tc."Customer No" = 145

    tc.unlockRecord()    ; Unlock the record.

    ; The dialog box 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

```

dmAttach method

TCursor

Associates a TCursor with a table in the data model.

Syntax

```
dmAttach ( const dmTableName String ) Logical
```

Description

dmAttach associates a TCursor with the table specified by *dmTableName*. The table must be in the data model. This method returns True if successful; otherwise, it returns False.

Example

The following example uses **dmAttach** to open a TCursor to a table in the data model. The TCursor respects the restricted view of the data model. The code uses **cSum** to gather information stored in the string variables *s1*, *s2*, and *s3*. The information is displayed in a dialog box.

```

;btnCustomerSummary :: pushButton
method pushButton(var eventInfo Event)
var
    tc    TCursor
    s1    String
    s2    String
    s3    String
endVar
tc.dmAttach("Orders.db")
s1 = string(tc.cSum("Total Invoice"))
s2 = string(tc.cSum("Amount Paid"))
s3 = string(tc.cSum("Balance Due"))

msgInfo("Customer Summary",
"Total Orders = " + s1 +
"\nTotal Paid = " + s2 +
"\nTotal Due = " + s3)
endMethod

```

dropGenFilter method

TCursor

Removes the filter criteria associated with a TCursor.

Syntax

```
dropGenFilter ( ) Logical
```

Description

dropGenFilter removes the filter criteria associated with a TCursor. Any indexes and tags remain in effect in the unfiltered TCursor.

Example 1

The following example attaches a TCursor to a table frame that is bound to the *Orders* table. This code calculates the average total invoice amount for the entire table by calling **dropGenFilter** to remove any user-defined or automatically generated filter criteria. The call to **dropGenFilter** operates on the TCursor only — it does not affect the table frame.

dropIndex method

```
; btnCalAvgInvoice::pushButton
method pushButton(var eventInfo Event)
var
    ordersTC    TCursor
    nuAvgInvoice Number
endVar
ordersTC.attach(Orders)    ; Attach to the Orders table frame.
ordersTC.dropGenFilter()    ; Remove any filters on the TCursor.

nuAvgInvoice = ordersTC.cAverage("Total Invoice")
nuAvgInvoice.view("Average Total Invoice:")
endMethod
```

Example 2

In the following example, a form contains a button named *btnCascadeDelete*. The **pushButton** method for *btnCascadeDelete* attaches a TCursor to a child table (the UIObject LINEITEM), uses **dropGenFilter** to ensure the TCursor can see all the child records, moves the TCursor to the first record, and puts the TCursor in edit mode. A **while** loop deletes all the child records and then the form is placed in edit mode and the parent record is deleted.

```
; btnCascadeDelete::pushButton
method pushButton(var eventInfo Event)
var
    tc          TCursor
    siCounter    SmallInt
endVar

tc.attach(LINEITEM)    ; Attach to detail table.
tc.dropGenFilter()    ; Drop any user set filters.
tc.home()              ; Make sure TCursor is on first record.

tc.edit()
while not tc.eot()      ; If there are any child
    tc.deleteRecord()    ; records, delete all of them.
endWhile

edit()                  ; Make sure form is in edit mode.
Order_No.deleteRecord() ; Delete the parent record.
endMethod
```

dropIndex method

TCursor

Deletes a specified index file or tag.

Syntax

1. (Paradox tables) dropIndex (const *indexName* String) Logical
2. (dBASE tables) dropIndex (const *indexName* String [, const *tagName* String]) Logical

Description

dropIndex deletes a specified index file or tag. You can't delete an index that's in use.

In a Paradox table, *indexName* is required to specify a secondary index. You can't use a TCursor to drop the primary index of a Paradox table.

In a dBASE table, you can use *indexName* to specify an .NDX file, or use *indexName* and *tagName* to specify an .MDX file and an index tag.

Note

- You must open the TCursor on a Table variable that has called the Table method **setExclusive** (before opening the table) before calling **dropIndex**.

For more information about indexes, see About keys and indexes in tables in the Paradox online Help.

Example

In the following example, the **pushButton** method for a button deletes the *CustName* tag from an .MDX file:

```
method pushButton(var eventInfo Event)
var
    tc1 TCursor
    tbl Table
endVar

if isTable("Sales.dbf") then
    tbl.attach("Sales.dbf") ; Sales.dbf is a dBASE table
    tbl.setExclusive (Yes)
    tc1.open(tbl)

    ; delete CustName tag from index2 file
    if tc1.dropIndex("index2.mdx", "CustName") then
        msgInfo("", "custname dropped")
    else
        errorShow("Could not drop index.")
    endIf
else
    msgStop("Stop!", "Could not find Sales.dbf table.")
endIf

endMethod
```

edit method**TCursor**

Places a TCursor in Edit mode.

Syntax

```
edit ( ) Logical
```

Description

edit places a TCursor in Edit mode allowing you to modify the active record. To remain in Edit mode, move off the record or use **postRecord** or **unlockRecord** to accept changes. 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 its contents 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 begin editing the table. After the new record is inserted, **endEdit** accepts changes and exits Edit mode:

```
; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    tc TCursor
    aa Array[3] AnyType
endVar
aa[1] = "Corel"
```

empty method

```
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
```

empty method

TCursor

Deletes all records from a table.

Syntax

```
empty ( ) Logical
```

Description

empty deletes all records from a table without asking for confirmation. If the TCursor is associated with a dBASE table, the records are flagged as deleted and the table is compacted (if possible). The TCursor does not have to be in Edit mode to empty records, but a write lock is required. This operation cannot be undone.

empty removes information from the table, but does not delete the table itself. Compare this method to **delete**, which does delete the table.

empty first tries to gain exclusive rights to the table. If it can't, it tries to place a write lock on the table.

If **empty** gains exclusive rights, it deletes all records in the table at once. If a write lock is placed on the table, **empty** must delete each record individually.

If **empty** gains exclusive rights to a dBASE table, all records are deleted and the table is compacted. If a write lock is placed on the table, 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.)

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 determine 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.")
    endIf
endIf
```

end method

```
    endIf
  else
    msgInfo("Error", "Can't find " + tblName + " table.")
  endIf
endMethod
```

end method

TCursor

Moves a TCursor to the table's last record.

Syntax

```
end ( ) Logical
```

Description

end sets the active record (and the record buffer) to the table's last record.

Example

The following example uses **end** to move a TCursor to the last record in the *Orders* table. The information in the last record is displayed in a dialog box.

```
; 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",
        "Outstanding balance: " + tc."Balance Due")

endMethod
```

endEdit method

TCursor

Exits Edit mode and accepts changes to the active record.

Syntax

```
endEdit ( ) Logical
```

Description

endEdit exits Edit mode and accepts changes to the active record. This method does not close the TCursor. Changes to previous records are committed or canceled as the user navigates the table.

Example

The following example creates an array and uses **copyFromArray** to copy its contents 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 begin editing the table. When 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] = "Corel"
aa[2] = "Frank"
```

enumFieldNames method

```
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
```

enumFieldNames method

TCursor

Fills an array with the table's field names.

Syntax

```
enumFieldNames ( const fieldArray Array[ ] String ) Logical
```

Description

enumFieldNames fills an array named *fieldArray* with a table's field names. If *fieldArray* is resizable, it expands to hold the field names. If *fieldArray* is fixed, it holds as many field names as possible and discards the rest. If *fieldArray* already exists, **enumFieldNames** overwrites it without asking for confirmation.

Example

In the following example, the **pushButton** method for the *btnEnumFields* button stores field names in a resizable array and 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
```

enumFieldNamesInIndex method

TCursor

Fills an array with a table index's field names.

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

enumFieldNamesInIndex 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 resizable, it

expands to hold all of the field names. If *fieldArray* is fixed, it holds as many field names as possible, and discards the rest. If *fieldArray* already exists, this method overwrites it without asking for confirmation.

In a dBASE table, you can use the optional argument *tagName* to specify an index tag within an .MDX file.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

In the following example, the **pushButton** method for the *enumIndex* button stores field names in a resizable array and 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.MDX", "byDate", fieldNames)
    ; display the index field names for byDate in DateIndx
    fieldNames.view()
else
    msgStop("Stop", "Couldn't open Sales.dbf.")
endif
endMethod
```

enumFieldStruct method

TCursor

Lists the field structure of a TCursor.

Syntax

1. enumFieldStruct (const *tableName* String) Logical
2. enumFieldStruct (*inMem* TCursor) Logical

Description

enumFieldStruct lists the field structure of a TCursor. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates a Paradox table *tableName*. If *tableName* already exists, this method overwrites it without asking for confirmation. If *tableName* is open, this method fails. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **struct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field	Type	Description
Field Name	A31	Specifies the name of field
Type	A31	Specifies the data type of field
Size	S	Specifies the size of field

enumFieldStruct method

Dec	S	Specifies the number of decimal places, or 0 if field type doesn't support decimal places
Key	AI	Specifies whether the field is a key (* = key field, blank = not key field)
_Required Value	AI	Specifies whether the field is required (T = required, N (or blank) = Not required)
_Min Value	A255	Specifies the field's minimum value
_Max Value	A255	Specifies the field's maximum value
_Default Value	A255	Specifies the field's default value
_Picture Value	A175	Specifies the field's picture
_Table Lookup	A255	Specifies the name of lookup table (including the full path if the lookup table is not in :WORK:)
_Table Lookup Type	AI	Specifies the type of lookup table 0 (or blank) = no lookup table, 1 = Current field + private 2 = All corresponding + no help 3 = Just current field + help and field 4 = All corresponding + help
_Invariant Field ID	S	Specifies the field's ordinal position in table (first field = 1, second field = 2, etc.)

Example

The following example assumes that you want a new table named *NewCust* that is similar to the *Customer* table. It also assumes that you want all of the fields in *NewCust* to be required fields. The following code uses **enumFieldStruct** to load a new table (CUSTFLDS.DB) with the field-level information from *Customer*. The code then scans *CustFlds* and modifies the field definitions so that each record describes a required field. *CustFlds* is then supplied in the **struct** clause of a **create** statement.

```
; makeNewCust::pushButton
method pushButton(var eventInfo Event)
var
    newCustTbl Table
    tc TCursor
    structName, sourceName String
endVar

structName = "CustFlds.db"
sourceName = "Customer.db"

if tc.open(sourceName) then

    tc.enumFieldStruct(structName)

    ; 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

```

enumIndexStruct method

TCursor

Lists the structure of a TCursor's secondary indexes.

Syntax

1. enumIndexStruct (const *tableName* String) Logical
2. enumIndexStruct (*inMem* TCursor) Logical

Description

enumIndexStruct lists the structure of a TCursor's secondary indexes. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **indexStruct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field	Type	Description
infoHeader	A1	Specifies whether this record is a header for (and the data it contains is shared by) subsequent consecutiverecords that have a value of N in this field
szName	A255	Specifies the index name, including path
szTagName	A31	Specifies the tag name, no path (dBASE only)
szFormat	A31	Specifies the optional index type, e.g., BTREE, HASH
bPrimary	A1	Specifies whether the index is primary

enumIndexStruct method

bUnique	AI	Specifies whether the index is unique
bDescending	AI	Specifies whether the index is descending
bMaintained	AI	Specifies whether the index is maintained
bCaseInsensitive	AI	Specifies whether the index is case-sensitive
bSubset	AI	Specifies whether the index is a subset index (dBASE only)
bExpIdx	AI	Specifies whether the index is an expression index (dBASE only)
iKeyExpType	N	Specifies the key type of index expression (dBASE only)
szKeyExp	A220	Specifies the key expression for expression index (dBASE only)
szKeyCond	A220	Specifies the subset condition for subset index (dBASE only)
FieldNo	N	Specifies the ordinal position of key field in table
FieldName	A31	Specifies the name of key field

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example assumes that you want a new table named *NewCust* that is similar to the *Customer* table. It also assumes that you don't want to borrow referential integrity or security information. The following code uses **enumFieldStruct** and **enumIndexStruct** to generate two tables (CUSTFLDS.DB and CUSTINDX.DB). *CustFlds* and *CustIdx* 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
    custTC.enumIndexStruct("CustIdx.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 "CustIdx.db"
                ENDCREATE

else
    msgStop("Error", "Can't find Customer table.")
endif
```


endMethod

enumLocks method**TCursor**

Creates a Paradox table listing the locks currently applied to a table.

Syntax

```
enumLocks ( const tableName String ) LongInt
```

Description

enumLocks creates a Paradox table specified by *tableName* that lists the number of locks on the specified table. If *tableName* already 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 also include an alias or path in *tableName*; if an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table describes the structure of *tableName* :

Field name	Field type	Description
UserName	A15	Specifies the user name
LockType	A32	Describes the type of lock (e.g., Table Write Lock)
NetSession	N	Specifies the net level session number
Session	N	Specifies the BDE session number (if the lock was placed by BDE)
RecordNumber	N	Specifies the record number (if the lock is a record lock; otherwise 0)

Example

In the following example, the built-in **pushButton** method for the *showOrdersLcks* button creates a table listing the locks currently applied to ORDERS.DB:

```
; 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
```

enumRefIntStruct method**TCursor**

Lists referential integrity information for a TCursor.

Syntax

1. enumRefIntStruct (const *tableName* String) Logical
2. enumRefIntStruct (*inMem* TCursor) Logical

Description

enumRefIntStruct lists referential integrity information for a TCursor. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. If *tableName* is open, this method fails. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **refIntStruct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field name	Type	Description
infoHeader	AI	Specifies whether the record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field
RefName	A31	Specifies the name to identify this referential integrity constraint
OtherTable	A255	Specifies the name (including path) of the other table in the referential integrity relationship
Slave	AI	Specifies whether the table is slave, not master (i.e., the table is dependent)
Modify	AI	Specifies the update rule (Y = Cascade, blank = Prohibit)
Delete	AI	Specifies the delete rule (blank = Prohibit). Paradox does not support cascading deletes for Paradox or dBASE tables.
FieldNo	N	Specifies the ordinal position of the field in this table involved in a referential Integrity relationship
aiThisTabField	A31	Specifies the name of the field in this table involved in a referential integrity relationship
Other FieldNo	N	Specifies the ordinal position of the field in the other table involved in a referential integrity relationship
aiOthTabField	A31	Specifies the name of the field in the other table involved in a referential integrity relationship

Example

The following example uses **enumRefIntStruct** to write CUSTOMER.DB referential integrity information to the *CustRef* table. 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")
tc.close()
; 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

```

enumSecStruct method

TCursor

Lists a TCursor's security information.

Syntax

1. enumSecStruct (const *tableName* String) Logical
2. enumSecStruct (*inMem* TCursor) Logical

Description

enumSecStruct lists the security information (access rights) of a TCursor. Syntax 1 creates a Paradox table; Syntax 2 stores the information in a TCursor variable.

Syntax 1 creates the Paradox table specified in *tableName*. For dBASE tables, this method lists the structure of the indexes associated with the table by the **usesIndexes** method. If *tableName* is open, this method fails. If *tableName* already exists, this method overwrites it without asking for confirmation. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory. You can supply *tableName* to the **secStruct** option in a **create** statement to borrow that table's field structure (including primary keys and validity checks) for use in the new table.

In Syntax 2, the structure information is stored in the TCursor variable *inMem* that you pass as an argument. Syntax 2 results in faster performance because the information is stored in system memory.

The following table displays the structure of the table in Syntax 1 or the TCursor in Syntax 2:

Field name	Type	Description
infoHeader	AI	Specifies whether the record is a header for (and the data it contains is shared by) subsequent consecutive records that have a value of N in this field
iSecNum	N	Specifies the number to identify security description (first description = 1)

enumSecStruct method

eprvTable	N	Specifies the table privilege value
eprvTableSym	A10	Specifies the table privilege name
iFamRights	N	Specifies the family rights value
iFamRightsSym	A10	Specifies the family rights name
szPassword	A31	Specifies the password
fldNum	N	Specifies the ordinal position of field in table
aprvFld	N	Specifies the field privilege value
aprvFldSym	A10	Specifies the field privilege name

Example

The following example creates a new table based on the security information that is associated with the *Secrets* table. This code uses **enumSecStruct** to write security information to the *SecInfo* table which is then used to create the *MySecrts* table:

```
method pushButton(var eventInfo Event)
var
    tc TCursor
    tbl Table
endVar

; Associate tc with SECRETS.DB.
tc.open("Secrets.db")
; Write security information to SECINFO.DB.
tc.enumSecStruct("SecInfo.db")

; Now create MYSECRS.DB.
; Borrow field names and types from SECRETS.DB.
; Borrow security information from SECINFO.DB.
tbl = CREATE "MySecrts.db"
        LIKE "Secrets.db"
        SECSTRUCT "SecInfo.db"
ENDCREATE
endMethod
```

Privilege values and names for enumSecStruct

The following table lists numeric values and symbolic names for table and field privileges.

Value	Name	Description
0	None	Specifies no privileges
1	ReadOnly	Specifies a read-only field or table
3	Modify	Specifies a read and modify field or table
7	Insert	Specifies insert + all of the above privileges (table only)
15	InsDel	Specifies delete + all of the above privileges (table only)
31	Full	Specifies full rights (table only)
255	Unknown	Specifies privileges unknown

Family rights values and names for enumSecStruct

The following table lists numeric values and symbolic names for family rights.

Value	Name	Description
0	NoFamRights	Specifies no family rights
1	FormRights	Specifies the right to change forms only
2	RptRights	Specifies the right to change reports only
4	ValRights	Specifies the right to change val checks only
8	SetRights	Specifies the right to change image settings
15	AllFamRights	Specifies all of the above

enumTableProperties method

TCursor

Writes the properties of a TCursor to a Paradox table.

Syntax

```
enumTableProperties ( const tableName String ) Logical
```

Description

enumTableProperties writes the properties of a table associated with a TCursor to the table specified in *tableName*. If *tableName* already exists, this method prompts the user for confirmation before overwriting the table. If *tableName* is open, this method fails. You can also include an alias or path in *tableName*. If an alias or path is not specified, Paradox creates the table in the working directory.

The following table displays the structure of *tableName*:

Field name	Field type	Description
TableName	A32	Specifies the table name only (i.e., no path, no extension)
PropertyName	A64	Specifies the property name (e.g., for Paradox and dBASE tables: Name, Type, FieldCount, LogicalRecordSize, PhysicalRecordSize, KeySize, IndexCount, ValCheckCount, ReferentialCount, BookMarkSize, StableBookMarks, OpenMode, ShareMode)
PropertyValue	A255	Specifies the corresponding property value

Example

The following example uses **enumTableProperties** to write ORDERS.DB properties to ORDPROPS.DB. If ORDPROPS.DB exists, this code asks for confirmation before overwriting the table:

```
; showTblProps::pushButton
method pushButton(var eventInfo Event)
var
    tblName, propTbl String
    tc TCursor
    tv TableView
endVar
```

eot method

```
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
```

eot method

TCursor

Determines whether a command attempts to move past the table's last record.

Syntax

```
eot ( ) Logical
```

Description

eot returns True if a command attempts to move past the table's last record; otherwise, it returns False. **eot** is reset by the next move operation.

eot (and **bot**) returns True if a command forces the TCursor to point to a nonexistent record. For example, assume that the *Customer* table has values in the first key field that range from 1,000 to 10,000. If you call **setRange** and point the TCursor 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 how **setRange** can affect **eot** and **bot**:

```
var tc TCursor endvar
tc.open("Customer.db")
; Suppose values in field 1 range from 1,000 to 10,000.
tc.setRange(1, 10) ; filter ranges from 1 to 10
; tc.eot() and tc.bot() are True at this point
```

If a call to **setGenFilter** forces the TCursor to point to a nonexistent record, **eot** and **bot** methods return True.

Example

In the following example, a **while** loop controls a TCursor's movement through the *Orders* table. When code within the loop attempts to move past 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

```

familyRights method

TCursor

Tests for a user's ability to create or modify objects in a table's family.

Syntax

```
familyRights ( const rights String ) Logical
```

Description

familyRights determines whether you can create or modify objects in a table's family. This method returns True if you have rights to the type of object specified in *rights*; otherwise, it returns False. *rights* is a single-letter string that indicates the object type to which you may have rights (e.g., F (form), R (report), S (image settings), or V (validity checks)). This method preserves the functionality required by Paradox 3.5 tables but does not apply to tables created in versions of Paradox after 3.5.

Example

The following example determines 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

```

fieldName method

TCursor

Returns the name of a field.

Syntax

```
fieldName ( const fieldNum SmallInt ) String
```

Description

fieldName returns the name of field specified by *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)

```

fieldNo method

```
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
```

fieldNo method

TCursor

Returns the position of a field in a table.

Syntax

```
fieldNo ( const fieldName String ) SmallInt
```

Description

fieldNo returns the position of the field specified by *fieldName*. Fields are numbered from left to right, beginning with 1.

Example

In the following example, code is attached to the **pushButton** method for *thisButton*. When you press *thisButton*, this code uses **fieldNo** to display the position of *Common Name* 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
```

fieldRights method

TCursor

Reports whether a user can read or modify a field in a table.

Syntax

1. fieldRights (const *fieldName* String, const *rights* String) Logical
2. fieldRights (const *fieldNum* SmallInt, const *rights* String) Logical

Description

fieldRights 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: ReadAll, ReadOnly, or None. Rights are obtained using **addPassword** (Session type). Rights cannot be acquired after the table is opened.

Example

The following example uses **fieldRights** to determine whether a TCursor has adequate field rights before modifying 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
```

fieldSize method**TCursor**

Returns the size of a field.

Syntax

1. fieldSize (const *fieldName* String) SmallInt
2. fieldSize (const *fieldNum* SmallInt) SmallInt

Description

fieldSize returns the size of a field, as defined when the table was created. The return value represents the maximum number of characters a field can contain. For example, given a field defined as Alpha20, **fieldSize** returns a value of 20. The return value can represent the maximum amount of data the field can display. For example, given a table or a Memo field **fieldSize** returns the number of characters that can be displayed.

Numeric fields in dBASE tables can specify the number of digits to display on each side of the decimal point. For example, a field defined as Number 8.2 displays up to 8 digits total, with 6 digits to the left of the decimal and 2 digits to the right. **fieldSize** returns the number of digits to the left of the decimal. To get the second part, use **fieldUnits2**.

Example

The following example uses a dynamic array to store the size of each field in the *BioLife* table and displays the contents of the dynamic array in a dialog box:

fieldType method

```
; 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
```

fieldType method

TCursor

Returns the data type of a field.

Syntax

- 1. fieldType (const *fieldName* String) String
- 2. fieldType (const *fieldNum* SmallInt) String

Description

fieldType returns the data type of a field. If the specified field is not found, this method returns "unknown." The following tables list the possible return values for Paradox and dBASE tables:

Paradox Field Type	Return Value
Alpha	ALPHA
Autoincrement	AUTOINCREMENT
BCD	BCD
Binary	BINARY
Bytes	BYTES
Date	DATE
Formatted Memo	FMTMEMO
Graphic	GRAPHIC
Logical	LOGICAL
Long Integer	LONG
Memo	MEMO
Money	MONEY

Number	NUMBER
OLE	OLE
Short	SHORT
Time	TIME
Timestamp	TIMESTAMP
dBASE Field Type	Return Value
BINARY	BINARY
CHARACTER	CHARACTER
DATE	DATE
FLOAT	FLOAT
LOGICAL	LOGICAL
MEMO	MEMO
NUMBER	NUMERIC
OLE	OLE

Example 2

The following example uses a dynamic array to store the data type of each field in the *BioLife* table and 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.fldtype(i)
    endFor
    ; now show the contents of the DynArray
    fldTypes.view(tblName + " field types.")
else
    msgStop("Sorry", "Can't open " + tblName + " table.")
endif
endMethod
```

fieldUnits2 method

TCursor

Returns the number of decimal places defined for a numeric field in a dBASE table.

fieldValue method

Syntax

1. fieldUnits2 (const *fieldName* String) SmallInt
2. fieldUnits2 (const *fieldNum* SmallInt) SmallInt

Description

fieldUnits2 returns the number of decimal places defined for a numeric field in a dBASE table. For a Paradox table or any other driver or field type that does not require field units to be specified, this method returns 0.

Numeric fields in dBASE tables can specify the number of digits to display on each side of the decimal point. For example, a field defined as Number 8.2 displays up to 8 digits total, with 6 digits to the left of the decimal and 2 digits to the right. **fieldSize** returns 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 (e.g., OLE, binary, graphic), this method returns 0.

Example

For the following example, the **pushButton** method for *thisButton* concatenates values returned from **fieldSize** and **fieldUnits2** so 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
```

fieldValue method

TCursor

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

fieldValue retrieves the value the field (*fieldName* or *fieldNum*) and assigns it to the variable *result*. This method returns True if successful; otherwise, it returns False.

You can also read the value of a specified field using dot notation. For example, the following 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

The following example assumes that a form has at least one field, named *paymentField*. When you right-click *paymentField*, the code presents a PopUpMenu listing possible values for the field. When you choose a menu item from the list, that item is added to 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 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 horizontal 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 PopUpMenu. The values that you choose is displayed in the field.

```
; 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
```

forceRefresh method**TCursor**

Forces a TCursor to point to the data in the underlying table.

Syntax

```
forceRefresh( ) Logical
```

Description

forceRefresh empties a TCursor's record buffer and refreshes it with data from the underlying table. The record position is maintained, provided the record still exists in the table. On an SQL server, a call to **forceRefresh** forces a read from the server. This is the only way to get a refresh from the server; it may be a time-consuming operation. **forceRefresh** only works on SQL tables that have a unique index.

Example

The following example opens a TCursor on the *Orders* table and executes two scan loops to perform two calculations. The first calculation returns the total quantity of orders from California. The code then calls **forceRefresh** to get the latest data from the table before executing the second scan loop. The second calculation calculates the total quantity of orders from Florida.

```
method pushButton(var eventInfo Event)
    var
        tc          TCursor
        tName,
        fName,
        fVal_1,
        fVal_2      String
        caQty,
        flQty       LongInt
    endVar

    ; initialize variables
    tName = "orders" ; assign table name
    fName = "State"  ; assign field name
    caQty = 0        ; assign CA quantity
    flQty = 0        ; assign FL quantity
    fVal_1 = "CA"    ; assign 1st field value
    fVal_2 = "FL"    ; assign 2nd field value

    tc.open(tName)

    scan tc for tc.State = fVal_1:
        caQty = caQty + tc.Qty
    endScan

    ; during the first scan, other users may
    ; change data in the underlying table

    tc.forceRefresh() ; Get latest data from table

    scan tc for tc.State = fVal_2:
        flQty = flQty + tc.Qty
    endScan

    msgInfo("CA Qty and FL Qty:",
        "CA = " + String(caQty) + "\n" + "FL = " + String(flQty))

endMethod
```

getGenFilter method

TCursor

Retrieves the filter criteria associated with a TCursor.

Syntax

1. `getGenFilter (criteria DynArray[] AnyType) Logical`
2. `getGenFilter (criteria Array[] AnyType [, fieldName Array[] AnyType]) Logical`
3. `getGenFilter (criteria String) Logical`

Description

getGenFilter retrieves the filter criteria associated with a TCursor. This method assigns values to a dynamic array (DynArray) variable in Syntax 1, or to two Array variables that you declare and include as arguments in Syntax 2.

In Syntax 1, the DynArray *criteria* lists fields and filtering conditions as follows: the index is the field name or number (depending on how it was set), and the item is the corresponding filter expression.

In Syntax 2, the Array *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to the second field, and so on).

If the arrays used in Syntax 2 are resizeable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as possible, beginning with criteria field 1. If there are more array items than fields, the remaining items are empty. If there are more fields than items, this method fills the array.

In Syntax 3, filter criteria is assigned to a String variable *criteria* that you must declare and pass as an argument.

Example

In the following example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to fill a dynamic array (DynArray) named *dyn* with a TCursor's filter criteria. The code then determines whether the current criteria filters the State/Prov field with a value of CA, and resets the filter if necessary.

```

;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)
var
    custTC      TCursor
    dyn          DynArray[] AnyType
    keysAr      Array[] AnyType
stFilterFld,
stCriteria  String
endVar
stFilterFld = "State/Prov"
stCriteria  = "CA"
custTC.open("Customer")

custTC.getGenFilter(dyn) ; Get filter info.
dyn.getKeys(keysAr)
if keysAr.contains(stFilterFld) then
if dyn[stFilterFld] = stCriteria then
return ; Filter is set correctly.
endif
else
dyn.empty() ; Set filter criteria correctly.
dyn[stFilterFld] = stCriteria
custTC.setGenFilter(dyn)

```

getIndexName method

```
endIf  
endMethod
```

getIndexName method

TCursor

Retrieves the name of a table's current index.

Syntax

1. (Paradox tables) `getIndexName (indexName String)` Logical
2. (dBASE tables) `getIndexName (indexName String [, tagName String])` Logical

Description

getIndexName retrieves the name of the current index. **getIndexName** can also retrieve the current tag for dBASE tables. This method assigns values to String variables that you must declare and provide as arguments.

For more information on indexes, see About keys and indexes in tables in the Paradox online Help.

Example

The following example retrieves and displays the name of the index associated with the *Orders* table:

```
method pushButton(var eventInfo Event)  
  var  
    ordersTC TCursor  
    indexName String  
  endVar  
  
  ordersTC.open("orders")  
  
  ; Get the index name and assign the value to the String variable indexName.  
  ordersTC.getIndexName(indexName)  
  
  if indexName.isAssigned() then  
    indexName.view("Current index")  
  else  
    msgInfo("indexName", "No value for indexName.")  
  endIf  
endMethod
```

getLanguageDriver method

TCursor

Returns the name of the table's current language.

Syntax

```
getLanguageDriver ( ) String
```

Description

getLanguageDriver returns a String value that specifies the language driver for a table.

Example

The following example displays the language driver for the *Customer* table in a dialog box:

```
; getDriver::pushButton  
method pushButton(var eventInfo Event)  
  var  
    tc TCursor  
  endVar  
  tc.open("Customer.db")
```



```
msgInfo("", tc.getLanguageDriver()) ; displays "ascii"
endMethod
```

getLanguageDriverDesc method

TCursor

Returns the name of the table's current language driver description.

Syntax

```
getLanguageDriverDesc ( ) String
```

Description

getLanguageDriverDesc returns a String value that specifies the table's language driver.

Example

The following example displays the language driver description for the *Customer* table in a dialog box:

```
; getDriverDesc::pushButton
method pushButton(var eventInfo Event)
var
    tc TCursor
endVar
tc.open("Customer.db")
msgInfo("", tc.getLanguageDriverDesc()) ; displays "Paradox ascii"
endMethod
```

getRange method

TCursor

Retrieves the values that specify a range for a TCursor.

Syntax

```
getRange ( var rangeVals Array[ ] String ) Logical
```

Description

getRange retrieves the values that specify a range for a TCursor. This method assigns values to an Array variable that you declare and include as an argument. The following table displays the array values and the corresponding range criteria:

Number of array items	Range specification
No items (empty array)	Specifies no range criteria is associated with the Table variable
One item	Specifies a value for an exact match on the first field of the index
Two items	Specifies a range for the first field of the index
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size n, specifies exact matches on the first n-2 fields of the index. The last two array items specify a range for the n-1 field of the index

If the array is resizable, this method sets the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

handle method

Example

In the following example, a button on a form is used to display the number of orders for any customer number per month. Assume that a form with the *Orders* table in its data model contains a *Customer_No* field, a *Month* field, and a button named *btnCustOrdersByMonth*. In this example a secondary index named *secCustomerMonth*, **getRange**, **getIndexName**, **switchIndex** and **setRange** is used to speed up the task.

```
;btnCustOrdersByMonth :: pushButton
method pushButton(var eventInfo Event)
var
    tc          TCursor
    nuCustomer   Number
arGet,
arSet          Array[2] AnyType
stMonth,
stActiveInd,
stDisplay     String
endVar
    nuCustomer = Customer_No.value      ;Customer field on form.
    nuCustomer.view("Customer #:")      ;Allow user to alter cust #.

    stMonth = Month.value               ;Month field on form.
    stMonth.view("Month:")              ;Allow user to alter month.

    arSet[1] = nuCustomer               ;Set array to range criteria.
arSet[2] = stMonth
    tc.attach(Customer_No)             ;Attach tc to Customer field.

    tc.getIndexName(stActiveInd)        ;Get the active index name.
if stActiveInd = "secCustomerMonth" then
    tc.getRange(arGet)                 ;Get the current range.
    if arGet arSet then                ;Compare current range.
tc.setRange(nuCustomer, stMonth, stMonth)
endif
else
;You must create a secondary index named secCustomerMonth
;for this example to work.
tc.switchIndex("secCustomerMonth")
tc.setRange(nuCustomer, stMonth, stMonth)
endif
stDisplay = String(nuCustomer) + " had "
• String(tc.nRecords()) +
" orders in " + stMonth
msgInfo("Orders in a month", stDisplay)
endMethod
```

handle method

TCursor

Returns a cursor handle for use in an external DLL call.

Syntax

```
handle ( ) LongInt
```

Description

handle returns the cursor handle for use in an external DLL call.

Example

In the following example, the `displayTable` method of the `PDXTEST.DLL` is called with the handle of the `TCursor`.

The following code appears in an ObjectPAL Editor window for the script's built-in **run** method"

```
; Define the prototype information for the displayTable method
; of PDXTEST.DLL
Uses PDXTEST
    displayTable( handle CLONG ) CLONG
endUses

method run(var eventInfo Event)
var
    tc    TCursor
    hCur LongInt
endvar

; Open the TCursor and get the handle of the opened table
tc.open( "aspace.db" )
hCur = tc.handle()

; Call the DLL's displayTable method, which displays the table's data.
; The DLL method should clone the cursor then close the cloned cursor
; after it has completed the pack.
displayTable ( hCur )
; Close the TCursor
tc.close()

endMethod
```

home method**TCursor**

Moves to a table's first record.

Syntax

```
home ( ) Logical
```

Description

home moves to a table's first record.

Example

For the following example, the **pushButton** method associates a *TCursor* with the *Orders* table and loads an array with field values in a **scan** loop. When the loop terminates, the *TCursor* is positioned in the table's last record. This code uses **home** to move the *TCursor* back to the table's first record:

```
; 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.fldVal
```

initRecord method

```
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
```

initRecord method

TCursor

Empties the record buffer.

Syntax

```
initRecord ( ) Logical
```

Description

initRecord initializes the record buffer by filling it with blanks (*not* spaces). If you have set default values for fields, **initRecord** initializes those fields with the default.

Example

See the example for **lockRecord**

insertAfterRecord method

TCursor

Inserts a record below the active record.

Syntax

```
insertAfterRecord ( [ const pointer TCursor ] ) Logical
```

Description

insertAfterRecord inserts a record below active record. This method can be used to add new records to the end of a table. The optional argument *pointer* inserts the record pointed to by a different TCursor. Omitting the argument inserts a blank record.

If the table is indexed, the record is placed in its sorted position when the data is committed; otherwise, it is inserted after the active record.

This method fails if the table is not in Edit mode, or if the active record cannot be committed (e.g., because of a key violation).

Example

The following example assumes that a form has a table frame named CUSTOMER that is bound to CUSTOMER.DB. When the user deletes a record, the built-in action method for CUSTOMER moves the record to CUSTARC.DB before deleting it 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.

This code uses 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 active 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
  message("Record not deleted.")        ; else user didn't confirm delete
endIf
else
  msgStop("Stop!", "Press F9 to edit data.") ; else form is not in Edit mode
endIf
endIf
endMethod

```

insertBeforeRecord method

TCursor

Inserts a record above the active record.

Syntax

```
insertBeforeRecord ( [ const pointer TCursor ] ) Logical
```

Description

insertBeforeRecord inserts a record above the active record. You can use the optional argument *pointer* to insert the record pointed to by another TCursor. If you omit the pointer argument, a blank record is inserted.

If the table is indexed, the record is placed in its sorted position when the data is committed; otherwise, it is inserted after the active record.

This method fails if the table is not in Edit mode, or if the active record cannot be committed (e.g., because of a key violation).

Example

The following example assumes that a form has a table frame named CUSTOMER that is bound to CUSTOMER.DB. When the user deletes a record, the built-in action method for CUSTOMER moves the record to CUSTARC.DB before deleting it 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.

This code uses 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

```

insertRecord method

```
"Delete the active 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 active record in CustArc.db
    doDefault                      ; process DataDeleteRecord now
else
    msgStop("Stop!", "Sorry, Can't archive record.")
endif
else
    message("Record not deleted.") ; else user didn't confirm delete
endif
else
    msgStop("Stop!", "Press F9 to edit data.") ; else form is not in Edit mode
endif
endif
endMethod
```

insertRecord method

TCursor

Inserts a record above the active record.

Syntax

```
insertRecord ( [ const pointer TCursor ] ) Logical
```

Description

insertRecord inserts a record into a table above the active record. You can use the optional argument *pointer* to insert the record pointed to by another TCursor. If you omit the *pointer* argument, a blank record is inserted.

If the table is indexed, the record is placed in its sorted position when the data is committed; otherwise, it is inserted after the active record.

This method fails if the table is not in Edit mode, or if the active record cannot be committed (e.g., because of a key violation).

Example

The following example assumes that a form has a table frame named CUSTOMER that is bound to CUSTOMER.DB. When the user deletes a record, the built-in action method for CUSTOMER moves the record to CUSTARC.DB before deleting it 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.

This code uses 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",
            "Delete the active 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 active record in CustArc.db
        doDefault                          ; process DataDeleteRecord now
    else
        msgStop("Stop!", "Sorry, Can't archive record.")
    endIf
    else
        message("Record not deleted.")      ; else user didn't confirm delete
    endIf
    else
        msgStop("Stop!", "Press F9 to edit data.") ; else form is not in Edit mode
    endIf
endIf
endMethod

```

instantiateView method

TCursor

Copies an in-memory TCursor to a physical table and points the TCursor to it.

Syntax

1. instantiateView (const *tableName* String) Logical
2. instantiateView (const *tableVar* Table) Logical

Description

instantiateView copies an in-memory TCursor to a physical table and points the TCursor to it. This method returns True if successful; otherwise, it returns False.

Syntax 1 creates the table using the name specified in *tableName*.

Syntax 2 associates the table with the Table variable specified in *tableVar*.

Use this method after executing a query that generates a TCursor onto a live query view.

instantiateView copies the data from the live query view to a table on disk and makes the TCursor point to it. You can use the TCursor to manipulate the table's data. The resulting table has no relationship to the underlying tables in the query.

For more information on live query views, see Live query views in the Paradox online Help.

You can also use instantiateView with TCursors created by ObjectPAL methods.

Example

The following example executes a query to a TCursor and determines whether the result is a live query view. If so, the code calls **instantiateView** to write the view to a physical table. The table is displayed in a Table window.

```

method pushButton(var eventInfo Event)
const
    kName = "salary"
endConst
var
    qbeVar      Query
    tcAnswer    TCursor
    tvAnswer    TableView
endVar

qbeVar.readFromFile(kName)
qbeVar.executeQBE(tcAnswer)

if tcAnswer.isView() then

```

isAssigned method

```
        tcAnswer.instantiateView(kName)
        tvAnswer.open(kName)
    else
        return
    endIf
endMethod
```

isAssigned method

TCursor

Reports whether a TCursor variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

isAssigned returns True if a TCursor variable has been assigned a value using open or attach; otherwise, it returns False.

Example

The following example associates a TCursor with a table, displays the last record and closes the TCursor. The code displays a message indicating whether the TCursor variable remains assigned when 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
```

isEdit method

TCursor

Reports whether a TCursor is in Edit mode.

Syntax

```
isEdit ( ) Logical
```

Description

isEdit returns True if the TCursor is in Edit mode; otherwise, it returns False. If you attach a TCursor to a display manager that is in Edit mode (e.g., a UIObject or TableView), the TCursor will be in Edit mode as well.

Example

The following example assumes that a form has a button and a table frame that is bound to the *Customer* table. The **pushButton** method for *thisButton* attaches a TCursor to the table frame and uses **isEdit** to determine whether the TCursor is in Edit mode. If the table frame is in Edit mode when the TCursor is attached, the TCursor is also 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
```

isEmpty method**TCursor**

Determines whether a table contains any records.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns True if there are no records in the table associated with the TCursor; otherwise, it returns False.

Example

In the following 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
```

isEncrypted method

```
String(tc.nRecords()) + " records")
    endIf
else
    msgStop("Sorry", "Can't open " + tblName + " table.")
endIf
endMethod
```

isEncrypted method

TCursor

Reports whether a table is password-protected.

Syntax

```
isEncrypted ( ) Logical
```

Description

isEncrypted returns True if a table is password-protected; otherwise, it returns False. You cannot open a TCursor on an encrypted table until you use **addPassword** (Session type) to present the required password. Use **tableRights** to report whether a user has access rights to the table.

Example

The following example determines 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.")
    endif
else
    msgStop("Error", "Can't open the Customer table.")
endif

endMethod
```

isInMemoryTCursor method

TCursor

Reports whether a TCursor points to a table in system memory or to a physical table.

Syntax

```
isInMemoryTCursor ( ) Logical
```

Description

isInMemoryTCursor returns True if the TCursor is associated with a table in system memory (e.g., a table generated by an ObjectPAL method that enumerates information to a TCursor); otherwise, it returns False.

By default, when you execute a query, Paradox attempts to create a live query view. Use **isInMemoryTCursor** to determine whether the query creates or an in-memory answer table. If the query creates a live query view, changes made to the TCursor affect the underlying tables. If the query creates an in-memory answer table, the underlying tables are not affected. If the query results in a live query view, **isInMemoryTCursor** returns False and **isView** returns True. You can use **wantInMemoryTCursor** to specify how to create a TCursor resulting from a query.

Example

The following example executes a query from a file and uses a **scan** loop to increase the salary of each employee by 12 percent. Because you cannot determine whether the query will create a live query view before it is run, this code calls **isInMemoryTCursor** to prevent changes from affecting the actual employee salary data:

```
method pushButton(var eventInfo Event)
    var
        qbeVar      Query
        tcAnswer    TCursor
    endVar

    ; Read the query from a file.
    qbeVar.readFile("Salary.qbe")

    ; We don't know if this query will generate a live
    ; query view, so use isInMemoryTCursor to find out.
    if qbeVar.executeQBE(tcAnswer) then

        ; If it is in memory (i.e., not live) and
        ; see the effects of a 12% raise for all employees.
        if tcAnswer.isInMemoryTCursor() then
            nuOldTotalPayroll = tcAnswer.cSum("Salary")

            tcAnswer.edit()
            scan tcAnswer :
                tcAnswer.Salary = tcAnswer.Salary * .15
            endScan
            tcAnswer.endEdit()

            nuNewTotalPayroll = tcAnswer.cSum("Salary")

            msgInfo("Before raise: " + String(nuOldTotalPayroll),
                "After raise: " + String(nuNewTotalPayroll))
        else
            ; If it is live, inform user and quit the method.
            msgStop("Live query view",
                "Edits would affect the underlying table.")
            return
        endIf
    else
        errorShow()
    endIf
endmethod
```

isOnSQLServer method**TCursor**

Reports whether a TCursor is associated with a table on a SQL server.

Syntax

```
isOnSQLServer ( ) Logical
```

Description

isOnSQLServer returns True if the TCursor is associated with a table on a SQL server; otherwise, it returns False.

isOpenOnUniqueIndex method

Example

The following example is a custom method that uses **isOnSQLServer** to determine whether a TCursor is associated with a remote table. If **isOnSQLServer** returns True, this code displays a **msgQuestion** dialog box and prompts the user to confirm the lock on the remote table:

```
method confirmRemoteLock(const tc TCursor) Logical
    if tc.isOnSQLServer() then
        ; you might not want to lock remote tables
        if msgQuestion("Lock table?",
            "Lock a remote table?") = "Yes" then
            return True
        else
            return False
        endif
    endif
endMethod
```

isOpenOnUniqueIndex method

TCursor

Reports whether a TCursor is open on a unique index.

Syntax

```
isOpenOnUniqueIndex ( ) Logical
```

Description

isOpenOnUniqueIndex returns True if a TCursor is open on a unique index; otherwise, it returns False. A unique index is an index that does not allow duplicate key values.

This method allows you to update remote tables easily. Remote operations (e.g., editing data or deleting records) may fail unless the TCursor is opened on a unique index.

Example

The following example is a custom method that calls **isOpenOnUniqueIndex** before placing the TCursor in Edit mode:

```
method editIfUniqueIndex(const tc TCursor) Logical
    if tc.isOpenOnUniqueIndex() then
        return tc.edit()
    else
        return False
    endif
endMethod
```

isRecordDeleted method

TCursor

Reports whether the active record has been deleted From a dBASE table.

Syntax

```
isRecordDeleted ( ) Logical
```

Description

isRecordDeleted reports whether the active record has been deleted. **isRecordDeleted** works only for dBASE tables because deleted Paradox records can't be displayed. This method returns True if the active record has been deleted; otherwise, it returns False.

By default, deleted records in a dBASE table are not displayed. To display deleted records in the table, call **showDeleted**; otherwise, deleted records are not visible to **isRecordDeleted**.

Example

The following example opens a TCursor for the SCORES.DBF dBASE table and uses showDeleted to display the table's deleted records. This code then attempts to locate a specific record in the table. This example uses isRecordDeleted to determine whether the record has been deleted. If it returns true, the record is undeleted using 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
```

isShared method

TCursor

Reports whether a table is currently shared with another user on the network.

Syntax

```
isShared ( ) Logical
```

Description

isShared returns True if another user has opened the table specified by a TCursor; otherwise, it returns False.

Example

In the following example, a form's built-in **open** method determines whether CUSTOMER.DB is currently 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
```

isShowDeletedOn method**TCursor**

Reports whether deleted records in a dBASE table are displayed.

Syntax

```
isShowDeletedOn ( ) Logical
```

Description

isShowDeletedOn reports whether the table pointed to by a **TCursor** displays its deleted records. Use the **showDeleted** method display deleted records and use **isShowDeletedOn** to determine states. **isShowDeletedOn** applies only to dBASE tables.

Example

The following example calls **showDeleted** to display deleted records in ORDERS.DBF if **isShowDeletedOn** returns False:

```
; 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
```

isValid method**TCursor**

Reports whether the contents of a field are valid and complete.

Syntax

1. isValid (const *fieldName* String, const *value* AnyType) Logical
2. isValid (const *fieldNum* SmallInt, const *value* AnyType) Logical

Description

isValid reports whether the value specified in value conforms with field type and validity checks for the field specified in *fieldNum* or *fieldName*. This method allows you to determine whether a new field value is valid before you attempt to post the record.

isValid returns True if *value* conforms to field type and validity checks; otherwise, it returns False.

Example

The following example uses **isValid** to determine whether a value is valid for a Date field. If the value is not valid, this code warns the user; 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 String
endVar
tryValue = "100/5/1994" ; Invalid date.
tc.open("Orders.db")
```

```

if NOT tc.isValid("Sale Date", tryValue) then
    msgStop("Error",
        String(tryValue) + " is not valid for this field.")
else
    ; this condition is never met
    tc."Sale Date" = tryValue
    tc.postRecord()
endif

endMethod

```

isView method

TCursor

Reports whether a TCursor is associated with a live query view.

Syntax

```
isView ( ) Logical
```

Description

isView returns True if the TCursor is associated with a live query view; otherwise, it returns False.

If **isView** is True, **isInMemoryTCursor** returns False.

Example

See the **instantiateView** example.

locate method

TCursor

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

locate searches a table for values that match the criteria specified in one or more field value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the first value matching *exactMatch* is found and given the current view of the records. If the TCursor is using a secondary index, **locate** finds the first record in the secondary index order.

The search begins at the top 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 active record cannot be posted (e.g., because of a key violation).

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is set to True.

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 code replaces the name with a new value and informs the user of the change:

```

; fixSpelling::pushButton
method pushButton(var eventInfo Event)
var

```

locateNext method

```
ordTC TCursor
endVar

ordTC.open("Customer.db")
; if locate finds "Proffessional Divers, Ltd." in the Name field
if ordTC.locate("Name", "Proffessional 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 \nProfessional Divers, Ltd.")
endif
ordTC.endEdit()
endMethod
```

locateNext method

TCursor

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

locateNext searches a table for values that match the criteria specified in one or more field value pairs. Specify the value to search for in *exactMatch* and the field to search in *fieldName* or *fieldNum*. This method guarantees that the first value matching *exactMatch* is found and given the current view of the records. If the TCursor is using a secondary index, **locate** finds the first record in the secondary index order.

The search begins at the top 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 active record cannot be posted (e.g., because of a key violation).

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is set to True.

Example

The following 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 **pushButton** method for *findFL* :

```
; 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

```

locateNextPattern method

TCursor

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 AnyType ) Logical
2. locateNextPattern ( [ const fieldNum SmallInt, const exactMatch AnyType ] * const
   fieldNum SmallInt, const pattern AnyType ) Logical

```

Description

locateNextPattern finds strings or sub-strings (e.g., comp in computer). The search begins with the record after the active 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 active record cannot be committed (e.g., because of a key violation). To start a search at the beginning of a table, use **locatePattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard pattern operators @ and .. in the *pattern* argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single character. Any combination of literal characters and wildcards can be used to construct a search. If **advancedWildCardsInLocate** (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

For example, the following statement examines the values in the first field of each record. If a value is anything except Corel, **locateNextPattern** returns True.

```
tc.locateNextPattern(1, [^Corel])
```

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is set to True.

Example

In the following 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 searches for records whose Name field contains Corel and whose Product field begins with Par. This code keeps track of the matches found and stores field values in a resizable array. If the method can't locate any more records

locatePattern method

that match the criteria, the results are displayed in a dialog box. The following code is attached to a button's **pushButton** method:

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
    myNames TCursor
    searchFor String
    numFound SmallInt
    productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Corel"

; this searches for records with "Corel" 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 productNames 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
```

locatePattern method

TCursor

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

locatePattern finds strings or sub-strings (e.g., comp in computer). The search always starts at the beginning of the table, but if no match is found, the TCursor returns 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 active record cannot be committed (e.g., because of a key violation). To start a search after the active record, use **locateNextPattern**. To start a search before the active record, use **locatePriorPattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard pattern operators @ and .. in the *pattern* argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single character. Any combination of literal characters and wildcards can be used to construct a search. If

advancedWildCardsInLocate (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

For example, the following statement examines values in the first field of each record. If a value is anything except Corel, **locatePattern** returns True.

```
tc.locatePattern(1, [^Corel])
```

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

To start a search from the beginning of a table, use **locateNextPattern**:

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is set to True.

Example

In the following 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 searches for records whose Name field contains Corel and whose Product field begins with Par. This code keeps track of the matches found and stores field values in a resizable array. If the method can't locate any more records that match the criteria, the results are displayed in a dialog box. The following code is attached to a button's **pushButton** method:

```
; findGoodProducts::pushButton
method pushButton(var eventInfo Event)
var
    myNames TCursor
    searchFor String
    numFound SmallInt
    productNames Array[] String
endVar
myNames.open("software.db")
searchFor = "Corel"

; this searches for records with "Corel" 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 productNames 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
```

locatePrior method

TCursor

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

locatePrior searches backwards from the active record in a table for record values that match one or more field/value pairs. Specify the search value in *exactMatch* and the search field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). This method guarantees that the previous value matching *exactMatch* 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.

The search begins with the record before the active record and moves up through the table. If a match is found, the TCursor moves to that record. This operation fails if the active record cannot be posted and unlocked (e.g., due to a key violation). If no match is found, the cursor returns to the active record. This method returns True if a successful match was made; otherwise, it returns False.

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is set to True.

Example

In the following 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
```

locatePriorPattern method

TCursor

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

locatePriorPattern finds strings or sub-strings (e.g., comp in computer). The search begins with the record before the active 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 active record cannot be committed (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search at the beginning of a table, use **locatePattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard pattern operators @ and .. in the *pattern* argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single character. Any combination of literal characters and wildcards can be used to construct a search. If **advancedWildCardsInLocate** (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

For example, the following statement examines values in first field of each record. If a value is anything except Corel, locatePriorPattern returns True.

```
tc.locatePriorPattern(1, [^Corel])
```

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

To start a search from the beginning of a table, use **locateNextPattern**.

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless ignoreCaseInLocate (Session type) is set to True.

Example

In the following 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*:

lock method

```
; 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 = "Corel"
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
```

lock method

TCursor

Places specified locks on a table.

Syntax

```
lock ( const lockType String ) Logical
```

Description

lock places locks on the TCursor. The *lockType* argument is one of the following String values, listed in order of decreasing strength and increasing concurrency.

String value	Description
Full	The current session has exclusive access to the table. Cannot be used with dBASE tables.
Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

If successful, **lock** returns True; otherwise, it returns False.

Example

The following example attaches a Table variable to *Customer*, places an exclusive lock on the table and uses **reIndex** to rebuild the *Phone_Zip* index. When the index is rebuilt, this code unlocks *Customer* so other network users can gain access to the table.

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
  tc TCursor
  pdoxTbl String
endVar
pdoxBtbl = "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

```

lockRecord method

TCursor

Puts a write lock on the active record.

Syntax

```
lockRecord ( ) Logical
```

Description

lockRecord attempts to place a write lock on the record pointed to by a TCursor (an explicit record lock). **lockRecord** returns True if successful; 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 code locks the record using **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

```

lockStatus method

TCursor

Returns the number of locks on a TCursor.

moveToRecNo method

Syntax

```
lockStatus ( lockType String ) SmallInt
```

Description

lockStatus returns the number of times you have placed a lock of type *lockType* on a TCursor. *lockType*'s value is Write, Read, or Any.

If you haven't placed any locks on the table **lockStatus** returns 0.

If you specify Any for *lockType*, **lockStatus** returns the total number of locks you've placed on the TCursor. **lockStatus** does not include locks placed by Paradox or by other users or applications.

Example

The following example uses **lockStatus** to report on locks you've placed explicitly on a TCursor. Assume a form contains a button named *thisButton* and a field object named *Balance_Due* that is bound to the Balance Due field of the *Orders* table.

```
; thisButton::pushButton
const
    kTbName = "locks"
    kStatus = "Any"
endConst

var
    tcOrders    TCursor
    tvLocks     TableView
endVar

proc displayLockInfo()
    tcOrders.enumLocks(kTbName)
    tvLocks.open(kTbName)
    tvLocks.setTitle("Locks on Orders table:")

    siNumLocks = tcOrders.lockStatus(kStatus)
    siNumLocks.view("Locks on TCursor:")
    tvLocks.close()
endProc

method pushButton(var eventInfo Event)

    ; Associate TCursor with a field object bound
    ; to the Balance Due field in the Orders table.
    ; TCursor gets locks from the UIObject.
    tcOrders.attach(Balance_Due)

    displayLockInfo() ; Table is locked, but not TCursor.

    tcOrders.lock("Write") ; Lock TCursor.

    displayLockInfo() ; Table and TCursor are locked,
                      ; but locks are different.
endmethod
```

moveToRecNo method

TCursor

Moves a TCursor to a specific record.

Syntax

```
moveToRecNo ( const recordNum LongInt ) Logical
```


Description

moveToRecNo moves to the record specified in *recordNum*. This method returns an error if *recordNum* doesn't exist. Use the **nRecords** method or examine the `NRecords` property to determine the number of records in a table. This method is recommended only for dBASE tables. If used for a Paradox table, **moveToRecNo** behaves exactly like the **moveToRecord** method.

Example

The following example uses **moveToRecNo** to move to a specified record in the dBASE table `ORDERS.DBF`. This code then displays the value of the `SALE_DATE` field for that record.

```
method pushButton(var eventInfo Event)
    var
        tcOrders    TCursor
        siRecNo      SmallInt
        daSaleDate   Date
    endVar

    tcOrders.open("orders.dbf")

    siRecNo = 0
    siRecNo.view("Enter a record number:")

    if siRecNo < 0 then
        if tcOrders.moveToRecNo(siRecNo) then
            daSaleDate = tcOrders."SALE_DATE"
            daSaleDate.view("Sale date: ")
        else
            errorShow("Invalid record number.")
        endIf
    else
        return
    endIf
endMethod
```

moveToRecord method**TCursor**

Moves a `TCursor` to a specific record in a table.

Syntax

```
moveToRecord ( const recordNum LongInt ) Logical
```

Description

moveToRecord moves a `TCursor` to the record specified in *recordNum*. This method returns an error if *recordNum* is greater than the number of records in the table. Use **nRecords** to determine how many records a table contains. This method can be very slow for dBASE tables; use **moveToRecNo** instead. This operation fails if the active record cannot be committed (e.g., because of a key violation).

Example

The following example uses **moveToRecord** to move to a specified record in the *Orders* table. This code then displays the value of the `Sale Date` field for the specified record:

```
method pushButton(var eventInfo Event)
    var
        tcOrders    TCursor
        siRecNo      SmallInt
        daSaleDate   Date
    endVar
```

nextRecord method

```
endVar

tcOrders.open("orders.db")

siRecNo = 0
siRecNo.view("Enter a record number:")

if siRecNo < 0 then
  if tcOrders.moveToRecord(siRecNo) then
    daSaleDate = tcOrders."Sale Date"
    daSaleDate.view("Sale date: ")
  else
    errorShow("Invalid record number.")
  endIf
else
  return
endIf

endMethod
```

nextRecord method

TCursor

Moves to the next record in a table.

Syntax

```
nextRecord ( ) Logical
```

Description

nextRecord moves the TCursor to the next record in the table. If the table is in Edit mode, nextRecord commits changes to the active record before moving. This operation fails if the active record cannot be committed (e.g., because of a key violation).

If you attempt to move past the end of the table, nextRecord returns False, the last record of the table becomes the active record, and eot returns True.

Example

In the following 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 dynamic array (DynArray) and displays field values in a dialog box. When **nextRecord** attempts to move past 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

```

nFields method

TCursor

Returns the number of fields in a table.

Syntax

```
nFields ( ) LongInt
```

Description

nFields returns the number of fields in the table associated with a TCursor.

Example

In the following 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

```

nKeyFields method

TCursor

Returns the number of fields in the index of a table.

Syntax

```
nKeyFields ( ) LongInt
```

Description

nKeyFields returns the number of fields in the active index of the table associated with a TCursor. Use **getIndexName** to get the name of the current index.

Example

The following example reports the number of key fields in a Paradox table:

```

; thisButton::pushButton
method pushButton(var eventInfo Event)
var
    pdoxTC TCursor

```

nRecords method

```
nkf      LongInt
pdoxTbl  String
endVar
pdoxTbl = "Orders.db"

if pdoxTC.open(pdoxTbl) then
  nkf = pdoxTC.nKeyFields() ; Key fields in the primary index
  msgInfo(pdoxTbl,
    pdoxTbl + " has " + String(nkf) + " key fields.")
else
  msgInfo("Sorry", "Can't open " + pdoxTbl + " table.")
endif

endMethod
```

nRecords method

TCursor

Returns the number of records in a table.

Syntax

```
nRecords ( ) LongInt
```

Description

nRecords returns the number of records in the table associated with a **TCursor**. This operation can take a long time for dBASE tables and large Paradox tables.

If working with a dBASE table, **nRecords** counts deleted records if **showDeleted** is turned on. Otherwise, deleted records are not counted.

Notes

- When you call **nRecords** after setting a filter, the returned value does *not* represent the number of records in the filtered set. To get that information, use **cCount**.
- When you call **nRecords** after setting a range, the returned value represents the number of records in the set defined by the range.

Example

In the following example, the **pushButton** method for *thisButton* runs a custom method. If there are more than 10,000 records in ORDERS.DB; otherwise, this 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 10,000 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
```

open method**TCursor**

Opens a TCursor on a table.

Syntax

1. `open (const tableName String [, const db DataBase] [, const indexName String]) Logical`
2. `open (const tableVar Table) Logical`

Description

open associates a TCursor with the table named in *tableName*.

In Syntax 1, *tableName* is a String and you can use arguments *db* and *indexName* to specify a database and an index. If *tableName* does not specify a filename extension, Paradox assumes the extension is .DB.

In Syntax 2, *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 1

The following example uses the Syntax 1 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
    tc1 TCursor
    samp Database
endVar

; Create the SampleTables alias for the default sample directory.
addAlias("SampleTables", "Standard", "c:\\Core1\\Paradox\\samples")

; Associate the samp Database var with SampleTables database.
samp.open("SampleTables")

; Associate tc1 to the Customer table in samp database,
; and use the NameAndState index.
if not tc1.open("Customer.db", samp, "NameAndState") then
    errorShow()
endif

endMethod
```

Example 2

The following example uses Syntax 2 to open a TCursor. The following code is attached to the **pushButton** method for *secondButton*:

```
; secondButton::pushButton
method pushButton(var eventInfo Event)
var
    tc1 TCursor
    samp DataBase
    tbl Table
endVar

; Create the SampleTables alias for the default sample directory.
addAlias("SampleTables", "Standard", "c:\\Core1\\Paradox\\samples")
```

postRecord method

```
; 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 tc1 TCursor to Customer table in samp database.
if not tc1.open(tbl) then
    errorShow()
endif

endMethod
```

postRecord method

TCursor

Posts changes to a record.

Syntax

```
postRecord ( ) Logical
```

Description

postRecord posts changes to a record immediately. The record remains locked throughout the posting process. If a key value changes in an indexed table and the record flies away, the corresponding TCursor flies with it. This method returns True if successful; otherwise, it returns False.

Example

In the following 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 and posts changes using **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
```

priorRecord method**TCursor**

Moves to the previous record in a table.

Syntax

```
priorRecord ( ) Logical
```

Description

priorRecord sets the active record to the previous record in a table. If the table is in Edit mode, **priorRecord** commits changes to the active record before moving. This method returns False if the TCursor is already at the first record. Also, the first record of the table becomes the active 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 active record cannot be committed (e.g., because of a key violation).

Example

In the following example, the **pushButton** method for *showPrevCust* uses **priorRecord** to move a TCursor back through the *Customer* table. Each time the TCursor lands on a new record, this code uses **copyToArray** to copy the record's contents to a dynamic array (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
```

qLocate method**TCursor**

Searches an indexed table for a specified field value.

recNo method

Syntax

```
qLocate ( const searchValue AnyType [ , const searchValue AnyType ] * ) Logical
```

Description

qLocate searches an indexed table for records which have key field values that exactly match the criteria specified in *searchValue*. **qLocate** searches for values in the active index (the first value corresponds to the index's first field, the second value corresponds to the index's second field, and so on).

The search always starts from the beginning of the table. If no match is found, the TCursor position is set to where it would be if there had been a match. If a match is found, the TCursor moves to that record. This method does not attempt to post the active record. The operation fails if the number of search values exceeds the number of fields in the current index.

qLocate does not clear existing record locks on the TCursor. If a lock is present, **qLocate** will fail. To prevent failure, issue an **unLockRecord** before the **qLocate** is called. This could be particularly helpful within a scan loop.

Note

- **qlocate** can be used to simulate incremental searches. If **qlocate** finds a matching record for *searchValue*, the TCursor position is set to that record. If **qlocate** fails to find a match, the TCursor position is left where it would have been had there been a match.

Example

The following example 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
```

recNo method

TCursor

Returns the record number of the active record.

Syntax

```
recNo ( ) LongInt
```

Description

recNo returns an integer representing the active 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 according to the current index.

Note

- When you call **recNo** after setting a filter, the returned value is represented by the ObjectPAL constant `peInvalidRecordNumber`.

Example

In the following example, the **pushButton** method for *thisButton* searches the *Customer* table for customers that reside in Oregon. If Oregon residents are found, this code stores record numbers in an array and displays 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
```

recordStatus method**TCursor**

Reports the status of a record.

Syntax

```
recordStatus ( const statusType String ) Logical
```

Description

recordStatus returns True or False answers to a question about the status of a record. Use the argument *statusType* to specify the status in question (i.e., is New, Locked, or Modified).

The New value means the record has just been added to the table. Locked means that an implicit or explicit lock 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

The following example determines whether the active record is locked. If the record is not locked, this code uses **lockRecord** to lock the record; otherwise this code informs the user:

reIndex method

```
; lockThisRecord::pushButton
method pushButton(var eventInfo Event)
var
    tc TCursor
endVar
tc.open("orders.db")
tc.edit()

; if the active record is NOT locked
if tc.recordStatus("Locked") = False then
    ; lock the active record
    tc.lockRecord()

    ; if record is locked, this statement will display True
    msgInfo("Record Status", "recordStatus(\"Locked\") = " +
        String(tc.recordStatus("Locked")))
else
    message("Active record is already locked.")
endif

endMethod
```

reIndex method

TCursor

Rebuilds an index or index tag that is not automatically maintained.

Syntax

```
reIndex ( const IndexName String [ , const TagName String ] ) Logical
```

Description

reIndex rebuilds an index or index tag that is not automatically maintained. In a Paradox table, use *indexName* to specify an index. In a dBASE table, use *indexName* to specify an .NDX file, or *indexName* and *tagName* to specify an index tag in an .MDX file. **reIndex** requires exclusive access to the table.

Example

The following example opens a TCursor for *Customer* (a Paradox table), gains exclusive access to the table and uses **reIndex** to rebuild the *Phone_Zip* index:

```
; reindexCust::pushButton
method pushButton(var eventInfo Event)
var
    tc TCursor
    pdoxTbl String
    tb Table
endVar
pdoxBtl = "Customer.db"

tb.attach(pdoxBtl)
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
```

reIndexAll method**TCursor**

Rebuilds all index files for a table.

Syntax

```
reIndexAll ( ) Logical
```

Description

reIndexAll rebuilds all indexes for the table associated with a TCursor. This method requires exclusive rights to rebuild a maintained index and a write lock to rebuild a non-maintained index.

reIndexAll works only with Paradox tables, because any index opened for a dBASE table is maintained automatically.

Example

The following example rebuilds all indexes for the *Customer* 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) ; Need exclusive rights for a maintained index.

if tc.open(tb) then
    tc.reIndexAll() ; Rebuild all Customer indexes.
    message("Indexes rebuilt.")
else
    msgStop("Sorry", "Can't open " + pdoxTbl + " table.")
endif
endMethod
```

seqNo method**TCursor**

Returns the record number of the active record.

Syntax

```
seqNo ( ) LongInt
```

Description

seqNo returns an integer representing the active record's position in a table. For dBASE tables, **seqNo** returns the sequential position of a record as viewed by the current index. **seqNo** and **recNo** always return the same value for Paradox tables.

Note

- If you call **seqNo** after setting a filter, the return value is represented by the ObjectPAL constant named **peInvalidRecordNumber**.

Example

The following example assumes that SCORES.DBF has three records and that the second record has been deleted. This code attaches to the **pushButton** method for *testSeqNo* and demonstrates the difference between **seqNo** and **recNo** methods:

setBatchOff method

Notes

- **setBatchOn** operates for less than two seconds. If another user attempts to update or access the current table, that user's system freezes. If **setBatchOn** is not followed by a **setBatchOff** statement, the other user's system remains frozen for up to two minutes. After two minutes, the operation that caused the user's system to freeze fails (due to a timeout error) and the user's system resumes operation.
- Other users cannot determine whether **setBatchOn** has been called. To minimize the chances of interfering with other users, call **setBatchOff** as soon as possible after calling **setBatchOn**.

Example 1

The following example assumes that a form's data model contains the *Orders* table and the *Lineitem* table linked 1:M, with *Orders* as the master table. This code deletes the records in the current detail set (the line items for the current order). In this example, *Lineitem* is a tableframe or a multi-record object that is bound to the *Lineitem* table:

```
method pushButton(var eventInfo Event)
  var
    ordersTC TCursor
  endVar

  ordersTC.attach(Lineitem) ; attach to the detail set
  ordersTC.edit()

  ordersTC.setBatchOn()
  while not ordersTC.eot()
    ordersTC.deleteRecord()
  endwhile
  ordersTC.setBatchOff()

endMethod
```

Example 2

Many applications require an autosequence number that must be incremented by each user who attempts to add a record to a table. This code serializes access to an autosequence number using **setBatchOn** and **setBatchOff**. The following example assumes that the *NumTable* table contains a single numeric field named *Sequence Number*.

In this example, each user who attempts an operation calls the custom method **GetAutoSequence**. The first user who calls the method gets the lowest sequence number. The call to **setBatchOn** holds every other user out without locking the table. Every other user who has issued a **GetAutoSequence** call gains access to the table sequentially.

```
method GetAutoSequence() LongInt
  var
    numTableTC TCursor
    SequenceVar LongInt
  endVar

  numTableTC.open("numtable.db")
  numTableTC.edit()

  numTableTC.setBatchOn()
  numTableTC."Sequence Number" = numTableTC."Sequence Number" + 1
  numTableTC.postRecord()
  SequenceVar = numTableTC."Sequence Number"
  numTableTC.setBatchOff()
```

setFieldValue method

```
        return SequenceVar  
    endMethod
```

setFieldValue method

TCursor

Assigns a value to a specified field.

Syntax

1. setFieldValue (const *fieldName* String, const *value* AnyType) Logical
2. setFieldValue (const *fieldNum* SmallInt, const *value* AnyType) Logical

Description

setFieldValue sets the value of the field specified by *fieldName* (or *fieldNum*) to *value*. This method returns True if successful; otherwise, it returns False.

You can also set the value of this field 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 change the value in the Last Bid field:

```
tcVar.setFieldValue("Last Bid", 32.25)
```

Example

In the following example, the **pushButton** method for *correctName* locates a misspelled name in the Name field and 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  
    message("Can't find " + badName)      ; can't find "Usco" in Name field  
endif  
endMethod
```

setFlyAwayControl method

TCursor

Specifies the mode for the FlyAwayControl.

Syntax

```
setFlyAwayControl ( [ const yesNo Logical ] )
```

Description

setFlyAwayControl specifies in *yesNo* whether or not the **FlyAwayControl** should be active. If **FlyAwayControl** is active, flyaway information is available to the **didFlyAway** method.

If you're working with indexed tables, the **didFlyAway**, **setFlyAwayControl**, and **unlockRecord** methods are closely related. When you call **unlockRecord**, the record is posted to the table (if no key violation exists) and moved to its sorted position. **FlyAway** occurs if the record's new position is different from its original position. If flyaway is *inactive*, the TCursor will follow the record to its new position. If flyaway is *active*, the TCursor will not follow the records, but will instead point at the new record at the current location. With **flyAwayControl** active, you can use **didFlyAway** to determine whether the record did, in fact, fly away.

If **setFlyAwayControl** is set to Yes, Paradox performs extra record-level checking for many operations. To maintain an application's speed set **setFlyAwayControl** to Yes only when the application needs flyaway information. By default, **setFlyAwayControl** is set it to No.

Example

See the **didFlyAway** example.

setGenFilter method

TCursor

Specifies conditions for including records in a TCursor.

Syntax

1. `setGenFilter ([idxName String, [tagName String,]] criteria DynArray[] AnyType) Logical`
2. `setGenFilter ([idxName String, [tagName String,]] criteria Array[] AnyType [, fieldId Array[] AnyType]) Logical`

Description

setGenFilter specifies conditions for including records in a TCursor. Records that meet all the specified conditions are included, records that don't are filtered out. Unlike **setRange**, this method does not require an indexed table. **setGenFilter** must be executed before opening a table using a TCursor.

In Syntax 1, a dynamic array (DynArray) named *criteria* specifies fields and filtering conditions. The index is the field name or number, and the item is the filter expression.

The following code specifies criteria based on the values of three fields:

```
criteriaDA[1]      = "Widget"                ; The value of the first field
                                     ; in the table is Widget.

criteriaDA["Size"] = " 4"                    ; The value of the field named
                                     ; Size is greater than 4.

criteriaDA["Cost"] = "= 10.95, 22.50" ; The value of the field named
                                     ; Cost is greater than or
                                     ; equal to 10.95 and less
                                     ; than 22.50.
```

If the DynArray is empty, all existing filter criteria are removed.

In Syntax 2, an Array named *criteria* specifies filtering conditions, and an optional Array named *fieldId* specifies field names and numbers. If you omit *fieldId*, conditions are applied to fields in the order they appear in the *criteria* array (the first condition applies to the first field, the second condition applies to

setGenFilter method

the second field, and so on). The following example specifies the same criteria as the example for Syntax 1.

```
criteriaAR[1] = "Widget"
criteriaAR[2] = " 4"
criteriaAR[3] = "= 10.95, 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"
```

If the Array is empty, all existing filter criteria are removed.

For both syntaxes, *idxName* specifies an index name (Paradox and dBASE tables) and *tagName* specifies a tag name (dBASE tables only). If you use these optional items, the index (and tag) are applied to the TCursor before the filtering criteria.

This method fails if the active record cannot be committed.

Filtering on special characters

If you are filtering on special characters, you must precede the number or literal value that can be interpreted as an operator (like `/`, `\`, `—`, `+`, `=`, etc.) with a backslash(`\`). In **setGenFilter()**, the filter criteria is put into a string and parsed to pick out numbers and operators for calculations. If the number or operator in the filter needs to be interpreted literally, it needs to be preceded by a backslash(`\`). For example to filter a table with the following records:

```
1st Base
1st Love
2nd Base
3rd Base
```

and retrieve only those that start with "1st," the filter would look like the following:

```
filter = "\\1st.."
```

One backslash for the number and another to indicate the first backslash is not an escape sequence.

Example

In this example, the built-in **run** method for a script opens a TCursor onto the *Customer* table and sets filter criteria on the *State/Prov* field to equal CA. Then a **scan** loop is used to fill a dynamic array (DynArray) named *dynView* with the customer name and phone number. Finally, a **view** dialog box displays the data.

```
;Script :: run
method run(var eventInfo Event)
    var
        tc TCursor
        dyn,
        dynView DynArray[] AnyType
    endVar

    dyn["State/Prov"] = "CA"

    tc.open("CUSTOMER.DB")
    tc.setGenFilter(dyn)

    scan tc:
        dynView[tc."Name"] = tc."Phone"
    endScan

    dynView.view()
endMethod
```


setRange method

TCursor

Specifies a range of records to associate with a Table variable. This method enhances the functionality of **setFilter**, which it replaces in this version. Code that calls **setFilter** continues to execute as before.

Syntax

1. `setRange ([const exactMatchVal AnyType] * [, const minVal AnyType, const maxVal AnyType]) Logical`
2. `setRange (rangeVals Array[] AnyType) Logical`

Description

setRange specifies conditions for including a range of records. Records that meet the conditions are included when the table is opened. **setRange** compares the criteria you specify with values in the corresponding fields of a table's index. If the table is not indexed, this method fails. If you call **setRange** without any arguments, the range criteria is reset to include the entire table.

Syntax 1 allows you to set a range based on the value of the first field of the index by specifying values in *minVal* and *maxVal*. For example, the following statement examines values in the first field of the index of each record:

```
tableVar.setRange(14, 88)
```

If a value is less than 14 or greater than 88, that record is filtered out. To specify an exact match on the first field of the index, assign the same value to *minVal* and *maxVal*. For example, the following statement filters out all values except 55:

```
tableVar.setRange(55, 55)
```

To set a range based on the values of more than one field, specify exact matches except for the last one in the list. For example, the following statement looks for exact matches on Corel and Paradox (assuming they are the first fields in the index), and values ranging from 100 to 500 (inclusive) for the third field:

```
tableVar.setRange("Corel", "Paradox", 100, 500)
```

In Syntax 2, you can pass an array of values to specify the range criteria, as listed in the following table.

Number of array items	Range specification
No items (empty array)	Resets range criteria to include the entire table
One item	Specifies a value for an exact match on the first field of the index
Two items	Specifies a range for the first field of the index
Three items	The first item specifies an exact match for the first field of the index; items 2 and 3 specify a range for the second field of the index.
More than three items	For an array of size n, specify exact matches on the first n-2 fields of the index. The last two array items specify a range for the n-1 field of the index.

Example 1

The following example assumes 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* and limits the number of records included in the *TCursor* to those with 1005 in the first key field. After the call to **setRange**, 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.

showDeleted method

```
; 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.setRange(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
```

Example 2

The following example calls **setRange** using a criteria array that contains more than three items. The following code sets a range to include orders from a person with a specific first name, middle initial, and last name, and an order quantity ranging from 100 to 500 items. This code then counts the number of records in this range and displays the value in a dialog box. This example assumes that the *PartsOrd* table is indexed on the FirstName, MiddleInitial, LastName, and Qty fields.

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
var
    tcPartsOrd TCursor
    arRangeInfo Array[5] AnyType
    nuCount Number
endVar

arRangeInfo[1] = "Frank" ; FirstName (exact match)
arRangeInfo[2] = "P." ; MiddleInitial (exact match)
arRangeInfo[3] = "Corel" ; LastName (exact match)
arRangeInfo[4] = 100 ; Minimum qty value
arRangeInfo[5] = 500 ; Maximum qty value

if tcPartsOrd.open("PartsOrd") then
    tcPartsOrd.setRange(arRangeInfo)
    nuCount = tcPartsOrd.cCount(1)
    nuCount.view("Number of big orders by Frank P. Corel:")
else
    errorShow("Can't open the table.")
endif
endMethod
```

showDeleted method

TCursor

Specifies whether to display deleted records in a dBASE table.

Syntax

```
showDeleted ( [ yesNo ] ) Logical
```

Description

showDeleted specifies whether to display deleted records in a dBASE table. You can use *yesNo* to specify Yes to display deleted records, or No if you don't want to display them. If omitted, *yesNo* is Yes by default. **showDeleted** is valid only for dBASE tables because deleted records in a Paradox table cannot be displayed.

Example

In the following example, the **pushButton** method attached to *showDeletedRecs* calls **showDeleted** to display 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
```

skip method**TCursor**

Moves forward or backward a specified number of records in a table.

Syntax

```
skip ( [ const nRecords LongInt ] ) Logical
```

Description

skip Moves forward or backward a specified number of records in a table. If **skip** attempts to move beyond the limits of the table, an error is produced, and the active record will be the first or last record of the table. This operation fails if the active record cannot be committed (e.g., 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**). A value of 0 doesn't move (**skip**(0) is the same as **currRecord**). If omitted, *nRecords* is 1 by default.

Example

The following example uses **skip** to change 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
```

sortTo method

Sorts a table.

Syntax

```

1. sortTo ( const destTable String, const numFields SmallInt, const sortFields Array[
] String, const sortOrder Array[ ] SmallInt ) Logical
2. sortTo ( const destTable Table, const numFields SmallInt, const sortFields Array[ ]
String, const sortOrder Array ) Logical

```

Description

sortTo sorts a table according to its field values, and saves the results in *destTable*.

sortFields is an array of strings or integers specifying which fields to sort. The size of the *sortFields* array is specified in *numFields*. *sortOrder* is an array of integers, where 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.

sortTo requires at least a read-only lock on the source table, and a full lock on the target table. If *destTable* already 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; use a sort structure for that.

Example

The following example sorts the *Customer* table to the CUSTSORT.DB table and opens the sorted table. If the *Customer* table cannot be write-locked, this example informs the user and aborts the operation. If the *CustSort* target table already exists, the user is prompted to continue or abort.

The following code goes in the Const window for the *sortCustButton* button:

```

; sortCustButton::Const
const
    kAscending = 0
    kDescending = 1
endConst

```

The following code is placed 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] = kAscending ; in ascending order

```

```

        sortFlds[2] = "Country"           ; then by Country
        sortOrder[2] = kDescending       ; 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, this code attempts to place a write lock on the source table (CUSTOMER.DB), asks the user if the target table exists (CUSTSORT.DB) and sorts *Customer* to *CustSort* based on the values in the *sortFlds* and *sortOrder* arrays. When CUSTSORT.DB is created or updated, 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?",
                "Replace " + destTbl + " ?") "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

```

subtract method

TCursor

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

subtract determines whether records that reside in the source table also reside in *destTableName*. If matching records are found, **subtract** deletes them from *destTableName* without asking for confirmation.

If *destTableName* is keyed, **subtract** deletes the records with keys that match the values of key fields in the source table. If *destTableName* is not keyed, **subtract** deletes the records that match any record in the source table. Whether tables are keyed or not, this method considers only fields that could be keyed (based on data type, not position). For example, numeric fields are considered, but formatted memos are not. This method requires read/write access to both tables.

Throughout the retry period, this method attempts to place a full lock on both tables. If locks cannot be placed, an error results.

switchIndex method

Note

- If the target table is not indexed, this operation can be time-consuming.

Example

In the following example, the **pushButton** method for *subtractCust* deletes records from the *Customer* table that match those in the *Answer* table:

```
; subtractCust::pushButton
method pushButton(var eventInfo Event)
var
    ansTC, custTC TCursor
endVar

if ansTC.open("PRIV: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
```

switchIndex method

TCursor

Specifies an index to use to view a table's records.

Syntax

```
1. switchIndex ( [ const indexName String ] [ , const stayOnRecord Logical ] ) Logical
2. switchIndex ( [ const indexFileName String [ , const tagName String ] ] [ , const
stayOnRecord Logical ] ) Logical
```

Description

switchIndex specifies in *indexName* an index file to use to view a table. In Syntax 1, *indexName* specifies an index to use with a Paradox table. If you omit *indexName*, the table's primary index is used.

Syntax 2 is for dBASE tables. *indexFileName* can specify an .NDX file or an .MDX file. The optional argument named *tagName* specifies an index tag in a production index (.MDX) file.

If the optional argument *stayOnRecord* is set to Yes in either syntax, this method maintains the active record after the index switch. If *stayOnRecord* is set to No (the default), the first record in the table becomes the active record.

Example

The following example assumes that *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 primary index to the NameAndState index and displays the first value in the Name field. Because 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() ; move to the first record
msgInfo("First Record", tc.Name) ; display value in Name field
tc.switchIndex( ) ; to restore primary index
{ tc.switchIndex ( " ", True) to stay on the same record. }
msgInfo("First Record", tc.Name) ; display value in Name field
endMethod
```

tableName method

TCursor

Returns the name of the table associated with a TCursor.

Syntax

```
tableName ( ) String
```

Description

tableName returns the name of the table associated with a TCursor. This method is used to pass variables to the TCursor **open** method.

Example

In the following example, the **pushButton** method for *thisButton* uses the **findFirst** and **findNext** methods from the FileSystem type to locate Paradox tables in the working directory. This code 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
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
```

tableRights method

TCursor

Specifies whether the user has the right to perform table operations.

Syntax

```
tableRights ( const rights String ) Logical
```

Description

tableRights specifies whether the user has the right to perform table operations. The following table describes *rights*:

type method

Value	Description
ReadOnly	Specifies the right to read from the table without making changes
Modify	Specifies the right to enter or change data
Insert	Specifies the right to add new records
InsDel	Specifies the right to add and delete records
Full or All	Specifies the right to perform all of the above operations

This method returns True if the user has the specified rights; otherwise, it returns False.

Example

The following 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
```

type method

TCursor

Returns a table's type.

Syntax

```
type ( ) String
```

Description

type returns the string value PARADOX or DBASE to specify the table's type.

Example

The following example removes deleted records from the *Orders* table if **type** returns DBASE. If **type** returns Paradox, a message is displayed:

```
; 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

```

unDeleteRecord method

TCursor

Undeletes the active record from a dBASE table.

Syntax

```
unDeleteRecord ( ) Logical
```

Description

unDeleteRecord undeletes the active record from a dBASE table. This operation is successful only if **showDeleted** is set to True, the active record is a deleted record, and the TCursor is in Edit mode.

Example

The following example opens a TCursor for SCORES.DBF (a dBASE table) and uses **showDeleted** to display deleted records. This code then attempts to locate a specific record in the table. **isRecordDeleted** determines whether the record has been deleted; if it has, it is undeleted using **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

```

unlock method

TCursor

Unlocks a specified table that is pointed to by TCursor.

Syntax

```
unlock ( const lockType String ) Logical
```

Description

unlock attempts to remove locks explicitly placed on the table pointed to by a TCursor. *lockType* is one of the following String values, listed in order of decreasing strength and increasing concurrency:

String value	Description
Full	The current session has exclusive access to the table. No other session can open the table. Cannot be used with dBASE tables.

unLockRecord method

Write	The current session can write to and read from the table. No other session can place a write lock or a read lock on the table.
Read	The current session can read from the table. No other session can place a write lock, full lock, or exclusive lock on the table.

unlock removes locks that have been explicitly placed by a particular user or application using **lock**. **unlock** has no effect on locks placed automatically by Paradox. To ensure maximum concurrent availability of tables unlock a table that has been explicitly locked as soon as the lock is no longer needed. If you lock a table twice, you must unlock it twice. You can use **lockStatus** (defined for the TCursor and UIObject types) to determine how many explicit locks you have placed on a table. If you try to unlock a table that isn't locked or cannot be unlocked, **unlock** returns False .

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 and 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
pdoxBtl = "Customer.db"

if tc.open(pdoxBtl) 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
```

unLockRecord method

TCursor

Unlocks the active record.

Syntax

unLockRecord () Logical

Description

unLockRecord unlocks the active record. If you attempt to unlock a record that isn't locked, you'll get an error. This operation fails if the active record cannot be committed (e.g., because of a key violation).

If the table containing the record is indexed, the record is posted to the table and moved to its sorted position. Record fly away occurs if the record's new position is different from its original position.

By default, the TCursor will follow the record to its new location. This default can be changed using the **setFlyAwayControl** method. If the **setFlyAwayControl** has been set to true, the **didFlyAway** method can be called to determine whether the record did fly away.

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 and unlocks the record using **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 active record
        tc.Name = "Unisco"       ; change field value
        tc.unlockRecord()        ; unlock active 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
```

update method**TCursor**

Assigns values to fields in the active record of a TCursor.

Syntax

1. **update** (const *fieldName* String, const *fieldValue* AnyType
[, const *fieldName* String, const *fieldValue* AnyType] *) Logical
2. **update** (const *fieldNum* SmallInt, const *fieldValue* AnyType
[, const *fieldNum* SmallInt, const *fieldValue* AnyType] *) Logical

Description

update assigns values to one or more fields in the active record of a TCursor. **update** allows you to update an entire record using a single statement instead of assigning field values one at a time. Use *fieldName* (Syntax 1) or *fieldNum* (Syntax 2) to specify fields. Use *fieldValue* in Syntax 2 to specify the new field value.

You can also combine field names and field numbers in the same update statement. Performance improves if you use field numbers instead of field names.

Example

The following example uses **update** to set the values of three fields using one statement.

First, the following code assigns values to the *PartNum*, *PartName*, and *Cost* fields of the *Parts* table without using *update*:

```
var
    partsTC TCursor
    partNumID SmallInt
endVar

partsTC.open("parts")
partNumID = partsTC.fieldNo("PartNum")
```

updateRecord method

```
if partsTC.locate("PartName", "Widget") then
  partsTC.edit()

  partsTC.(partNumID) = "G01"
  partsTC.PartName    = "Gadget"
  partsTC.Cost        = 2.50

  partsTC.endEdit()
endIf
```

The following code calls **update** to accomplish the same thing:

```
var
  partsTC    TCursor
  partNumID  SmallInt
endVar

partsTC.open("parts")
partNumID = partsTC.fieldNo("PartNum")

if partsTC.locate("PartName", "Widget") then
  partsTC.edit()

  partsTC.update(partNumID, "G01", "PartName", "Gadget", "Cost", 2.50)

  partsTC.endEdit()
endIf
```

updateRecord method

TCursor

Updates an existing record with data from a new record when a key violation exists.

Syntax

```
updateRecord ( [ const moveTo Logical ] ) Logical
```

Description

updateRecord overwrites an existing record with values from an unposted, new record when a key violation exists. The record is posted to the table. If an optional argument *moveTo* is False, the TCursor points to the record after it is posted to the table; if True, the TCursor points to the record following the position of the original record.

If no key violation exists, this method behaves like **unlockRecord**.

Example

See the **attachToKeyViol** example.

TextStream type

A TextStream is a sequence of characters read from or written to a text file. TextStreams contain ANSI characters only—formatting information such as font, alignment, and margins is not included. TextStreams also contains non-printing characters (e.g., carriage returns and line feeds (CR/LF)).

Paradox maintains a file position cursor that behaves like an insertion point cursor in a word processor. The cursor 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).

The TextStream type includes several derived methods from the AnyType type.

Methods for the TextStream type

AnyType	←	TextStream
blank		advMatch
dataType		close
isAssigned		commit
isBlank		create
isFixedType		end
unAssign		eof
		home
		open
		position
		readChars
		readLine
		setPosition
		size
		writeLine
		writeString

advMatch method

TextStream

Searches for a pattern of characters in a text file.

Syntax

```
advMatch ( var startIndex LongInt, var endIndex LongInt, const pattern String )
Logical
```

Description

advMatch searches a text file for a pattern of characters specified by *pattern*. If *startIndex* is assigned a value, the search begins at the *startIndex* position; otherwise, the search begins 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. By default, this method is case sensitive but you can use the String procedure **ignoreCaseInStringCompares** to change the case behavior.

If you supply *pattern* from within a method, you must 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 the compiler and **advMatch** understand backslashes differently. When the compiler sees a string with an embedded escape sequence (e.g., 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**. For more information, 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 backslash, and one backslash and plus symbol (`\+`) yields a literal character.

To specify *pattern*, use a string with the following optional symbols:

Symbol	Matches
\	Includes special characters as regular characters (e.g., \t for Tab). Use two backslashes in quoted strings.
[]	Match the enclosed set. (e.g., [aeiou0-9] match a, e, i, o, u, and 0 through 9)
[^]	Does not match the enclosed set. (e.g., [^ aeiou0-9] matches anything except a, e, i, o, u, and 0 through 9)
()	Specifies grouping
^	Specifies the beginning of string
\$	Specifies the end of string
..	Matches anything
*	Specifies zero or more of the preceding character or expression
+	Specifies one or more of the preceding character or expression
?	Specifies none or one of the preceding character or expression
	Specifies OR operation

Example

The following example assumes that a file named PDXQUOTE.TXT exists in the 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. If the specified 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)
        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
```

```
endIf
endMethod
```

close method

TextStream

Closes a text file.

Syntax

```
close ( ) Logical
```

Description

close closes a text file and writes the contents of all text buffers to a disk. **close** also ends the association between a TextStream variable and the underlying text file.

Example

The following example declares a TextStream variable named *ts*, and calls **open** to associate *ts* with the text file named PDXQUOTE.TXT. The code 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
```

commit method

TextStream

Writes the contents of the text buffer to a disk.

Syntax

```
commit ( )
```

Description

commit empties the text buffer and writes the contents to a disk. The file remains open and the cursor position does not change.

Example

In the following example, the *createText* button creates a new file named MYTEXT.TXT, adds a line to it, commits the current version of the TextStream and 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()
```

create method

```
msgInfo("TextStream position is now", ts.position()) ; still 21
ts.close()
endMethod
```

create method

TextStream

Creates a text file for reading and writing.

Syntax

```
create ( const fileName String ) Logical
```

Description

create creates the text file specified by *fileName* and opens it for reading and writing. If *fileName* already exists, **create** overwrites it without asking for confirmation. You can specify where to create the file using a full DOS path or an alias. If you don't specify a path or alias, Paradox creates the file in the working directory.

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 statements are equivalent:

```
ts.create("newText.txt")
ts.open("newText.txt", "NW")
```

Example

In the following example, 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 **findFirst** determines whether a file named RICK.TXT exists. If it doesn't exist, a custom procedure named **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()
; Create a file, open for writing and reading
ts.create(":PRIV:rick.txt")
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
```



```

endIf
endMethod

```

end method

TextStream

Sets the cursor to the end in a text file.

Syntax

```
end ( )
```

Description

end sets the cursor to the last character in a text file.

Example

The following example assumes that a file named PDXQUOTE.TXT is stored in the private 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 following code is attached to the built-in **newValue** method of a field object. The field object displays two radio buttons with the values Overwrite and Append. Choose one overwrite the file or append information to the end of the file. If you choose Overwrite, the call to **home** moves the cursor to position 1. If you choose Append, the call to **end** moves the cursor to the end of the 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

```

eof method**TextStream**

Determines whether the cursor attempts to move past the end of a text file.

Syntax

```
eof ( ) Logical
```

Description

eof returns True if the cursor attempts to move past the end of a text file; otherwise, it returns False.

Example

The following example assumes that a file named PDXQUOTE.TXT resides in the private 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 them in a dialog box. **eof** displays a dialog box telling 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
```

home method**TextStream**

Sets the cursor to the beginning of a text file.

Syntax

```
home ( )
```

Description

home sets the cursor to the first character of a text file.

Example

See the **end** example.

isAssigned method**TextStream**

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just TextStream.

Example

The following 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 is attached 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
```

open method**TextStream**

Opens a text file in a specified mode.

Syntax

```
open ( const fileName String, const mode String ) Logical
```

Description

open opens a text file named *fileName* in the mode specified by *mode*. **open** then and associates a FileSystem variable with the underlying file. Mode specifications are case-insensitive. The following table displays mode specifications:

Mode specification	Description
a	Append and read
r	Read only
w	Write and read
nw	New file (write and read)

If the file exists, the nw mode overwrites the file without asking for confirmation.

Note

- The following statements are equivalent:

```
ts.open("new.txt", "NW")
ts.create("new.txt")
```

If you open a file in any r, w, or nw modes, the cursor moves to the beginning of the file.

position method

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 searches for the file in the working directory.

This method returns True if successful; otherwise, it returns False.

Example 1

The following 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
```

Example 2

The following example declares two TextStream variables (*ts1* and *ts2*) and calls open to associate each of them with a text file named NEWTEXT.TXT. Both variables have equal rights to the file, and Paradox maintains separate cursors for each variable. As statements are written to the file, messages display the cursor position for each variable.

```
; openStreams::pushButton
method pushButton(var eventInfo Event)
var
    ts1, ts2 TextStream
    firstLine String
    allLines Array[] String
endvar
ts1.open("newText.txt", "nw") ; open a new file read/write
ts1.writeLine("Written by ts1.")
ts1.writeLine("This is line 2.")
msgInfo("Text stream one", ts1.position()) ; displays 35
ts1.commit() ; write it out to disk, so that
; ts2 will get most current version

ts2.open("newText.txt", "w") ; open existing file read/write
msgInfo("Text stream one", ts1.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 1

ts2.writeLine("Written by ts2.")
msgInfo("Text stream one", ts1.position()) ; displays 35
msgInfo("Text stream two", ts2.position()) ; displays 18

ts1.home()
ts1.readLine(allLines) ; reads all lines into an array
allLines.view("ts1") ; displays:
; Written by ts1.
; This is line 2.
; ts1 does not reflect changes made by ts2
; unless ts1 is closed and reopened.
endMethod
```

position method

TextStream

Returns the cursor's position in a text file.

Syntax

```
position ( ) LongInt
```

Description

position returns an integer representing the cursor's position in a text file. **position** counts both printing and non-printing characters, beginning with 1 (not with 0).

It may be helpful to think of **position** as returning the number of the next character in the file.

Example

The following example creates a new text file and calls **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)
```

readChars method**TextStream**

Reads a specified number of characters in a text file.

Syntax

```
readChars ( var string String, const nChars SmallInt ) Logical
```

Description

readChars reads the number of characters specified in *nChars* and stores them in string. **readChars** begins at the current cursor position and returns True if successful; otherwise, it returns False.

Example

The following example assumes that a file named PDXQUOTE.TXT resides in the 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 following code calls **readChars** to read the first 100 characters in 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
```

readLine method**TextStream**

Reads the characters in a line of text.

readLine method

Syntax

1. `readLine (var value String)` Logical
2. `readLine (var stringArray Array[] String)` Logical

Description

readLine reads the characters in a line of text until it encounters a CR/LF pair (or 32,767 characters have been read). **readLine** then moves the cursor to the first position after the CR/LF pair (or after the 32,767th character). **readLine** begins reading from the current cursor position. This method returns True if successful; otherwise, it returns False.

Syntax 1 stores a single line in *value* (the CR/LF pair is not stored).

Syntax 2 stores the entire file in *stringArray*. *stringArray* is a resizeable array of strings and each array item stores one line from the file (the CR/LF pair is not stored).

Example 1

The following example creates a two-line text file and calls **readLine** to read the first line into a String variable. **readLine** reads four characters in the first line, skips over the CR/LF characters, and sets the cursor.

```
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(1000)
message(ts.position()) ; displays 7 (skips over CR/LF)
sleep(1000)
endMethod
```

Example 2

The following example creates a three-line text file and calls **readLine** to read the entire file into an array. The array is displayed in a dialog box.

```
method pushButton(var eventInfo Event)
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 cursor to the beginning of the file

letter.readLine(allLines)
allLines.view("Entire letter") ; displays the entire letter
letter.close()
endMethod
```

setPosition method**TextStream**

Sets the cursor position in a text file.

Syntax

```
setPosition ( const offset LongInt )
```

Description

setPosition sets the cursor position in a text file. *offset* specifies the cursor's position from the beginning of a text file. CR/LF characters are considered part of the file and can be overwritten.

Example 1

In the following example, the *showPositions* button writes a line to a new text file, MEMO.TXT. The code then moves back to the fourth character, overwrites that character with the number 4, 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
```

Example 2

The following example shows what happens when you attempt to move the cursor 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
```

size method**TextStream**

Returns the number of characters in a text file.

Syntax

```
size ( ) LongInt
```

writeLine method

Description

size returns the number of characters in a text file, including non-printing characters (e.g., carriage returns and line feeds).

Example

The following example creates a `TextStream`, writes a line to it and displays the file's size.

```
; 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
```

writeLine method

TextStream

Writes a string to a text file.

Syntax

```
writeLine ( const value AnyType [ , const value AnyType ] * ) Logical
```

Description

`writeLine` writes a comma-separated list of values to a text file and appends a CR/LF character pair. Compare this method to `writeString`, which doesn't append a CR/LF pair.

Note

- If the cursor position is in the middle of the file, the current line will be overwritten with *value* and the following line will be cleared.

Example

See the create example.

writeString method

TextStream

Writes a character string to a text file.

Syntax

```
writeString ( const value AnyType, [ , const value AnyType ] * ) Logical
```

Description

writeString writes a comma-separated list of *values* to a text file, but does not append a CR/LF pair. Compare this method to **writeLine**, which does append a CR/LF pair.

Note

- If the cursor position is in the middle of the file, the current line will be overwritten with *value* for as many characters as there are in *value*.

Example

The following example assigns strings to the variables *lo* and *hi* and 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 characters, Sell High. = 10 characters. 10 + 9 = 19.
myText.close()
endMethod
```

Time type

Time variables store times in hour-minute-second-millisecond format. The following characters can be used as separators: blank, tab, space, comma (,), hyphen (—), slash (/), period (.), colon (:), and semicolon (;). Time values must be enclosed in quotation marks.

Time values must be explicitly declared. For example, the following statements assign a time of 10 minutes and 40 seconds past eleven o'clock in the morning to the Time variable ti :

```
var ti Time endVar
ti = Time("11:10:40 am")
```

Valid time values are determined by the current Paradox time format. If the current time format is set to a 12-hour format (e.g., hh:mm:ss), Time type methods consider hh:mm:ss to be a valid time format. Use **formatSetTimeDefault** procedure defined for the System type to set Paradox time formats with ObjectPAL.

The Time type includes several derived methods from the DateTime and AnyType types.

Methods for the Time type

AnyType	←	DateTime	←	Time
blank		hour		time
dataType		milliSec		
isAssigned		minute		
isBlank		second		
isFixedType				
view				

time procedure

Time

Casts a value as a time or returns the current time.

Syntax

```
time ( [ const value AnyType ] ) Time
```

time procedure

Description

time casts *value* as a time or returns the current time according to the system clock. *value*, if specified, must match the current Paradox time format. For more information, see to the System type procedure **formatSetTimeDefault**.

Example 1

The following 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)
```

Example 2

The following 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
```

Toolbar type

The Toolbar type contains methods that create, delete, manipulate, and modify Toolbars. The Toolbar type includes several derived methods from the AnyType type.

Methods for the Toolbar type

AnyType	←	Toolbar
blank		addButton
dataType		attach
isAssigned		create
isBlank		empty
isFixedType		enumToolbarNames
unAssign		getPosition
		getState
		hide
		isVisible
		remove
		removeButton
		setPosition
		setState
		show
		unAttach

Note

Although `isAppBarVisible` and `showApplicationBar` are related to toolbars, they are methods of the `System` type.

addButton method**Toolbar**

Adds a button to a Toolbar.

Syntax

```
1. addButton ( const idCluster SmallInt, const buttonType SmallInt, const idCommand
SmallInt, const grBmp Graphic, const buttonText String [ , const tooltip String ] )
Logical
2. addButton ( const idCluster SmallInt, const buttonType SmallInt, const idCommand
SmallInt, const idBmp SmallInt, const buttonText String [ , const tooltip String ] )
Logical
```

Description

addButton adds a button to a Toolbar. The new button's position is appended to the existing toolbar. The first parameter is ignored but still must be specified by a Cluster Identifier named *idCluster*. *idCluster* is an integer that ranges from 0 to 12. The type of button added is specified by *buttonType*. Button types include pushbutton, radio button, toggle button, and repeat button. When the button is pressed, the menu command that is sent is specified by *idCommand*. The text that appears below the button icon is defined by *buttonText* and the tooltip text, which appears in the popup window when you hold your mouse above the button is defined by the optional parameter *tooltip*.

If *tooltip* is omitted, only the *buttonText* is displayed in the popup window.

Syntax 1 adds a button to the Toolbar using a graphic bitmap (*grBmp*) to specify the button's image on the Toolbar. This allows you to use a user-defined bitmap file or a bitmap object of a graphic type stored in a table.

Syntax 2 adds a button to the Toolbar using a bitmap constant. The bitmap constant specifies the button's image on the Toolbar. This method allows you to create a button using any of the defined Toolbar button bitmaps in the system resource.

The only item that can be added to a Toolbar is a button.

addButton returns `True` if the button is successfully created.

Example

The following example creates a Toolbar named `Edit` and adds three buttons to the Toolbar using defined Paradox bitmap constants:

```
method pushButton (var eventInfo Event)
var
    tb Toolbar
endvar

; Create a Toolbar named "Edit" with 3 buttons: Cut, Copy, Paste
if tb.create("Edit") then
    tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditCut, BitmapEditCut, "Cut")

    tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditCopy, BitmapEditCopy, "Copy")

    tb.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditPaste, BitmapEditPaste, "Paste")
```

attach method

```
endif  
endMethod
```

The following example creates a Toolbar named File and adds three buttons using Paradox constants. A fourth button is added using a custom graphic object.

```
method pushButton (var eventInfo Event)  
var  
    tb Toolbar  
    gr graphic  
endvar  
  
if tb.create("File") then  
    tb.addButton(ToolbarFileCluster, ToolbarButtonPush,  
        MenuTableOpen, BitmapOpenTable, "Open Table")  
  
    tb.addButton(ToolbarFileCluster, ToolbarButtonPush,  
        MenuFormOpen, BitmapOpenForm, "Open Form")  
  
    tb.addButton(ToolbarFileCluster, ToolbarButtonPush,  
        MenuReportOpen, BitmapOpenReport, "Open Report")  
  
    ; Add a button with a custom bitmap (pick a valid name)  
    gr.readFromFile("Alias.bmp")  
    tb.addButton(ToolbarModeCluster, ToolbarButtonPush,  
        MenuFileAliases, gr, "Alias")  
endif  
endMethod
```

attach method

Toolbar

Binds a Toolbar type to an existing Toolbar.

Syntax

```
attach ( const toolbarName String ) Logical
```

Description

attach binds a Toolbar type to an existing Toolbar using the name specified in *toolbarName*. The reserved name Standard can be used to attach to the Paradox Toolbar.

You can access a Toolbar by attaching to an existing toolbar or creating a new one.

Example

The following example attaches a Toolbar named MyToolbar. This code assumes that the Toolbar already exists:

```
method pushButton (var eventInfo Event)  
var  
    tbar Toolbar  
endvar  
  
if tbar.attach("Standard") then  
    msginfo("Attach", "Successful")  
else  
    msginfo("Attach", "Failed")  
endif  
endMethod
```

create method**Toolbar**

Creates a Toolbar.

Syntax

```
create ( const toolbarName String ) Logical
```

Description

create creates a Toolbar specified by *toolbarName*. *toolbarName* is used in the caption when the Toolbar is floating. The name cannot be Standard, which is reserved for the Paradox Toolbar.

You can access a Toolbar by attaching to an existing toolbar or creating a new one.

Example

Test is created using two Toolbars created with the **create** method (the Edit and File Toolbars).

```
method pushButton (var eventInfo Event)
var
    tbEdit Toolbar
    tbFile Toolbar
endvar

; Create a Toolbar named "Edit" with 3 buttons: Cut, Copy, and Paste
if tbEdit.create("Edit") then
    tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditCut, BitmapEditCut, "Cut")

    tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditCopy, BitmapEditCopy, "Copy")

    tbEdit.addButton(ToolbarEditCluster, ToolbarButtonPush,
        MenuEditPaste, BitmapEditPaste, "Paste")
endif

; Create another toolbar "File"
if tbFile.create("File") then
    tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
        MenuTableOpen, BitmapOpenTable, "Open Table")

    tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
        MenuFormOpen, BitmapOpenForm, "Open Form")

    tbFile.addButton(ToolbarFileCluster, ToolbarButtonPush,
        MenuReportOpen, BitmapOpenReport, "Open Report")
endif
endMethod
```

empty method**Toolbar**

Removes the existing buttons from the Toolbar.

Syntax

```
empty ( ) Logical
```

Description

empty removes the existing buttons from the attached Toolbar. **empty** returns True if the Toolbar is successfully emptied.

enumToolbarNames method

Example

The following example attaches a Toolbar named MyToolbar and removes the Toolbar's buttons using **empty**. If the attach fails, an "Unable to attach" message appears.

```
method pushButton (var eventInfo Event)
var
    tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.empty()
else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

enumToolbarNames method

Toolbar

Returns a list of all the existing toolbars.

Syntax

```
enumToolbarNames (var toolbarNames Array[] String)
```

Description

enumToolbarNames creates an array listing all of the toolbars which currently exist in the application. *toolbarNames* is a resizable array that you must declare before calling this method. If *toolbarNames* already exists, this method overwrites it without asking for permission.

Example

The following example fills a resizable array *toolbarlist* with the names of all the toolbars which currently exist in the application, and uses **view** to display the array's contents.

```
method run (var eventInfo Event)
var
    toolbarlist      array[]string
endvar

enumToolbarNames(toolbarlist)
toolbarlist.view()

endMethod
```

getPosition method

Toolbar

Returns the position of a floating Toolbar.

Syntax

```
getPosition ( var x LongInt, var y LongInt ) Logical
```

Description

getPosition returns the position of a floating Toolbar. The Toolbar's coordinates are specified in pixels, relative to the top-left corner of the screen.

Example

The following example displays the X and Y coordinates of the attached Toolbar named MyToolbar:

```

method pushButton (var eventInfo Event)
var
    liX, liY    LongInt
    tbar        Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.getPosition(liX, liY)
    liX.view("X coordinate")
    liY.view("Y coordinate")
endif
endMethod

```

getState method

Toolbar

Retrieves the Toolbar's current state.

Syntax

```
getState ( ) smallInt
```

Description

getState retrieves the Toolbar's current state. There are six Toolbar states:

- ToolbarStateTop: docked at the top of the window
- ToolbarStateLeft: docked at the left of the window
- ToolbarStateRight: docked on the right side of the window
- ToolbarStateBottom: docked at the bottom of the window
- ToolbarStateFloatHorizontal: floating horizontally
- ToolbarStateFloatVertical: floating vertically

Example

The following example displays the current state of a Toolbar named MyToolbar. If this code cannot attach to MyToolbar, an "Unable to attach" message appears.

```

method pushButton (var eventInfo Event)
var
    tbar        Toolbar
endvar

if tbar.attach("MyToolbar") then
    msgInfo("MyToolbar", "Current State: " + String(tbar.getState()))
else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod

```

hide method

Toolbar

Hides a Toolbar.

Syntax

```
hide ( ) Logical
```

isVisible method

Description

hide hides a Toolbar. **hide** returns True if the Toolbar is successfully hidden. **hide** performs the same function as the procedure `hideToolbar`.

Example

The following example hides a Toolbar named `MyToolbar`. If the Toolbar is not visible, this method displays it:

```
method pushButton (var eventInfo Event)
var
    tbar Toolbar
endvar

if tbar.attach("MyToolbar") then
    if tbar.isVisible() then
        tbar.hide()
    else
        tbar.show()
    endif
endif

endMethod
```

isVisible method

Toolbar

Determines whether the specified Toolbar is visible.

Syntax

```
isVisible ( ) Logical
```

Description

isVisible determines whether the specified Toolbar is visible. **isVisible** returns True if the Toolbar is visible and False if the Toolbar is hidden. This method performs the same function as the **isToolbarShowing** procedure.

Example

The following example prints a message stating whether a Toolbar named `MyToolbar` is visible. If this code can not attach to `MyToolbar`, an "Unable to attach" message appears.

```
method pushButton (var eventInfo Event)
var
    tbar Toolbar
endvar

if tbar.attach("MyToolbar") then
    if tbar.isVisible() then
        msgInfo("MyToolbar" , "Toolbar is Visible")
    else
        msgInfo("MyToolbar" , "Toolbar is not Visible")
    endif
else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```


remove method**Toolbar**

Removes the specified Toolbar from the screen.

Syntax

```
remove ( ) Logical
```

Description

remove removes the specified Toolbar from the screen. **remove** returns True if the Toolbar was successfully removed; otherwise it returns False.

Example

The following example removes a Toolbar named MyToolbar from the screen. If this code can not attach to MyToolbar, an "Unable to attach" message appears.

```
method pushButton (var eventInfo Event)
var
    tbar Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.remove()
else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

removeButton method**Toolbar**

Removes a button from the Toolbar.

Syntax

```
removeButton ( const idCluster SmallInt, const index SmallInt ) Logical
```

Description

removeButton removes a button from the Toolbar based on a zero based index placement. The position of the button (from left to right starting at 0) is specified by *index*. **removeButton** returns True if the button is successfully removed.

Note

- The first parameter is ignored must still be set as a valid Cluster Identifier.
- A separator bar is considered a button position and must be taken into account when determining which position to place into the **removeButton** method. For example,

```
removeButton(idcluster, 5)
```

will remove the fifth button on the toolbar unless the fifth position is a separator bar, in which case the separator bar will be removed.

Example

The following example removes the a button from a Toolbar named MyToolbar. Because *index* starts with zero, this example removes the third button from the toolbar. If this code cannot attach to MyToolbar, an "Unable to attach" message appears. When you define the parameters, the first parameter is ignored but must still be a valid Cluster identifier.

setPosition method

```
method pushButton (var eventInfo Event)
var
    tbar Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.removebutton(0,2) ;idcluster=0, but this is a reserved parameter and
must be 0
                        ;index=2, the 3rd from left

else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

setPosition method

Toolbar

Changes the position of a floating Toolbar.

Syntax

```
setPosition ( const x LongInt, const y LongInt ) Logical
```

Description

setPosition changes the position of a floating Toolbar to the coordinates specified in *x* and *y*. The *x* and *y* coordinates are specified in pixels and relative to the upper-left corner of the screen. **setPosition** returns True if the position of the Toolbar is successfully changed.

Example

The following example changes the position of a Toolbar named MyToolbar 500 pixels to the right and 400 pixels up from its current position.

```
method pushButton (var eventInfo Event)
var
    liX, liY LongInt
    tbar Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.getPosition(liX, liY)
    view("From: " + string(liX) + ", "
        + string(liY) +
        "To: " + string(liX + 2800) + " , "
        + string(liY + 2800))
    tbar.setPosition(liX + 500, liY + 400)
endif

endMethod
```

setState method

Toolbar

Sets the state of the Toolbar.

Syntax

```
setState ( const state SmallInt ) Logical
```

Description

setState sets the Toolbar to the specified *state*. There are six Toolbar states:

- ToolbarStateTop: docked at the top of the window
- ToolbarStateLeft: docked at the left of the window
- ToolbarStateRight: docked on the right side of the window
- ToolbarStateBottom: docked at the bottom of the window
- ToolbarStateFloatHorizontal: floating horizontally
- ToolbarStateFloatVertical: floating vertically

setState returns True if the Toolbar state is successfully set.

Example

The following example displays a dialog that allows the user to set the state of the Toolbar named MyToolbar. If this code can not attach to MyToolbar, an "Unable to attach" message appears.

```
method pushButton (var eventInfo Event)
var
    siState  SmallInt
    tbar     Toolbar
endvar

if tbar.attach("MyToolbar") then
    siState = tbar.getState()
    siState.view("Enter State: (0-7)")
    tbar.setState(siState)
else
    msgInfo("Toolbar error", "Unable to attach.")
endif

endMethod
```

show method**Toolbar**

Shows a Toolbar.

Syntax

```
show ( ) Logical
```

Description

show shows a Toolbar.

There are five toolbars available on the Paradox desktop: Standard, Global, Object, Align, and Format. The Standard toolbar is available by default and can be displayed or hidden using the **show** and **hide** methods.

This method performs the same function as the **showToolbar** procedure.

The Object and Align toolbars are used in the design environment only and are not available through ObjectPAL. To display the Global and Format toolbars using the **show** method, you must first issue the following PAL statements:

```
; to display the Global toolbar
winPostMessage(ap.windowHandle(),winGetMessageID("WM_COMMAND"),30831,0)

; to display the Format toolbar
winPostMessage(ap.windowHandle(),winGetMessageID("WM_COMMAND"),7903,0)
```

unAttach method

Example

The following example hides a Toolbar named MyToolbar. If the Toolbar is already hidden, this method displays it.

```
method pushButton (var eventInfo Event)
var
    tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
    if tbar.isVisible() then
        tbar.hide()
    else
        tbar.show()
    endif
endif

endMethod
```

unAttach method

Toolbar

Removes the attachment from the Toolbar.

Syntax

```
unAttach ( ) Logical
```

Description

unAttach removes the attachment from the Toolbar.

Example

The following example attaches a Toolbar named MyToolbar, sets its state, and then unattaches.

```
method pushButton (var eventInfo Event)
var
    tbar      Toolbar
endvar

if tbar.attach("MyToolbar") then
    tbar.setState(ToolbarStateTop)
    tbar.unattach()
endif

endMethod
```

showApplicationBar method

Toolbar

Toggles the visible property of the Application Bar.

Syntax

```
showApplicationBar ( Show Logical ) Logical
```

Show=True If the Application bar is not already visible, it will appear

Show=False If the Application bar is visible, it will be hidden

Description

showApplicationBar toggles the visible property of the Application Bar.

Example

This example uses the `isAppBarVisible` method to check the state of the Application bar, then uses `showApplicationbar` to either show or hide the Application bar depending on its state.

```
method pushButton(var eventInfo Event)
  if isappbarvisible()==True then      ;checks current state of application bar
    msginfo("stop","application bar is being hidden")
    Showapplicationbar(False)          ; hides application bar
  else
    msginfo("Stop","application bar is being shown")
    showapplicationbar(True)           ;shows application bar
  endif
endMethod
```

isAppBarVisible method**Toolbar**

Checks the state of the Application Bar.

Syntax

```
isAppBarVisible ( ) Logical
```

Description

isAppBarVisible checks the state of the Application bar and returns a logical.

Example

See the example for `showApplicationBar`.

TimerEvent type

The `TimerEvent` type includes methods that process information used by the timer method. Timer methods are built into each design object. Use **setTimer** (defined for the `UIObject` type) to specify when to send timer events to an object and then modify the object's built-in **timer** method to control the object's response when a timer goes off. Use **killTimer** (defined for the `UIObject` type) to turn off an object's timer.

The `Timer` type contains only derived methods from the `Event` type.

Methods for the TimerEvent Type

Event	←	TimerEvent
errorCode		(All <code>TimerEvent</code> methods are derived methods from <code>Event</code> type.)
getTarget		
isFirstTime		
isPreFilter		
isTargetSelf		
reason		
setErrorCode		
setReason		

The following example assumes that a form contains a multi-record object bound to the *Customer* table. The record container in the multi-record object is named *custRecordMRO*.

isAppBarVisible method

For the following example, suppose you want to give the user 60 seconds to edit a record in a data entry program. After 60 seconds, you want to alert the user. To accomplish this, the built-in **action** method for *custRecordMRO* tests every action. If the action is *DataArriveRecord*, the method uses **killTimer** to stop old timers and uses **setTimer** to set a new timer. When the timer goes off, a message pops up alerting the user. The following code defines a constant in the Const window for *custRecordMRO*. This code makes it easy for you to change the time:

```
; custRecordMRO::Const
const
    alertTime = 60000      ; data-entry alert at 60 seconds
endConst
```

The following code is attached to the **action** method for *custRecordMRO*;

```
; custRecordMRO::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 *custRecordMRO*:

```
; custRecordMRO::timer
method timer(var eventInfo TimerEvent)
self.killTimer()
beep()
msgInfo("Alert", "You have been processing this record for " +
        "one minute now.")
endMethod
```

UIObject type

UIObjects (user interface objects) create the user interface for an application. Anything you can place in a form or report is a UIObject. UIObjects include bands, bitmaps, boxes, buttons, cells, charts, crosstabs, ellipses, field objects, forms, groups, lines, lists, multi-record objects, OLE objects, pages, record objects, table frames, and text boxes.

Only UIObjects in forms have built-in event methods. You can attach code to those built-in event methods, and a form responds to events. For methods and procedures that work only with forms, use the Form type.

You can also use built-in object variables to refer to UIObjects. This technique is especially useful for creating generalized code.

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 to which the table is bound. TCursor methods 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.

Note

- Some table operations require Paradox to create temporary tables. Paradox creates these tables in the private directory.

Some table operations are considerably faster with TCursors than with UIObjects. For example, to perform a table-oriented operation that causes a high volume of screen refreshes, you can declare a TCursor, attach it to the object the table is already bound to (e.g., a table frame), perform the operation with the TCursor and 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 active record for the object. After you perform a TCursor operation (e.g., a **locate**), the TCursor might point to a different record. To have the object point to the same record as the TCursor, use the **resync** method; to make the TCursor point to the same record as the object, use the **attach** method. For more information, see the example for **insertRecord**.

The UIObject type includes several derived methods from the AnyType type.

Methods for the UIObject type

AnyType	← UIObject		
blank	action	getPosition	mouseDown
dataType	atFirst	getProperty	mouseExit
isAssigned	atLast	getPropertyAsString	mouseMove
isBlank	attach	getRange	mouseRightDouble
isFixedType	bringToFront	getRGB	mouseRightDown
unAssign	broadcastAction	hasMouse	mouseRightUp
	cancelEdit	home	mouseUp
	convertPointWithRespectTo	insertAfterRecord	moveTo
	copyFromArray	insertBeforeRecord	moveToRecNo
	copyToArray	insertRecord	moveToRecord
	copyToToolbar	isContainerValid	nextRecord
	create	isEdit	nFields
	currRecord	isEmpty	nKeyFields

action method

delete	isLastMouseClickedValid	nRecords
deleteRecord	isLastMouseRightClickedValid	pixelsToTwips
dropGenFilter	isRecordDeleted	postAction
edit	keyChar	postRecord
empty	keyPhysical	priorRecord
end	killTimer	pushButton
endEdit	locate	recordStatus
enumFieldNames	locateNext	resync
enumLocks	locateNextPattern	rgb
enumObjectNames	locatePattern	sendToBack
enumSource	locatePrior	setGenFilter
enumSourceToFile	locatePriorPattern	setPosition
enumUIClasses	lockRecord	setProperty
enumUIObjectNames	lockStatus	setRange
enumUIObjectProperties	menuAction	setTimer
execMethod	methodEdit	skip
forceRefresh	methodGet	switchIndex
getBoundingBox	methodSet	twipsToPixels
getGenFilter	methodDelete	unDeleteRecord
getHTMLTemplate	mouseClick	unlockRecord
		view
		wasLastClicked
		wasLastRightClicked

action method

UIObject

Performs a specified action.

Syntax

action (const *actionId* SmallInt) Logical

Description

action specifies an *actionId* to perform in response to an event. *actionId* is a constant in one of the following action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

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 ObjectPAL's constants. You can use them to signal other parts of an application.

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, while this method causes an ActionEvent.

Example

The following example is attached to a button's **mouseUp** method. If you click the button, the pointer moves to the next record. If you press and hold SHIFT and click the button, the pointer moves to the next set of records.

The action constants `DataFastForward` and `DataNextRecord` behave like the Fast Forward and Next Record buttons on the Toolbar. Assume that *CUSTOMER* refers to a table frame on the form and that *nextRecordOrFast* is a button on the form. Because the *nextRecordOrFast* button is not in the same containership hierarchy as *CUSTOMER*, the action doesn't bubble up to *CUSTOMER* automatically. Instead, the action must be sent to the *CUSTOMER* object explicitly.

```
; nextRecordOrFast::mouseUp
method mouseUp(var eventInfo MouseEvent)
; if the tableFrame isn't active, then move to it
if NOT CUSTOMER.focus then
    CUSTOMER.Name.moveTo()
endif
; 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
```

atFirstmethod**UIObject**

Reports if the pointer is positioned at the first record of a table.

Syntax

```
atFirst ( ) Logical
```

Description

atFirst returns True if the pointer is positioned 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

The following example assumes that 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 whether the TCursor points to the first record. If **atFirst** returns False, this code moves the TCursor 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 it
    orders.moveToRecord(tc) ; move highlight to first record
else
    msgStop("Currently on record " + String(tc.recNo()),
            "You're already at the top of the list!")
endif
```

atLast method

```
endIf  
endMethod
```

atLast method

UIObject

Reports if the pointer is positioned at the last record in a table.

Syntax

```
atLast ( ) Logical
```

Description

atLast returns True if the pointer is positioned 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

The following example assumes that 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 whether the TCursor points to the last record. If **atLast** returns False, this code moves the TCursor 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
```

attach method

UIObject

Binds a UIObject variable to a specified design object.

Syntax

1. **attach** () Logical
2. **attach** (const **objectVar** UIObject) Logical
3. **attach** (const **objectName** String) Logical
4. **attach** (const **form** Form [, **objectName** String]) Logical
5. **attach** (const **report** Report [, **objectName** String]) Logical

Description

attach binds a UIObject variable to a specified design object. You can also use **attach** to assign a UIObject to an item in an Array.

Syntax 1 binds the variable to the object that called **attach** (*self*).

Syntax 2 binds the variable to another UIObject which is specified by a UIObject variable (*objectVar*) in one of the following ways:

Specification	Example
UIObject variable	<pre>var u1, u2 UIObject endVar u1.attach() ; Attach to self. u2.attach(u1) ; Attach to a UIObject variable.</pre>
UIObject name	<pre>var u1 UIObject endVar ; Attach to an object named nameFld. u1.attach(nameFld)</pre>
Containership path	<pre>var u1 UIObject aForm Form endVar aForm.open("aform.fsl") ; Attach to an object named aFld. u1.attach(aForm.aPage.aField)</pre>

Syntax 3 binds the variable to another UIObject specified by name in *objectName*. For example, if a form contains a box named *theFrame*, the following statement binds the UIObject variable *ui* to the box:

```
ui.attach("theFrame")
```

Syntax 4 binds the variable to the form which is specified by the Form variable *form*, or to a UIObject in that form which is specified by *objectName*.

Syntax 5 binds the variable to the report which is specified by the Report variable *report*, or to a UIObject which is specified by *objectName*.

Example 1

The following example displays various forms of the **attach** syntax. First, the code attaches a variable named *objBox* to the active object (*self*) and changes its color. Next, the code attaches *objBox* to another object and uses *objBox* to change that object's color. A second example for of the same syntax opens another form, attaches *objBox* to a box on the second form, and uses *objBox* to change the color of the other form's object.

You can attach to an object name on another form by including the form handle in the object name. Provide the handle to the form in the first argument and the object name on the specified form as a string in the string.

This example assumes the active form contains two boxes, *thisBox* and *thatBox* and the secondary form contains one box, named *otherBox*. The code is attached to *thisBox*.

```
; thisBox::mouseUp
method mouseUp(var eventInfo MouseEvent)
var
  objBox,
  objForm    UIObject
  otherForm  Form
endVar

objBox.attach() ; binds objBox to thisBox
```

attach method

```
objBox.color = DarkMagenta

objBox.attach(thatBox)    ; binds objBox to thatBox
objBox.color = Magenta

; assume the form uiattch2.fsl exists and it has
; one object named otherBox
if otherForm.open("uiattch2.fsl") then
  objBox.attach(otherForm.otherBox)
  objBox.color = DarkBlue
  sleep(2000)
  otherForm.close()
endIf

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
```

Example 2

The following example uses **attach** to assign a UIObject to an item in an array:

```
method pushButton(var eventInfo Event)
  const
    kOneInch = 1440 ; One inch = 1,440 twips.
    kShowHandles = Yes
  endConst

  var
    foForm      Form
    uiTempObj    UIObject
    arObjects    Array[2] UIObject
  endVar

  foForm.create()

  uiTempObj.create(BoxTool, 700, 700, kOneInch, kOneInch, foForm)
  arObjects[1].attach(uiTempObj)

  uiTempObj.create(BoxTool, 700, 2500, kOneInch, kOneInch, foForm)
  arObjects[2].attach(uiTempObj)

  foForm.setSelectedObjects(arObjects, kShowHandles)

endMethod
```

Note

- Some of the methods in the UIObject class can be used for forms if you attach a UIObject variable to the form. Syntax 4 of the attach method allows you to attach a UIObject variable to a form so that you can access those methods. For example, to send a mouseUp event to another form's form-level **mouseUp** built-in event method, you must attach a UIObject variable to an open form.

bringToFront method**UIObject**

Displays an object in front of other objects.

Syntax

```
bringToFront ( )
```

Description

bringToFront moves a UIObject to the front drawing layer of a window, displaying it in front of other objects. If UIObject is a form, this method displays the form window in front of all other windows.

This method works in both design and run mode, and you do not have to select the object. The effects of this method might not be noticeable unless the objects partially overlap. This method is also used if you want to rearrange the objects' tab order.

Note

- When you change the front-to-back positions of objects, you also change their tab order. Objects always tab from back to front.

Example

In the following example, the **pushButton** method for a button displays an animated sequence of twelve bitmaps. This code uses two **for** loops and the **bringToFront** method to cycle through the bitmaps forward and backward.

```
;btn1 :: pushButton
method pushButton(var eventInfo Event)
  var
    siCounter SmallInt
  endVar

  ;Cycle through bitmaps.
  for siCounter from 1 to 12
    ; Assume the bitmap objects have names like bmp1, bmp2, etc.
    pge1("bmp" + string(siCounter)).bringToFront()
    sleep(100)
  endFor

  ;Cycle through bitmaps in reverse order.
  for siCounter from 11 to 1 step -1
    pge1("bmp" + string(siCounter)).bringToFront()
    sleep(100)
  endFor
endMethod
```

broadcastAction method**UIObject**

Broadcasts an action to an object and the objects it contains.

Syntax

```
broadcastAction (const actionID SmallInt)
```

Description

broadcastAction sends the ActionEvent specified in *actionID* to an object, and then sequentially to each object it contains. The action is sent depth-first through the containership hierarchy, not breadth-first. By default, contained objects bubble the action up through the hierarchy.

broadcastAction method

For example, suppose a page named *thePage* contains two boxes (*boxOne* and *boxTwo*) and *boxOne* contains a button *btnOne*. A call to the **Page.broadcastAction(actionID)** sends the action specified by *actionID* to the objects in the following order:

1. *thePage* (specified by dot notation)
2. *boxOne* (contained by thePage)
3. *btnOne* (contained by boxOne, at a lower level in the hierarchy)
4. *boxTwo* (also contained by thePage, at the same level as boxOne in the hierarchy)

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCommands
- ActionSelectCommands

Example

In the following example, the form's built-in **action** method uses **broadcastAction** to send all the objects in the page *pgel* a user-defined action. When the form switches to edit mode, it sends *UserAction + 1*. When the form leaves edit mode, it sends *UserAction + 2*. Each field's label then uses the user-defined action to color each label Red or Black.

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

```
;frm1 :: action
method action(var eventInfo ActionEvent)

    if eventInfo.isPreFilter() then
        ; This code executes for each object on the form:

    else
        ; This code executes only for the form:

        switch
            case eventInfo.id() = DataBeginEdit :
                pgel.broadcastAction(UserAction + 1)

            case eventInfo.id() = DataEndEdit :
                pgel.broadcastAction(UserAction + 2)
        endSwitch

    endIf

endmethod
```

The following code is attached to each label's built-in **action** method:

```
;label :: action
method action(var eventInfo ActionEvent)
    ;Duplicate this code on each object (or create a prototype object)
    ;you wish toggle the font color from black to red when
    ;the form goes in and out of edit mode.

    switch
        case eventInfo.id() = UserAction + 1 : self.font.color = Red
```

```

        case eventInfo.id() = UserAction + 2 : self.font.color = Black
    endSwitch
endmethod

```

cancelEdit method

UIObject

Cancels record changes without ending Edit mode.

Syntax

```
cancelEdit ( ) Logical
```

Description

cancelEdit cancels changes you've made to the active record. This method returns True if successful; otherwise, it returns False. To cancel record changes, use **cancelEdit** before moving the pointer from the active record. If you move the pointer, changes to the record are committed.

cancelEdit has the same effect as the action constant `DataCancelEdit`. This means that the following statements are equivalent:

```

obj.cancelEdit()
obj.action(DataCancelEdit)

```

Example

The following example attaches a UIObject variable (*noChange*) to a table frame (*CUSTOMER*). (From then on *noChange* is used as a handle to the table frame.) The code searches for a value in the *Customer* table, and, if found, changes the value. Before leaving the record, the change is canceled using the **cancelEdit** method. This example assumes that you have one page on the form (*pageOne*), a table frame attached to the *Customer* table, and a button (*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

```

convertPointWithRespectTo method

UIObject

Changes the frame of reference for calculating the coordinates of a point.

Syntax

```

convertPointWithRespectTo ( const otherUIObject UIObject, const oldPoint Point, var
convertedPoint Point )

```

copyFromArray method

Description

convertPointWithRespectTo changes the frame of reference for calculating the coordinates of a point. Coordinates are usually 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

The following example retrieves and displays the position of an object named *innerBox*. *innerBox* is contained by *outerBox* on a page named *pageOne*. First, the position of *outerBox* relative to the upper-left corner of the page is calculated. Next, the position of *innerBox*, relative to the upper-left corner of *outerBox* is calculated. Finally, the position of *innerBox* is converted with respect to the page, allowing you to determine the distance between *innerBox* and 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
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
innerBox.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?
outerBox.convertPointWithRespectTo(pageOne, innerPos, convertedPos)
convertedPos.view("Inner box position converted")
endMethod
```

copyFromArray method

UIObject

Copies data from an array to a table record.

Syntax

```
copyFromArray ( const ar Array[ ] AnyType) Logical
```

Description

copyFromArray copies data from an array *ar* to a UIObject (usually 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.

This method fails if you 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.) This method also 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

The following example assumes that 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 code edits *CUSTNAME*, creates an array with three elements, creates a new record in *CUSTNAME* and 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) ; copy the array to the new record
CUSTNAME.endEdit()
endMethod

```

copyToArray method

UIObject

Copies data from a record to an array.

Syntax

```
copyToArray ( var ar Array [ ] AnyType ) Logical
```

Description

copyToArray copies fields from the record of a UIObject (usually a table frame or multi-record object) to an array. You must declare the array as AnyType, or as a type that matches each field in the table. If the array is resizable, it expands to hold the number of fields in the record. If the array is not resizable, it discards the fields it cannot hold.

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 are not copied to the array.

Example

The following example assumes that there are two table frames on a form, named *CUSTOMER* and *CUSTARC*, and one button, named *archiveButton*. The form itself is renamed *thisForm*. When *archiveButton* is pushed, the active record in *CUSTOMER* is moved to *CUSTARC*.

This code looks at the Editing property of the form; if it's False, this code starts Edit mode. *copyToArray* then copies the active record in *CUSTOMER* to the *arcRecord* array and deletes the active record. If the active record can't be locked and deleted, it is not copied to the target table *CUSTARC*. If the record can be deleted, *copyToArray* writes the contents of the array to a new blank record in the target table.

```

; 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 active 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

```

copyToToolbar method

```
CUSTARC.insertRecord()          ; insert blank record
CUSTARC.copyFromArray(arcRecord) ; copy array to blank record
endIf

endMethod
```

copyToToolbar method

UIObject

Copies an object to the Toolbar where it can be used as a prototype object.

Syntax

```
copyToToolbar ( ) Logical
```

Description

copyToToolbar copies an object (including its properties and methods) to the Toolbar. New objects created using the corresponding Toolbar tool will have the new properties, and existing objects do not change.

For example, create a box (interactively or using ObjectPAL) and set its color to red, and add code to its built-in **mouseClick** method. If you copy this box to the Toolbar, all new boxes you create are red and have the same code attached to the **mouseClick** method.

copyToToolbar copies all component objects in a compound object. For example, when you copy a labeled field object, you copy the field object, the label, and the edit region. Tables include headers, labels, records, and fields. Multi-record objects include records only. Crosstabs include cells and fields. They can distinguish the three different cell types, so you can have three different types of fields which have different colors, and so on.

You can also use **copyToToolbar** to copy the component objects separately. However, if an object contains objects, but is not a compound object, the contained objects are not copied.

Changes you make using **copyToToolbar** apply only to the current Paradox session. To save the tool's new properties to the next session, call **saveStyleSheet**.

If an object does not have a corresponding tool on the Toolbar, Paradox copies its properties and methods to a hidden tool. All new objects of that type will have those properties and methods. For example, the Toolbar does not have a tool for creating a page. However, you can set a page's properties and methods, and call **copyToToolbar** to apply the same properties and methods to a new page.

Example

In the following example, a button named *btnCreateStyleSheet* uses **enumObjectNames** to fill an array *arObjNames*. A **for** loop cycles through the array and copies the object to the Toolbar using **copyToToolbar**. A call to **saveStyleSheet** creates (or overwrites) a style sheet with the name in the String variable *stSheet*. You can paste the following code into the **pushButton** method for a button on any form you want to use as a style sheet:

```
;btnCreateStyleSheet :: pushButton
method pushButton(var eventInfo Event)
  var
    f          Form
    stSheet    String
    arObjNames Array[] Anytype
    siCounter  SmallInt
  endVar

  f.attach()          ; Attach to this form.
  f.enumObjectNames(arObjNames) ; Fill array with object names.

  ; Prompt user for name of new style sheet.
```

```

stSheet = "Style sheet name"
stSheet.view("Enter name of style sheet")

if stSheet = "Style sheet name" then ; If variable was not changed,
    return ; quit the operation.
endif

for siCounter from 1 to arObjNames.size() ; Cycle through array
    copyToToolbar(f.(arObjNames[siCounter])) ; and copy objects to
endfor ; the Toolbar.

if not f.saveStyleSheet(stSheet, True) then
    errorShow("Error saving style sheet", "Check path & filename")
endif
endMethod

```

create method

UIObject

Creates an object.

Syntax

1. **create** (const **objectType** SmallInt, const **x** LongInt, const **y** LongInt, const **w** LongInt, const **h** LongInt [, const **container** UIObject])
2. **create** (const nativeObject Binary, const container UIObject) Logical

Description

create creates the object specified in **objectType** (use one of the UIObjectTypes constants) 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 in twips. The optional argument container specifies a container object for the new object.

Syntax 2, uses **create** to create the object specified by *nativeObject*. *nativeObject* is a binary object that can be generated by pasting a UIObject (Corel Form Object) from the Clipboard. **create** works only in form design mode. **create** returns True if successful; otherwise it returns False.

Note

- When you use **create** to create an object, the object is invisible. To make it visible, set its Visible property to True. To delete an object at run time use the **delete** method.

Example

In the following example, code is attached to the **mouseUp** method for *pageOne* on a form. This example creates a box, names it *Fred*, colors it blue, and makes it visible. The code then creates an ellipse whose size position is specified in *Fred*, and whose container is set to *Fred*.

```

; pageOne::mouseUp
method mouseUp(var eventInfo MouseEvent)
const
    kOneInch = 1440 ; One inch = 1,440 twips.

endConst

var
    ui UIObject
    fm Form
endvar

; create a Blue box, named Fred, and make it visible
ui.create(BoxTool, 144, 144, 2 * kOneInch, 2 * kOneInch)

```

currRecord method

```
ui.Name = "Fred"
ui.Color = Blue
ui.Visible = True
; create a Green ellipse inside Fred, named Bill
fm.attach()

ui.create (EllipseTool, 288, 288, kOneInch, kOneInch, fm.Fred)
ui.Name = "Bill"
ui.Color = Green
ui.Visible = True
endMethod
```

currRecord method

UIObject

Reads the active record into the record buffer.

Syntax

```
currRecord ( ) Logical
```

Description

currRecord cancels changes you've made to the active record, and displays a refreshed version from saved data. **currRecord** leaves a locks on locked records. This method returns True if successful; otherwise, it returns False.

currRecord has the same effect as the action constant DataRefresh. This means that the following statements are equivalent:

```
obj.currRecord()
obj.action(DataRefresh)
```

Example

The following examples assumes 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
```

delete method

UIObject

Deletes an object from a form.

Syntax

```
delete ( )
```

Description

delete deletes an object from a form at run time.

Example

The following examples assumes that a form contains a method that creates a box named *Fred* and an ellipse inside *Fred* named *Bill*. Because these objects are created at run time, they can't be referenced directly by this method. This code attaches to the object using a string evaluated at run time. See the example for **create** for details about the **mouseUp** method (on the 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
```

deleteRecord method**UIObject**

Deletes the active record from a table.

Syntax

```
deleteRecord ( ) Logical
```

Description

deleteRecord deletes the active record from a table without prompting for confirmation. This method returns True if successful; otherwise, it returns False. Deleted dBASE tables can be restored after **deleteRecord**, but Paradox tables cannot.

deleteRecord has the same effect as the action constant `DataDeleteRecord`. This means that 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 active record in *CUSTOMER* is moved to *CUSTARC*.

To begin, this method determines the Editing property of the form; if it is False, this code starts Edit mode. This code then copies the active record in *CUSTOMER* to the *arcRecord* array and deletes the active record. If the active record can't be locked and deleted, it is not copied to the target table. If the

dropGenFilter method

record can be deleted, this code writes the contents of the array to the target table in a new blank 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 active 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
```

dropGenFilter method

UIObject

Removes the filter criteria associated with a field, multi-record object, or table frame.

Syntax

```
dropGenFilter ( ) Logical
```

Description

dropGenFilter removes the filter criteria associated with a UIObject. All associated indexes and ranges remain.

Example

In the following example, a form's data model contains the *Orders* and *Lineitem* tables linked 1:M. The form also contains a button named *btnDropFilters*. The **pushButton** method for *btnDropFilters* uses **dropGenFilter** on one UIObject connected to each table in the data model. This code allows you to remove filter criteria from complex forms.

```
;btnDropFilters :: pushButton
method pushButton(var eventInfo Event)
    ; Order_No is a field object bound to
    ; the Order No field in the Orders table.
    Order_No.dropGenFilter()

    ; LINEITEM is a table frame bound to the Lineitem table.
    LINEITEM.dropGenFilter()
endMethod
```

edit method

UIObject

Puts a table in Edit mode.

Syntax

```
edit ( ) Logical
```

Description

edit puts all tables on a form in Edit mode, allowing you to make changes. If the form is already in Edit mode, **edit** is ignored.

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** to cancel changes to the record before moving on.

edit has the same effect as the action constant **DataBeginEdit**. This means that the following statements are equivalent:

```
obj.edit()
obj.action(DataBeginEdit)
```

Example

The following examples assumes that a form contains a table frame bound to the *Orders* table, and one button, named *changeDate*. The **pushButton** method for *changeDate* examines the Sale Date and Ship Date fields of the active 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".value < ORDERS."Ship Date".value then
; start Edit mode for the form
ORDERS.edit()
; if Sale Date is later than Ship Date, change Ship Date
ORDERS."Ship Date".value = ORDERS."Sale Date".value + 5
ORDERS.endEdit()           ; end editing – changes to the record
                           ; can't be canceled
endif

endMethod
```

empty method**UIObject**

Deletes all records from a table.

Syntax

```
empty ( ) Logical
```

Description

empty deletes all records from a table without prompting for confirmation. The table does not have to be in Edit mode, but a write lock (at least) is required. This operation cannot be undone for Paradox tables, and does not affect SQL tables.

empty removes information from the table, without deleting the table itself. Compare this method to **delete** (Table type), which does delete the table.

empty tries to place a write lock on the table. **empty** must delete each record one at a time. For dBASE tables, this method flags all records as deleted, without removing them from the table. Records can be undeleted from a dBASE table using the **unDeleteRecord** method (unless they have been removed using the **compact** (TCursor type) method).

empty method

Example

The following example assumes that a form has three buttons: *createTable*, *emptyTable*, and *deleteTable*. *createTable* creates a copy of the *Orders* table named *TmpOrder* and 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 form's data model, and deletes the temporary table.

The following code attaches to 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 a TableFrame.
ui.TableName = "TmpOrder.db" ; This also adds table to data model.
ui.Visible = True

endMethod
```

The following code attaches to 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
```

The following code attaches to 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() ; Delete table frame.
    DMRemoveTable("TmpOrder.db") ; Remove table from data model.
    tbl.attach("TmpOrder.db")
    tbl.delete() ; Delete table.
endIf
endMethod
```


end method**UIObject**

Moves to the last record in a table.

Syntax

```
end ( ) Logical
```

Description

end moves to the last record in a table.

end has the same effect as the action constant `DataEnd`. This means that the following statements are equivalent:

```
obj.end()
obj.action(DataEnd)
```

Example

The following example moves to the last record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form, and that *moveToEnd* is a button on the 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
```

endEdit method**UIObject**

Removes a table from Edit mode and posts changes to the active record

Syntax

```
endEdit ( ) Logical
```

Description

endEdit removes a table from Edit mode and posts changes to the active record.

endEdit has the same effect as the action constant `DataEndEdit`. This means that the following statements are equivalent:

```
obj.endEdit()
obj.action(DataEndEdit)
```

Example

See the **edit** example.

enumFieldNames method**UIObject**

Fills an array with the names of fields in a table.

Syntax

```
enumFieldNames ( var fieldArray Array[ ] String ) Logical
```

Description

enumFieldNames fills *fieldArray* with the names of the fields in a table. *fieldArray* is a resizable array that you must declare and pass as an argument. If *fieldArray* already exists, this method overwrites it without asking for confirmation. **enumFieldNames** returns `True` if it succeeds; otherwise, it returns `False`.

enumLocks method

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 named *getFieldNames*.

```
; getFieldNames::pushButton
method pushButton(var eventInfo Event)
var
    fieldNames Array[] String
endVar
ORDERS.enumFieldNames(fieldNames)
fieldNames.view()
endMethod
```

enumLocks method

UIObject

Creates a Paradox table listing the locks currently applied to a UIObject, and returns the number of locks.

Syntax

```
enumLocks ( const tableName String ) LongInt
```

Description

enumLocks 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, **enumLocks** fails. For dBASE tables, this method lists only the lock that you've placed (not all locks currently on the table).

You can specify an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of *tableName*:

Field name	Field type	Description
UserName	A15	User name
LockType	A32	Lock type (e.g., Table Write Lock)
NetSession	N	Net level session number
Session	N	BDE session number (for locks placed by BDE)
RecordNumber	N	Record number (for record locks or image locks; otherwise, 0)

Example

In the following example, the built-in **pushButton** method for the *showLocks* button creates a table listing the locks specified for 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

```

enumObjectNames method/procedure

UIObject

Fills an array with the names of the objects in a form.

Syntax

```
enumObjectNames ( var objectNames Array[ ] String )
```

Description

enumObjectNames fills an array with object names. *arrayName* is a resizable array that you declare and pass as an argument. If *arrayName* already exists, this method overwrites it without asking for confirmation.

enumObjectNames returns the names of bound and unbound objects, beginning with the object that called the method, including the paths to objects that object contains. To enumerate all objects in a form, start **enumObjectNames** with the form. To enumerate all objects in a page, start it with the page. To enumerate all objects in a box, start it with the box.

To list object names in a table use **enumUIObjectNames**.

Example

The following example demonstrates the difference between **enumObjectNames** (which lists object names in an array) and **enumUIObjectNames** (which lists object names in a table). In this example, the **pushButton** method for *getObjectNames* writes all object names on the form to an array and to a table.

```

; getObjectNames::pushButton
method pushButton(var eventInfo Event)
  var
    foThisForm    Form
    arObjNames    Array[] String
    stTbName      String
    tvObjNames    TableView
  endVar

  stTbName = "objTable.db"
  foThisForm.attach() ; Get a handle to the current form.

  foThisForm.enumObjectNames(arObjNames)
  arObjNames.view("Objects in this form:")

  foThisForm.enumUIObjectNames(stTbName)
  tvObjNames.open(stTbName)
endMethod

```

enumSource method

UIObject

Fills a table with the source code of the methods on a form.

Syntax

```
enumSource ( const tableName String, [ const recurse Logical ] ) Logical
```

Description

enumSource 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 specify an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of the table created by **enumSource**:

Field name	Type	Description
Object	A128	Object name
MethodName	A128	Method name
Source	M64	ObjectPAL code

If *recurse* is False, **enumSource** returns the method definitions for overridden methods on the active object. To include the source code for overridden methods on objects contained by the active object, set *recurse* to True.

If *recurse* is True, **enumSource** returns the definitions for overridden methods, beginning with the object that called this method, and including paths to objects that object contains. To enumerate all objects in a form, start **enumSource** with the form. To enumerate all objects in a page, start it with the page. To enumerate all objects in a box, start it with the box.

Note

- If the *recurse* parameter is not included, then it is assumed to be True and the source code of methods for all objects contained by the object will be returned.

Example

The following example uses **enumSource** to retrieve the source code for the entire form and to retrieve only the source code for a button named *btnCancel*:

```
; getObjectNames::pushButton
const
    kRecurse    = Yes
    kNoRecurse  = No
endConst

method pushButton(var eventInfo Event)
    var
        foThisForm    Form
        stTbName       String
        tvSource       TableView
    endVar

    stTbName = "objSrc.db"
    foThisForm.attach() ; Get a handle to the current form.

    foThisForm.enumSource(stTbName, kRecurse)
    tvSource.open(stTbName)

    ; Suspend execution until you close the table view.
    tvSource.wait()

    btnCancel.enumSource(stTbName, kNoRecurse)
```

```

    tvSource.open(stTbName)
endMethod

```

enumSourceToFile method

UIObject

Writes the source code for a form or an object to a text file.

Syntax

```
enumSourceToFile ( const fileName String [ , const recurse Logical ] ) Logical
```

Description

enumSourceToFile 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 specify an alias or path in *fileName* . If an alias or path is not specified, Paradox creates *fileName* in the working directory.

If *recurse* is False, **enumSourceToFile** returns the method definitions for overridden methods on the active object. To include the source code for overridden methods on objects contained by the active object, set *recurse* to True.

If *recurse* is True, **enumSourceToFile** returns the definitions for overridden methods, beginning with the object that called this method, and including paths to objects that object contains. To enumerate all objects in a form, start **enumSourceToFile** with the form. To enumerate all objects in a page, start it with the page. To enumerate all objects in a box, start it with the box.

Note

- If the *recurse* parameter is not included, then it is assumed to be True and the source code of methods for all objects contained by the object will be returned.

Example

The following example uses **enumSourceToFile** to retrieve the source code for the entire form and to retrieve only the source code for a button named *btnCancel*:

```

; getObjectNames::pushButton
const
    kRecurse    = Yes
    kNoRecurse  = No
endConst

method pushButton(var eventInfo Event)
var
    foThisForm  Form
endVar

foThisForm.attach() ; Get a handle to the current form.
foThisForm.enumSourceToFile("formSrc.txt", kRecurse)

    btnCancel.enumSourceToFile("btnSrc.txt", kNoRecurse)
endMethod

```

enumUIClasses procedure

UIObject

Writes a list of UIObject classes to a table.

Syntax

```
enumUIClasses ( const tableName String ) Logical
```

enumUIObjectNames method/procedure

Description

enumUIClasses creates a table named *tableName* that contains a list of all UIObject classes (e.g., bitmap, box, and field) and the names of their associated properties. You can specify an alias or path in *tableName*; if an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of the table created by **enumUIClasses**:

Field name	Type	Description
ClassName	A32	Name of object class
PropertyName	A64	Name of property

Example

The following example writes a list of UIObject classes, including their types and properties, to a table named *Tmpclass*:

```
; writeClasses::pushButton
method pushButton(var eventInfo Event)
enumUIClasses("TmpClass.db")
endMethod
```

enumUIObjectNames method/procedure

UIObject

Writes the names of each object in a form to a table.

Syntax

```
enumUIObjectNames ( const tableName String ) Logical
```

Description

enumUIObjectNames writes the names of each object in a form to a table. You can specify an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory.

The following table displays the structure of the table created by **enumUIObjectNames**:

Field name	Type	Description
ObjectName	A128	Name of object
ObjectClass	A32	Type of object (e.g., button)

enumUIObjectNames returns the names of bound objects and unbound objects, beginning with the object that called this method, and including paths to any objects that object contains. To enumerate all objects in a form, make **enumUIObjectNames** start with the form. To enumerate all objects in a page, make it start with the page. To enumerate all objects in a box, make it start with the box.

To write the form's object names to an array, use **enumObjectNames**.

Example

See the **enumObjectNames** example.

enumUIObjectProperties method/procedure

UIObject

Lists the properties of an object.

Syntax

1. enumUIObjectProperties (const tableName String) Logical
2. enumUIObjectProperties (const properties DynArray[] String) Logical

Description

enumUIObjectProperties lists the properties of an object in a table or a dynamic array (DynArray).

Syntax 1 writes the data to the Paradox table specified in *tableName*. You can specify an alias or path in *tableName*. If an alias or path is not specified, Paradox creates *tableName* in the working directory. If the table already exists, Paradox overwrites it without asking for confirmation.

The table lists the properties of the specified object, and the properties of the objects it contains. The following table displays the structure of the table created by **enumUIObjectProperties**:

Field name	Type	Description
ObjectName	A128	Name of the object
PropertyName	A64	Name of the property
PropertyType	A48	Data type of the corresponding property
PropertyValue	A255	Value of the corresponding property

In Syntax 2, the properties of the object (but *not* the properties of objects it contains) are written to a DynArray named *properties*. The DynArray keys are the property names, and the items are the corresponding values. You must declare the DynArray before calling **enumUIObjectProperties**.

Example 1

The following example assumes that *getProperties* is a button on a form designed to display 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 a table named *CstProps*.

```
; getProperties::pushButton
method pushButton(var eventInfo Event)
    enumUIObjectProperties("CstProps.db")
endMethod
```

Example 2

The following example assumes that *getProperties* is a button on a form. The **pushButton** method for *btnProperties* uses **enumUIObjectProperties** to write all of its property values to a dynamic array named *dyn* and then display it.

```
; btnProperties::pushButton
method pushButton(var eventInfo Event)
    var
        dyn    DynArray[] String
    endVar

    self.enumUIObjectProperties(dyn)
    dyn.view("Properties of this button:")
endMethod
```

execMethod method/procedure**UIObject**

Calls a custom method that takes no arguments.

execMethod method/procedure

Syntax

```
execMethod ( const methodName String )
```

Description

execMethod calls the custom method specified by *methodName*. The method specified in *methodName* takes no arguments. Because **execMethod** allows you to call a method based on the contents of a variable, the compiler does not know which method to call until run time.

Example

The following examples assumes that a form contains three fields: *fieldOne*, *fieldTwo*, and *fieldThree*. The form's Var window declares a dynamic array named *objPreProc*, and the form's custom method is named *fieldOnePreProc*. The form's **open** method (which appears in the **isPreFilter=False** clause) creates elements in the *objPreProc* array. An element is created for each object on the form that has a preprocessing custom method.

In the following example, *fieldOne* is assumed to require some preprocessing. An array element is created with the index *pageOne.fieldOne*, and the custom method name *fieldOnePreProc*. The **isPreFilter=True** clause is called for each object on the form to determine whether an array element in *objPreProc* corresponds to the active object. If so, the custom method for that object is called.

The following code attaches to 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
```

The following code goes in 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
```

The following code attaches to 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) ; identify the current target
  targName = targObj.name ; retrieve the name of the target
  forEach element in objPreProc ; iterate through array
  if element = targName then ; is the target name there?
                                ; if so, execute the corresponding
                                ; custom method
    execMethod(objPreProc[targName])
  endIf
endForEach
else
  ; code here executes just for form itself
end
```



```

        ; assign elements to the objPreProc array to indicate
        ; objects for which there is a preprocess custom method
        objPreProc["fieldOne"] = "fieldOnePreProc"
    endIf
endMethod

```

forceRefresh method

UIObject

Instructs an object to display the specified data in the underlying table, and causes a calculated field to recalculate.

Syntax

```
forceRefresh ( ) Logical
```

Description

forceRefresh instructs an object to display the specified data in the underlying table, and causes a calculated field to recalculate. **forceRefresh** also causes a calculated field to recalculate its value, and causes a crosstab or chart to re-evaluate its components.

Calling **active.forceRefresh()** is the same as calling **active.action(DataRecalc)** or pressing SHIFT + F9. **active.forceRefresh()** is a UIObject counterpart to the **forceRefresh** method defined for the TCursor type.

A call to **forceRefresh** affects the target object, objects contained by the target object, and objects bound to the same table as the target object. This method does not affect objects in other windows. For example, calling **forceRefresh** in a form does not refresh data displayed in a table window. Refresh each object in a form by declaring a UIObject variable and calling attach to assign it a value. Do not use a variable declared as a Form variable.

forceRefresh behaves as follows:

- If a table frame or MRO is active when you call **forceRefresh**, only the underlying table refreshes. Child tables repaint, but do not discard cached data.
- If a field object is active when you call **forceRefresh**, the table associated with that field refreshes, and all fields dependent on it are repainted.
- You will not lose your active record position, provided the record still exists in the table.
- On an SQL server, a call to **forceRefresh** forces a read from the server. This is the only way to get a refresh from the server. **forceRefresh** only works on an SQL table if the table has a unique index.

Example

The following example uses **forceRefresh** in code attached to a button's built-in **pushButton** method, allowing the user to control when data is refreshed. This example assumes you have interactively chosen the Database page from the Preferences tabbed dialog box and entered a large value (at least 3,600 seconds) in the Refresh rate dialog box. This code uses **forceRefresh** to refresh the Parts table each time the user clicks the button. Other tables that are bound to this form are refreshed once every 3,600 seconds (one hour).

```

method pushButton(var eventInfo Event)
    Parts.forceRefresh()
endMethod

```

getBoundingBox method**UIObject**

Returns the coordinates of the frame that bounds an object.

Syntax

```
getBoundingBox ( var topLeft Point, var bottomRight Point )
```

Description

getBoundingBox returns the coordinates of the top left corner (*topLeft*) and the bottom right corner (*bottomRight*) of the frame that bounds an object. The coordinates are specified relative to the form. When you select an object in the design window, its bounding box is visible.

Example

The following example draws a box around an ellipse based on the ellipse's bounding box. Assume that a form contains an ellipse named `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
```

getGenFilter method**UIObject**

Retrieves the filter criteria associated with a field, table frame, or multi-record object.

Syntax

```
1. getGenFilter ( criteria DynArray[ ] AnyType ) Logical
2. getGenFilter ( criteria Array[ ] AnyType [ , fieldName Array[ ] AnyType ] ) Logical
3. getGenFilter ( criteria String ) Logical
```

Description

getGenFilter retrieves the filter criteria associated with a field, table frame, or multi-record object. **getGenFilter** assigns them to a DynArray variable (Syntax 1) or to two Array variables (Syntax 2) that you declare and include as arguments. Values are not returned directly.

In Syntax 1, a dynamic array (DynArray) named *criteria* lists fields and filtering conditions as follows: the index is the field name, and the item is the corresponding filter expression.

In Syntax 2, an Array named *criteria* lists filtering conditions, and the optional Array *fieldName* lists corresponding field names. If you omit *fieldName*, conditions apply to fields in the order they appear in the *criteria* array. The first condition applies to the first field in the table, the second condition applies to the second field, and so on.

If the arrays used in Syntax 2 are resizable, **getGenFilter** adjusts the array size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty. If there are more fields than items, this method fills the array and then stops.

In Syntax 3, the filter criteria is assigned to a String variable named *criteria*. You must declare and pass *criteria* as an argument.

Example 1

In the following example, the **pushButton** method for a button named *btnSetFilter* uses **getGenFilter** to populate a dynamic array (DynArray) named *dyn* with a table frame's filter criteria. The code then it examines the DynArray to see if the current criteria filters the Balance Due field for values greater than 10,000 and the Total Invoice field for values less than 65,000 and resets the filter if necessary.

```
;btnSetFilter :: pushButton
method pushButton(var eventInfo Event)
  var
    currentDyn,
    filterDyn   DynArray[] AnyType
    keysAr      Array[] AnyType
  endVar

  filterDyn["Balance Due"]   = " 10000"
  filterDyn["Total Invoice"] = "65000"

  ORDERS.getGenFilter(currentDyn) ; ORDERS is a table frame on a form.

  if currentDyn = filterDyn then
    return ; Filter is OK.
  else
    ORDERS.setGenFilter(filterDyn) ; Reset filter.
  endIf
endMethod
```

Example 2

In the following example, the **pushButton** method for a button named *btnShowFilter* uses **getGenFilter** to populate a dynamic array (DynArray) named *dyn* with the current filter criteria. The code then displays the DynArray in a **view** dialog box. Use this technique as an alternative to setting flags to track the current filter criteria.

```
;btnShowFilter :: pushButton
method pushButton(var eventInfo Event)
  var
    dyn   DynArray[] AnyType
  endVar

  ORDERS.getGenFilter(dyn) ; ORDERS is a table frame on a form.
  dyn.view("Current filter criteria")
endMethod
```

getHTMLTemplate method

UIObject

Returns the HTML string of the UIObject.

Syntax

```
getHTMLTemplate ( ) String
```

getPosition method

Description

getHTMLTemplate generates source HTML for a UIObject and returns it as a String.

Example

In the following example, code in a script window manipulates a customer form by associating a form variable with the form, attaching to a particular field object on the form and generating a string containing the HTML source for this field. The HTML source is displayed in a window. Assume that the customer form contains the customer table in its data model. Fields on the form (including Customer No) are bound to corresponding fields in the table.

```
method run(var eventInfo Event)
var
    fm      Form
    UIO      UIObject
    stHTML   String
endVar

fm.open("customer.fsl")
UIO.attach(fm.Customer_No)      ; attach the UIO variable to the
                                ; Customer_No field of the customer
                                ; table

stHTML=UIO.getHTMLTemplate()
stHTML.view()                   ; displays: <TABLE><TR><TD>Customer No:
                                ; </TD><TD><INPUT TYPE="TEXT"
                                ; NAME="Customer_No"></TD></TR></TABLE>

endMethod
```

getPosition method

UIObject

Reports the position of an object on the screen.

Syntax

```
getPosition ( const x LongInt, const y LongInt, const w LongInt, const h LongInt)
```

Description

getPosition retrieves the position of an object on the screen, relative to its container. Variables x and y specify the coordinates (in twips) of the upper-left corner of the object. Variables w and h specify the object's width and height (in twips). If the object is not specified, *self* is implied.

To ObjectPAL, the screen is a two-dimensional grid, with the origin (0, 0) at the upper-left corner of an object's container, positive x-values extending to the right, and positive y-values extending down.

For dialog boxes and for the Paradox desktop application, the object's position is specified relative to the entire screen; for forms, reports, and table windows, the position is specified relative to the Paradox desktop.

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 button's position. 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 locates the current position of the circle using **getPosition**, and moves it 100 twips to the right using **setPosition**.

The following code attaches 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

```

The following code attaches to *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

```

getProperty method

UIObject

Returns the value of a specified property.

Syntax

```
getProperty ( const propertyName String ) AnyType
```

Description

getProperty 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 each case, use **getPropertyAsString**.

Use **getProperty** when *propertyName* is a variable as an alternative to retrieving a property directly. Otherwise, access the property directly/ The following code displays the syntax for getting property directly:

```
thisColor = myBox.Color
```

Example

The following example creates a dynamic array that is indexed by property names and contains property values. The array's index is used 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"] = ""

```

getPropertyAsString method

```
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
```

getPropertyAsString method

UIObject

Returns the value of a specified property as a string.

Syntax

```
getPropertyAsString ( const propertyName String ) String
```

Description

getPropertyAsString returns a string containing the value of the property specified in *propertyName*.

Example

The following example assigns the value of the Color property to an AnyType variable. 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 (e.g. 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
```

getRange method/procedure

UIObject

Retrieves the values that specify a range for a field, table frame, or multi-record object.

Syntax

```
getRange ( var rangeVals Array[ ] String ) Logical
```

Description

getRange retrieves the values that specify a range for a field, table frame, or multi-record object. This method assigns values to an Array variable that you declare and include as an argument. The following table displays the array values and their range criteria:

Number of array items	Range specification
No items (empty array)	No range criteria is associated with the UIObject

One item	Specifies a value for an exact match on the index's first field
Two items	Specifies a range for the index's first field
Three items	The first item specifies an exact match for the index's first field; items 2 and 3 specify a range for the index's second field
More than three items	For an array of size n, specify exact matches on the index's n-2 fields. The last two array items specify a range for the index's n-1 fields.
	If the array is resizeable, getRange sets the size to equal the number of fields in the underlying table. If fixed-size arrays are used, this method stores as many criteria as it can, starting with criteria field 1. If there are more array items than fields, the remaining items are left empty; if there are more fields than items, this method fills the array and then stops.

Example

The following example uses ObjectPAL to link two unlinked tables in the data model. Assume that a form has the Orders and *Lineitem* tables in its data model and they are not linked. **getRange** is used on a table frame bound to the *Lineitem* table to retrieve the values that specify the current range.

The following code is attached to the record object's **arrive** method of a table frame that is bound to the *Orders* table:

```
;Record :: arrive
method arrive(var eventInfo MoveEvent)
  var
    arSet    Array[] AnyType
    arGet    Array[] AnyType
  endVar

  LINEITEM.getRange(arGet)      ;Retrieve values of range.

  arSet.setSize(2)              ;Specify size of array.
  arSet[1] = string(Order_No.value)
  arSet[2] = string(Order_No.value)

  if (arSet.size() = arGet.size()) and (arSet arGet) then
    LINEITEM.setRange(arSet)    ;Specify range of records.
  endIf
endMethod
```

getRGB procedure

UIObject

Returns the red, green, and blue components of a color.

Syntax

```
getRGB ( const rgb LongInt, var red SmallInt, var green SmallInt, var blue SmallInt )
```

Description

getRGB returns the component red, green and blue components of the color specified in *rgb*. *rgb* is a Colors constant. **getRGB** assigns the component values to the variables *red*, *green*, and *blue*. You must declare and pass the *red*, *green*, and *blue* variables as arguments.

Example

The following example determines the red, green, and blue components of the constant Brown.

hasMouse method

```
; 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
```

hasMouse method

UIObject

Determines whether the pointer is positioned over an object.

Syntax

```
hasMouse ( ) Logical
```

Description

hasMouse returns True if the pointer is positioned within the boundaries of an object; otherwise, it returns False.

Example

The following example assumes that a form has a bitmap object named *cat*. The **open** method for *cat* sets a timer interval to 250 milliseconds. The **timer** method uses **hasMouse** to determine if the mouse is within *cat*'s boundaries; if not, it moves *cat* to the mouse's position.

The following code attaches to *cat*'s **open** method:

```
; cat::open
method open(var eventInfo Event)
; set the timer interval to 250 milliseconds
self.setTimer(250)
endMethod
```

The following code attaches to *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 Toolbar
endif
endMethod
```

home method

UIObject

Moves to the first record in a table.

Syntax

```
home ( ) Logical
```

Description

home sets the active record to the first record in a table. **home** respects the limits of restricted views that are 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. This means that the following statements are equivalent:

```
obj.home()
obj.action(DataBegin)
```

Example

The following example moves to the first record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form, and *moveToHome* is a button on the 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
```

insertAfterRecord method**UIObject**

Inserts a record below the active record in a table.

Syntax

```
insertAfterRecord ( ) Logical
```

Description

insertAfterRecord inserts a record below the active record in a table. The table must be in Edit mode.

Example

The following example assumes that *CustSort* is a copy of the *Customer* table that has been sorted by the Name field. The form in this example contains a table frame named *CUSTSORT* that is bound to the *CustSort* table, an undefined field named *newField*, and a button named *insRecButton*. To add a 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 determines if a value has been added to *newField* and if the form is in Edit mode. The method attaches the TCursor *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If it detects a name greater than the new name, the method uses **insertRecord** to add a blank record before the name found; otherwise, it uses **insertAfterRecord** to insert a new blank record to 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
```

insertBeforeRecord method

```
nameStr = newField.Value           ; Get the name to add.
CUSTSORT."Name".moveTo()

if CUSTSORT.isEdit() then          ; Check for edit mode first.
    custTC.attach(CUSTSORT)

    scan custTC for custTC."Name" = nameStr:
        quitloop                  ; Stop when you find the name.
    endscan

    msgInfo("Active record no", custTC.recno())
    CUSTSORT.resync(custTC)        ; Resync CUSTSORT to custTC.

    if NOT CUSTSORT.atLast() then
        CUSTSORT.insertBeforeRecord()
    else
        CUSTSORT.insertAfterRecord() ; Add blank record.
    endIf

    CUSTSORT.Name=newField.Value
    CustSort.postRecord()
    msgInfo("New name added", "Please enter remaining customer information")

else
    msgInfo("Sorry", "Form must be in Edit mode.")
endIf
endMethod
```

insertBeforeRecord method

UIObject

Inserts a record above the active record in a table.

Syntax

```
insertBeforeRecord ( ) Logical
```

Description

insertBeforeRecord inserts a record above the active record in a table. The table must be in Edit mode.

insertBeforeRecord has the same effect as the action constant `DataInsertRecord`. This means the following statements are equivalent:

```
obj.insertBeforeRecord()
obj.action(DataInsertRecord)
```

Example

The following example assumes that *CustSort* is a copy of the *Customer* table that has been sorted by the Name field. The form contains a table frame named *CUSTSORT* that is bound to *CustSort*, an undefined field named *newField*, and a button named *insRecButton*. To add a name to the table, type the name in *newField* and press *insRecButton*.

The following method overrides the **pushButton** method for *insRecButton*. This method determines if a value has been added to *newField* and if the form is in Edit mode. The method attaches a TCursor named *custTC* to *CUSTSORT*, and scans *custTC* for a value greater than the string given in *newField*. If the method detects a name greater than the new name, the method uses **insertBeforeRecord** to

insert a 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("Active record no", custTC.recno())
    CUSTSORT.resync(custTC)           ; Resync CUSTSORT to custTC.

    if NOT CUSTSORT.atLast() then
        CUSTSORT.insertBeforeRecord()
    else
        CUSTSORT.insertAfterRecord()
    endif

    ; ... fill the record with the rest of the customer information
; Put new name in the field of the tableframe and post. Inform user.
CUSTSORT.Name=newField.Value
CustSort.postRecord()

msgInfo("New name added", "Please enter remaining customer information")

else
    msgInfo("Sorry", "Form must be in Edit mode.")
endif
endMethod

```

insertRecord method

UIObject

Inserts a record before the active record in a table.

Syntax

```
insertRecord ( ) Logical
```

Description

insertRecord inserts a record before the active record in a table.

insertRecord has the same effect as **insertBeforeRecord** and the action constant **DataInsertRecord**. This means the following three statements are equivalent:

isAssigned method

```
obj.insertRecord()  
obj.insertBeforeRecord()  
obj.action(DataInsertRecord)
```

Example

See the **insertBeforeRecord** example.

isAssigned method

UIObject

Reports whether a variable has been assigned a value.

Syntax

```
isAssigned ( ) Logical
```

Description

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

Note

- This method works for many ObjectPAL types, not just UIObject.

Example

The following 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 is attached 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
```

isContainerValid procedure

UIObject

Reports whether an object's container is valid.

Syntax

```
isContainerValid ( ) Logical
```

Description

isContainerValid reports if the active object's container is valid. For example, if a form does not have a container the **ContainerName** property for a form is not valid.

Example

In the following example, the **arrive** built-in event method for a form uses **isContainerValid** to search for a valid 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

```

isEdit method

UIObject

Reports whether an object is in Edit mode.

Syntax

```
isEdit ( ) Logical
```

Description

isEdit reports whether an object is in Edit mode.

Example

See the **lockRecord** example.

isEmpty method

UIObject

Reports whether a table contains records.

Syntax

```
isEmpty ( ) Logical
```

Description

isEmpty returns True if none of the table's records 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 determine if a table is empty by determining the value returned by the **NRecords** method or the value of the object's **NRecords** property.

Example

The following example uses the *cascadeDelete* button to delete an order and all the linked detail records for that order. Assume that a form contains a single-record object that is bound to the *Orders* tables and a linked table frame that is 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
  endif
endif

```

isLastMouseClickedValid method

```
        endwhile
        ORDERS.action(DataDeleteRecord)    ; delete the master record
    endif
else
    msgInfo("Status", "You must be editing to delete a record.")
endif
endMethod
```

isLastMouseClickedValid method

UIObject

Reports whether the last object clicked is valid.

Syntax

```
isLastMouseClickedValid ( ) Logical
```

Description

isLastMouseClickedValid reports whether the active form has been clicked since it opened.

Example

The following example determines whether a form has been clicked:

```
; 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
```

isLastMouseRightClickedValid method

UIObject

Reports whether the last object right-clicked is valid.

Syntax

```
isLastMouseRightClickedValid ( ) Logical
```

Description

isLastMouseRightClickedValid reports whether the current form has been right-clicked since it opened.

Example

The following example determines whether a form has been right-clicked:

```
; 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
```

isRecordDeleted method**UIObject**

Reports whether the active record has been deleted (dBASE tables only).

Syntax

```
isRecordDeleted ( ) Logical
```

Description

isRecordDeleted reports whether the active record has been deleted. **isRecordDeleted** only works for dBASE tables. Deleted Paradox records can't be displayed. This method returns True if the active record has been deleted; otherwise, it returns False.

For **isRecordDeleted** to work correctly, you must call **showDeleted** (TCursor type) to display deleted records in the table; otherwise, deleted records are not visible to **isRecordDeleted**.

Example

See the **isRecordDeleted** (TCursor type) example.

keyChar method**UIObject**

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

keyChar creates an event and to call the object's built-in **keyChar** event method. Specify one or more characters in *characters* (Syntax 1), in *ansiKeyValue* (Syntax 2), or in *ansiKeyValue* and *vChar* (Syntax 3). Specify the keyboard state in state using KeyboardStates constants. You can add these constants together to create combined key states (e.g., ALT + CTRL).

Example

The following example overrides the **pushButton** method of a button named *sendKeyChar*. This method sends keystrokes *fieldOne* on the 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
```

keyPhysical method**UIObject**

Sends an event to an object's built-in **keyPhysical** method.

killTimer method

Syntax

```
keyPhysical ( const aChar SmallInt, const vChar SmallInt, const state SmallInt )
```

Description

keyPhysical sends an event to an object's built-in **keyPhysical** method. Specify the ANSI character code in *aChar*, the virtual key code in *vChar*, and the keyboard *state* in state using KeyboardStates constants.

Example

In the following example, code is attached to the **pushButton** method of a button named *sendKeyPhys*. This method sends the character a to *fieldOne*.

```
; sendKeyPhys::pushButton
method pushButton(var eventInfo Event)
    fieldOne.keyPhysical(97, 97, Shift)    ; send an "a"
endMethod
```

killTimer method

UIObject

Stops the timer associated with an object.

Syntax

```
killTimer ( )
```

Description

killTimer stops the 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 respectively. When the timer starts, it issues a TimerEvent every 100 milliseconds. Each TimerEvent causes *toggleButton*'s **timer** method to execute. The **timer** method uses **getPosition** to retrieve the current position of the ellipse and uses **setPosition** to move it 100 twips to the right.

The following code is attached to *toggleButton*'s **pushButton** method:

```
; toggleButton::pushButton
method pushButton(var eventInfo Event)
if buttonLabel = "Start Timer" then    ; if stopped and 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
```

The following code is attached to *toggleButton*'s **timer** method. FloatCircle is a circle UI Object on the form:

```
; 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

```

locate method

UIObject

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

locate searches a table frame, multi-record object, record object, or field object for record values that match one or more field/value pairs. Specify the search value in *exactMatch* and the search field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). When possible, **locate** uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

If a match is found, the cursor moves to that record. This operation fails if the active record cannot be posted and unlocked (e.g., due to a key violation). If no match is found, the cursor returns to the active record. The search always starts from the beginning of the table.

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example

The following example assumes that a form contains a table frame bound to the *Customer* table and a button named *locateButton*. The **pushButton** method for *locateButton* searches for the customer named Sight Diver in the city named Kato Paphos. If a match is 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

```

locateNext method

UIObject

Searches forward from the active record for a specified field value.

locateNextPattern

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

locateNext searches a table for record values that match one or more field/value pairs. Specify the search value in *exactMatch* and the search field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). When possible, **locateNext** uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record after the active record. If a match is found, the cursor moves to that record. This operation fails if the active record cannot be posted and unlocked (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search from the beginning of a table, use **locate**.

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example

The following example assumes that a form contains a table frame bound to the *Customer* table and a button named *locateButton*. The **pushButton** method for *locateButton* searches for customers in the city of Freeport. If **locate** is successful, the code 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

endmethod
```

locateNextPattern

UIObject

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

locateNextPattern finds strings or substrings (e.g., comp in computer). When possible, this method uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record after the active record. If a match is found, the cursor moves to that record. This operation fails if the active record cannot be committed (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search from the beginning of a table, use **locatePattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard pattern operators @ and .. in the pattern argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single character. Any combination of literal characters and wildcards can be used to construct a search. If **advancedWildCardsInLocate** (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example 1

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. Assume that the *CUSTOMER* table frame is bound to *Customer*, and that *locateButton* is a button on the 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
```

locatePattern method

```
endIf  
endMethod
```

Example 2

The following example searches for records by the value in the City field and the 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
```

locatePattern method

UIObject

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  
   fieldName SmallInt, const pattern String ) Logical
```

Description

locatePattern finds strings or substrings (e.g., comp in computer). When possible, this method uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record after the active record. If a match is found, the cursor moves to that record. This operation fails if the active record cannot be committed (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search from the beginning of a table, use **locatePattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard *pattern* operators @ and .. in the pattern argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single

character. Any combination of literal characters and wildcards can be used to construct a search. If **advancedWildCardsInLocate** (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

To start a search from the beginning of a table, use **locateNextPattern**.

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example

See the **locateNextPattern** example.

locatePrior method

UIObject

Searches backward from the active record 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

locatePrior searches backwards from the active record in a table for record values that match one or more field/value pairs. Specify the search value in *exactMatch* and the search field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance). When possible, **locateNext** uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record before the active record and moves up through the table. If a match is found, the cursor moves to that record. This operation fails if the active record cannot be posted and unlocked (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search from the beginning of a table, use **locate**.

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example

The following example locates the last occurrence of a value in a table by searching up from the end of the table using **locatePrior**. Assume that the form contains a table frame that is bound to the *Customer* table, and a 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
```

locatePriorPattern method

```
msgInfo("Status", "The last record with a City of " +
        searchFor + " is record " + Cust.recno + ".")
endIf

endMethod
```

locatePriorPattern method

UIObject

Searches backward from the active record for a field that contains 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

locatePriorPattern finds strings or substrings (e.g., comp in computer). When possible, this method uses active indexes to speed the search. This method respects the limits of restricted views in linked detail tables.

The search begins with the record after the active record. If a match is found, the cursor moves to that record. This operation fails if the active record cannot be committed (e.g., due to a key violation). If no match is found, the cursor returns to the active record. To start a search at the beginning of a table, use **locatePattern**.

To search for records by the value of a single field, specify the field in *fieldName* or *fieldNum* (use *fieldNum* for faster performance) and specify a pattern of characters in *pattern*.

You can include the standard pattern operators @ and .. in the pattern argument. The .. operator specifies any string of characters (including no string). The @ operator specifies for any single character. Any combination of literal characters and wildcards can be used to construct a search. If **advancedWildCardsInLocate** (Session type) is enabled, you can use advanced match pattern operators. For more information, see the description of **advMatch**.

To search for records by the values of more than one field, specify exact matches on all fields except the last one in the list. For example, the following code searches the Name field for exact matches on the word Corel, the Product field for Paradox, and the Keywords field for words beginning with data (e.g., database).

To start a search from the beginning of a table, use **locateNextPattern**.

```
tc.locateNextPattern("Name", "Corel" "Product", "Paradox" "Keywords", "data..")
```

Note

- The search is case-sensitive unless **ignoreCaseInLocate** (Session type) is enabled.

Example

The following example locates the last occurrence of a value in a table by searching up from the end of the table and using **locatePriorPattern**. Assume that the form contains a table frame that is bound to the *Customer* table, and a 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

```

lockRecord method

UIObject

Puts a write lock on the active record.

Syntax

```
lockRecord ( ) Logical
```

Description

lockRecord returns True if it places a write lock on the active record; otherwise, it returns False.

Note

- The Locked property is a read-only property. This means that you can't change the property setting to lock or unlock an object.

Example 1

The following example determines whether the *Customer* table is in Edit mode. If it is, the method locates a record, attempts to lock it using **lockRecord** and determines the status of the lock using **recordStatus**. Assume that a form contains a table frame that is bound to the *Customer* table, and a button named *lockButton*. 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 thisForm.editing then
    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
endif
endMethod

```

Example 2

The following example examines a record object's Locked property:

lockStatus method

```
; 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
```

lockStatus method

UIObject

Returns the number of locks on a table.

Syntax

```
lockStatus (const lockType String ) SmallInt
```

Description

lockStatus returns the number of locks of type *lockType* on a table. *lockType*'s value is Write, Read, or Any.

If you haven't placed any locks on the table **lockStatus** returns 0.

If you specify Any for *lockType*, **lockStatus** returns the total number of locks you've placed on the table. **lockStatus** does not include locks placed by Paradox or by other users or applications.

Example

The following example assumes that a form has a table frame named *CUSTOMER* that is bound to the *Customer* table, and a button named *lockButton*. The **pushButton** method for *lockButton* removes all locks from *CUSTOMER*, searches for locks using **lockStatus**, places a lock and reports on the locks using **lockStatus**.

```
; 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

```

menuAction method

UIObject

Sends an event to an object's **menuAction** method.

Syntax

```
menuAction ( const action SmallInt ) Logical
```

Description

menuAction constructs a MenuEvent and sends it to a specified UIObject's **menuAction** method. *action* is one of the MenuCommands constants, or a user-defined menu constant.

Note

- You can't use menuAction to simulate a File menu command. To send a menu command constant that is equivalent to a File menu command, use one of the regular Action constants, manipulate a property, or use a System type method.

Example

The following example uses the *sendATile* button on the current form to send *thisForm* a MenuWindowTile action.

```

; sendATile::pushButton
method pushButton(var eventInfo Event)
thisForm.menuAction(MenuWindowTile)
endMethod

```

methodDelete method

UIObject

Deletes a specified method or event.

Syntax

```
methodDelete ( const methodName String ) Logical
```

Description

methodDelete deletes the method or event specified by *methodName*. The form that contains the object must be in a Form Design window.

Example

The following example uses **methodGet**, **methodSet**, and **methodDelete** to copy methods from one object to another. The code overrides the **pushButton** method for a button named *copyMethods*. The form contains four other objects. 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. The final object is a radio button field named *copyOrMove* which specifies whether methods in the source are copied, or copied then deleted.

methodDelete method

```
; copyMethods::pushButton
method pushButton(var eventInfo Event)
var
  otherForm          Form          ; a handle to a form
  sourceObj,         UIObject      ; 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")
    else
        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
    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

```

methodEdit method

UIObject

Opens an object's method or event in an Editor window.

Syntax

```
methodEdit (const methodName String) Logical
```

Description

methodEdit opens the method or event specified by *methodName* in an Editor window. If you specify a method or event that doesn't exist, **methodEdit** will create it for you. **methodEdit** fails if you try to open a method that is running.

Example

The following example opens the object's **testMethod** method in an Editor window:

```

method pushButton(var eventInfo Event)
var
    MyForm form
    MyObject uiobject
endvar

MyForm.load("vendors.fsl")
MyObject.attach(MyForm,"Preferred")
MyObject.methodEdit("testMethod")

endMethod

```

methodGet method

UIObject

Returns the text of a specified method or event.

methodSet method

Syntax

```
methodGet ( const methodName String ) String
```

Description

methodGet returns the text of the method or event specified in *methodName*.

Example

See the **methodDelete** example.

methodSet method

UIObject

Sets the text of a specified method or event.

Syntax

```
methodSet ( const methodName String, const methodText String ) Logical
```

Description

methodSet specifies the source code for the method or event named in *methodName*. Open the form that contains the object in a Form Design window.

Note

- The method specified by *methodname* does not need to previously exist in the form.

Example

See the **methodDelete** example.

mouseClick method

UIObject

Sends an event to an object's **mouseClick** method.

Syntax

```
mouseClick ( ) Logical
```

Description

mouseClick constructs a **mouseClick** MouseEvent to call the object's built-in **mouseClick** event method.

Example

The following example sends a **mouseClick** MouseEvent to *fieldTwo* on the form:

```
; sendMouseClicked::pushButton
method pushButton(var eventInfo Event)
; send a mouseClicked to fieldTwo
fieldTwo.mouseClick()
endMethod
```

mouseDouble method

UIObject

Sends an event to an object's **mouseDouble** method.

Syntax

```
mouseDouble ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDouble constructs a double-click event to call the object's built-in **mouseClick** event method. The event's coordinates are specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using `KeyboardStates` constants. You can add these constants together to create combined key states (e.g., `CTRL + Left Arrow` key).

Example

The following example sends a double-click to *fieldTwo* on the form:

```
; sendMouseDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseDouble to fieldTwo
fieldTwo.mouseDouble(100, 100, LeftButton)
endMethod
```

mouseDown method**UIObject**

Sends an event to an object's **mouseDown** method.

Syntax

```
mouseDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseDown constructs an event to call the object's built-in **mouseDown** event method. The event's coordinates are specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using `KeyboardStates` constants. You can add these constants together to create combined key states (e.g., `Left Arrow key + CTRL`).

Example

The following example sends a **mouseDown** and a **mouseUp** `MouseEvent` to the object *fieldOne* on the 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
```

mouseEnter method**UIObject**

Sends an event to an object's **mouseEnter** method.

Syntax

```
mouseEnter ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseEnter constructs an event to call the object's built-in **mouseEnter** event method. The event's coordinates are specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using `KeyboardStates` constants. You can add these constants together to create combined key states (e.g., `Left Arrow key + CTRL`).

mouseExit method

Example

The following example sends a **mouseEnter** MouseEvent to a field named *fieldSix* on the form:

```
; sendMouseEnter::pushButton
method pushButton(var eventInfo Event)
; send a mouseEnter to fieldSix
fieldSix.mouseEnter(100,100,LeftButton)
endMethod
```

mouseExit method

UIObject

Sends an event to an object's **mouseExit** method.

Syntax

```
mouseExit ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseExit constructs an event to call the object's built-in **mouseExit** event method. The event's coordinates are specified in x and y (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Example

The following example sends a **mouseExit** MouseEvent to *fieldSeven* on the form:

```
; sendMouseExit::pushButton
method pushButton(var eventInfo Event)
; send a mouseExit to fieldSeven
fieldSeven.mouseExit(100, 100, LeftButton)
endMethod
```

mouseMove method

UIObject

Sends an event to an object's **mouseMove** method.

Syntax

```
mouseMove ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseMove constructs an event to call the object's built-in **mouseMove** event method. The event's coordinates are specified in x and y (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Example

The following example sends a **mouseDown**, a **mouseUp**, and a **mouseMove** MouseEvent to a field named *fieldFive* on the 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
```

mouseRightDouble method

UIObject

Sends an event to an object's **mouseRightDouble** method.

Syntax

```
mouseRightDouble ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseRightDouble constructs an event to call the object's built-in **mouseRightDouble** event method. The event's coordinates are specified in x and y (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Tip

- You should disable the **mouseRightUp** default of the UIObject to which the **mouseRightDouble** method is attached in order for this method to work.

Example

The following example sends a **mouseRightDouble** MouseEvent to a field named *fieldTwo* on the form. The **mouseRightUp** default for *fieldTwo* should be disabled in order for this method to work.

```
; sendMouseRightDouble::pushButton
method pushButton(var eventInfo Event)
; send a mouseRightDouble to fieldTwo
fieldTwo.mouseRightDouble(100, 100, LeftButton)
endMethod
```

mouseRightDown method

UIObject

Sends an event to an object's **mouseRightDown** method.

Syntax

```
mouseRightDown ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseRightDown constructs an event to call the object's built-in **mouseRightDown** event method. The event's coordinates are specified in x and y (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Example

The following example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the 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 and send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

mouseRightUp method

UIObject

Sends an event to an object's **mouseRightUp** method.

Syntax

```
mouseRightUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseRightUp constructs an event to call the object's built-in **mouseRightUp** event method. The event's coordinates are specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Example

The following example sends a **mouseRightDown** and a **mouseRightUp** MouseEvent to a field named *fieldThree* on the form:

```
; sendMouseRightUp::pushButton
method pushButton(var eventInfo Event)
var
    fPt Point
endVar
fPt = fieldThree.position      ; get the position, send a mouseRightDown
fieldThree.mouseRightDown(fPt.x(), fPt.y(), LeftButton)
sleep(500)                    ; pause and send a mouseRightUp
fieldThree.mouseRightUp(fPt.x(), fPt.y(), LeftButton)
endMethod
```

mouseUp method

UIObject

Sends an event to an object's **mouseUp** method.

Syntax

```
mouseUp ( const x LongInt, const y LongInt, const state SmallInt ) Logical
```

Description

mouseUp constructs an event to call the object's built-in **mouseUp** event method. The event's coordinates are specified in *x* and *y* (in twips). Specify the mouse and keyboard state in *state* using KeyboardStates constants. You can add these constants together to create combined key states (e.g., Left Arrow key + CTRL).

Example

The following example sends a **mouseDown** and a **mouseUp** MouseEvent to the object *fieldOne* on the 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
```


moveTo method

UIObject

Sets the focus to a specified object.

Syntax

1. (Method) `moveTo ()` Logical
2. (Procedure) `moveTo (const objectName String)` Logical

Description

moveTo moves the focus to a specified object. When you call **moveTo** as a procedure (Syntax 2), *objectName* specifies the destination object (the object to which the focus is moved).

Note

- If **moveTo()** succeeds, then the Focus property for the destination object is set to True, and the SetFocus event is executed.

Example

The following example assumes that a form contains a table frame that is bound to *Orders*, and another table frame that is bound to *LineItem*. Orders has a one-to-many link to *LineItem*. The form also contains a button named *findDetails*. In this example, the **pushButton** method for *findDetails* searches the entire table for orders that include the current part number.

The following 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
```

The following code is attached to *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
```

The following code is attached to *findDetails*' **close** method:

moveToRecNo method

```
; findDetails::close
method close(var eventInfo Event)
lineTC.close() ; close the TCursor to LineItem
endMethod
```

moveToRecNo method

UIObject

Moves to a specific record in a dBASE table.

Syntax

```
moveToRecNo ( const recordNum LongInt ) Logical
```

Description

moveToRecNo sets the active record to *recordNum*. This method returns an error if *recordNum* is not in the table. Use **nRecords** or examine the **NRecords** property to determine the number of records in a table. Use **moveToRecNo** only for dBASE tables. Use **moveToRecord** for Paradox tables.

Example

The following example moves to the middle record in a table. Assume that a form contains a table frame that is bound to the *LineItem* table, and a button named *MidWay*.

```
; MidWay::pushButton
method pushButton(var eventInfo Event)
var
    halfWay LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecNo(halfWay)

endMethod
```

moveToRecord method

UIObject

Moves to a specific record in a table.

Syntax

1. moveToRecord (const recordNum LongInt) Logical
2. moveToRecord (const tc TCursor) Logical

Description

moveToRecord moves to a specific record in a table.

Syntax 1 moves to the record number specified in *recordNum*. This method 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 determine the number of records in a table.

Syntax 2 moves to the record pointed to by the TCursor *tc*. Use **moveToRecNo** to accelerate performance in dBASE tables.

Example

The following example moves to the middle record of a table. Assume that the form contains a table frame that is bound to the *LineItem* table, and a button named *MidWay*. For an example of how to use **moveToRecord** using a TCursor, see the **moveTo** example.

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

```

    halfWay LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endMethod

```

nextRecord method

UIObject

Moves to the next record in a table.

Syntax

```
nextRecord ( ) Logical
```

Description

nextRecord moves to the next record in a table. This method returns an error if the cursor is already at the last record.

nextRecord has the same effect as the action constant `DataNextRecord`. This means that the following statements are equivalent:

```
obj.nextRecord()
obj.action(DataNextRecord)
```

Example

The following example moves to the next record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form and that *moveToNext* is a button on the 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

```

nFields method

UIObject

Returns the number of fields in a table.

Syntax

```
nFields ( ) LongInt
```

Description

nFields returns the number of fields in a table. To determine the number of columns displayed in an object that is bound to a table, examine the value of the `NCols` property for that object.

Example

The following example returns the number of fields and key fields in the *LineItem* table. Assume that a form has a table frame named *LINEITEM* that is bound to the *LineItem* table, and a button named *tableStats*.

```

; tableStats::pushButton
method pushButton(var eventInfo Event)

```

nKeyFields method

```
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
```

nKeyFields method

UIObject

Returns the fields in the active index.

Syntax

```
nKeyFields ( ) LongInt
```

Description

nKeyFields returns the number of fields in the index associated with a UIObject.

Example

See the **nFields** example.

nRecords method

UIObject

Returns the number of records in a table.

Syntax

```
nRecords ( ) LongInt
```

Description

nRecords returns the number of records in a table that is bound to a table frame, multi-record object, or field object. You can also examine an object's **NRecords** property to determine the number of records in the table bound to that object. Both operations are time consuming for dBASE tables and large Paradox tables.

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 table.

For a Paradox table, **nRecords** returns the number of records in the underlying table - not the number of records displayed in the object. For example, if the *Customer* table contains 100 records and a table frame that is bound to the *Customer* table displays 5 records, this method would return 100, not 5.

For a dBASE table, **nRecords** counts deleted records if they are displayed in the form. To make a form display deleted records, choose Form, Show Deleted, or call **action(DataShowDeleted)** or **action(DataToggleDeleted)**.

Note

- When you call **nRecords** after setting a filter, the returned value does not represent the number of records in the filtered set. To retrieve the number of records in the filtered set, attach a TCursor to the UIObject and call **cCount**. When you call **nRecords** after setting a range, the returned value represents the number of records in the set that are defined by the range.

Example

The following example moves to the middle record in a table. Assume that a form contains a table frame named *LINEITEM* that is bound to the *LineItem* table, and a button named *MidWay*.

```

; MidWay::pushButton
method pushButton(var eventInfo Event)
var
    halfWay    LongInt
endVar

halfWay = LongInt(LINEITEM.nRecords()/2)
LINEITEM.moveToRecord(halfWay)

endMethod

```

pixelsToTwips method

UIObject

Converts screen coordinates from pixels to twips.

Syntax

```
pixelsToTwips ( const pixels Point ) Point
```

Description

pixelsToTwips converts the screen coordinates from pixels to twips. A pixel is a dot on the screen, and a twip is a unit equal to 1/1440 of a logical inch (1/20 of a printer's point).

Example

The following example assumes that a form contains a two-inch square box named *twoSquare*. The *twoSquare* box contains two text boxes: *pixNum* and *twipNum*. *pixNum* displays the width of the box in pixels and *twipNum* displays the width of the box 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

```

postAction method

UIObject

Posts an action to an action queue for delayed execution.

Syntax

```
postAction ( const actionId SmallInt )
```

Description

postAction posts an action to an action queue for delayed execution. This method works like **action**, except that the action is not executed immediately. Instead, the action is posted to an action queue

postRecord method

when the method is called. Paradox waits until a yield occurs (e.g., the current method completes execution or calls **sleep**).

The value of *actionID* can be a user-defined action constant or a constant from one of the following Action classes:

- ActionDataCommands
- ActionEditCommands
- ActionFieldCommands
- ActionMoveCoomands
- ActionSelectCommands

Example

The following example demonstrates how to store a value from a calculated field in a table. In this example, an unbound calculated field object named *fldLineTotal* calculates the line total. Whenever the calculation occurs, **postAction** sends a custom user action. This custom user action posts the value to a table frame that is bound to the *Lineitem* table.

The following code defines the calculation for the calculated field.

```
;fldLineTotal :: Calculation
[LINEITEM.SELLING PRICE]*[LINEITEM.QTY] ;Calculated field.
```

The following code is attached to the field object's built-in **newValue** method.

```
;fldLineTotal :: newValue
method newValue(var eventInfo Event)
  if Qty.isEdit() then ;If edit mode,
    Qty.postAction(UserAction + 1) ;send a custom user
  endIf ;action to QTY.
endmethod
```

The following code is attached to the table frame's built-in **action** method.

```
;recTFrame :: action
method action(var eventInfo ActionEvent)
  if eventInfo.id() = UserAction + 1 then ;If ID is user
    dmPut("LINEITEM", "Total", Total.value) ;action and
    Qty.postRecord() ;post changes.
  endIf
endmethod
```

postRecord method

UIObject

Posts a pending record to a table.

Syntax

```
postRecord ( ) Logical
```

Description

postRecord returns True if the active 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. This means that the following statements are equivalent:

```
obj.postRecord()
obj.action(DataPostRecord)
```

Example

The following example locates a record, uses **lockRecord** to lock it, and determines the status of the lock using *recordStatus*. The code changes the record and posts it using **postRecord**. Assume that a form contains a table frame that is 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
  msgInfo("Status", "You must be in edit mode to lock and change records.")
endIf
endMethod
```

priorRecord method**UIObject**

Moves to the previous record in a table.

Syntax

```
priorRecord ( ) Logical
```

Description

priorRecord moves to the previous record in a table. This method returns an error if the cursor is already at the first record.

priorRecord has the same effect as the action constant *DataPriorRecord*. This means that the following statements are equivalent:

```
obj.priorRecord()
obj.action(DataPriorRecord)
```

Example

The following example moves to the previous record in the *Customer* table. Assume that *Customer* is bound to a table frame on the form and that *moveToPrior* is a button on the 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.")
endMethod
```

pushButton method

```
endIf  
endMethod
```

pushButton method

UIObject

Generates a **pushButton** event and sends it to an object.

Syntax

```
pushButton ( ) Logical
```

Description

pushButton creates a **pushButton** event to call the object's built-in **pushButton** method of an object with that event.

Example

The following example sends a **pushButton** event to *buttonTwo* on the form:

```
; sendPushButton::pushButton  
method pushButton(var eventInfo Event)  
; send a pushButton to buttonTwo  
buttonTwo.pushButton()  
endMethod
```

recordStatus method

UIObject

Reports the status of a record.

Syntax

```
recordStatus ( const statusType String ) Logical
```

Description

recordStatus returns True or False answers to a question about the status of a record. Use the argument *statusType* to specify the status in question (i.e., is New, Locked, or Modified).

The New value means the record has just been added to the table. Locked means that an implicit or explicit lock has been placed on the record. Modified means at least one of the field values has been changed. You can also obtain information about the active record by examining the Inserting, Locked, Focus, and Touched properties for the record.

Example

The following example locates a record, attempts to lock it using **lockRecord** and determines the status of the lock using **recordStatus**. The method changes the record and unlocks it using **unlockRecord**. Assume that a form contains a table frame that is bound to the *Customer* table and a button named *lockButton*. The record object of the table frame is named *custRec*.

```
; lockButton::pushButton  
method pushButton(var eventInfo Event)  
var  
    Cust    UIObject  
    newKey Number  
endVar  
  
Cust.attach(CUSTOMER)          ; attach to CUSTOMER table frame  
Cust.locate("Name", "Sight Diver") ; find the record  
if NOT isEdit() then           ; check if form is in Edit mode  
    msgInfo("Status", "You must be in Edit mode for this operation.")  
else  
end
```



```

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.value = newKey    ; change the key value
    Cust.custRec.Customer_No.action(EditCommitField)
    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.value = 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

```

resync method

UIObject

Resynchronizes an object to a TCursor.

Syntax

```
resync ( const tc TCursor ) Logical
```

Description

resync changes the active record pointer of a UIObject to the active record of a TCursor named *tc*. When you resynchronize a table object to a TCursor, the table's filters and indexes are changed to those of the TCursor. A dBASE table also takes the Show Deleted setting of the TCursor.

Note

- **resync** only applies when the UIObject and the TCursor are associated with the same table.

Example

See the **insertBeforeRecord** example.

rgb procedure

UIObject

Defines a color.

Syntax

```
rgb ( const red SmallInt, const green SmallInt, const blue SmallInt ) LongInt
```

Description

rgb defines a color using *red*, *green*, and *blue*, which can be integers ranging from 0 to 255, or Colors constants.

sendToBack method

Example

The following example uses **rgb** to set the color of boxes as they're created. The code also creates a color palette. Assume that the titles exist on the form in the appropriate locations. The form has a 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 = 720              ; amount to move horizontally (twips)
    vertInc = 540               ; 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
    endfor
endfor
endMethod
```

sendToBack method

UIObject

Displays an object behind other objects.

Syntax

sendToBack ()

Description

sendToBack moves a UIObject to a window's back drawing layer, displaying it behind other objects. If the UIObject is a form, this method displays the form window behind other windows. **sendToBack** works in design mode and run mode and you do not have to select the object. Use **sendToBack** if

- you have objects that overlap each other
- you want to rearrange the tab order

When you change the position of an object, you also change its tab order. An object always tabs from back to front.

Example

The following example assumes that a form contains two multi-record objects that occupy the same location and size. Two buttons toggle between each multi-record object: *btnShowVendors* and *btnShowStock*. *btnShowVendors* uses **sendToBack** to send the *STOCK* multi-record object to the background; the *VENDORS* multi-record object is in front. *btnShowStock* uses **sendToBack** to send the *VENDORS* multi-record object to the background; the *STOCK* multi-record object is in front.

The following code is attached to *btnShowVendors*.

```
;btnShowVendors :: pushButton
method pushButton(var eventInfo Event)
    STOCK.sendToBack() ; Send the VENDORS MRO to the back
    Vendor_No.moveTo() ; so the STOCK MRO may be seen.
endmethod
```

The following code is attached to *btnShowStock*.

```
;btnShowStock :: pushButton
method pushButton(var eventInfo Event)
    VENDORS.sendToBack() ; Send the STOCK MRO to the back
    Stock_No.moveTo() ; so the VENDORS MRO may be seen.
endmethod
```

setGenFilter method**UIObject**

Specifies conditions for including records in a field, table frame, or multi-record object.

Syntax

```
1. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria DynArray [ ]
AnyType ) Logical
2. setGenFilter ( [ idxName String, [ tagName String, ] ] criteria Array[ ] AnyType [
, fieldId Array[ ] AnyType ] ) Logical
```

Description

setGenFilter specifies conditions for including records in a field, table frame, or multi-record object. Records that meet the specified conditions are included, and all remaining records are filtered out. Unlike **setRange**, this method does not require an indexed table. **setGenFilter** must be executed before opening a table using a TCursor.

In Syntax 1, a dynamic array (DynArray) named *criteria* specifies the index as the field name or number, and the item as the criteria expression. For example, the following code specifies criteria based on the values of three fields:

```
; The value of the first field in the table is Widget.
criteriaDA[1] = "Widget"
; The value of the field named Size is greater than 4.
criteriaDA["Size"] = " 4"
; The value of the field named Cost is greater than or equal to 10.95
; and less than 22.50.

criteriaDA["Cost"] = "= 10.95, 22.50"
```

If the DynArray is empty or contains at least one empty item, any existing filter criteria are removed.

In Syntax 2, the array named *criteria* specifies conditions, and the optional Array *fieldId* specifies field names and numbers. If you omit *fieldID*, conditions are applied to fields in the order that they appear in the *criteria* array (the first condition applies to the first field in the table, the second condition applies to

setGenFilter method

the second field, etc.). The following example fills arrays for Syntax 2 to specify the criteria outlined in the Syntax 1 example.

```
criteriaAR[1] = "Widget"
criteriaAR[2] = " 4"
criteriaAR[3] = "= 10.95, 22.50"
fieldAR[1] = 1
fieldAR[2] = "Size"
fieldAR[3] = "Cost"
```

If the Array is empty or contains at least one empty item, the existing filter criteria is removed.

For both syntaxes, *idxName* specifies an index name (Paradox and dBASE tables) and *tagName* specifies a tag name (dBASE tables only). If you use these optional items, the index and tag are applied to the underlying table before the filtering criteria.

This method fails if the active record cannot be committed.

Filtering on special characters

If you are filtering on special characters, you must precede the number or literal value that can be interpreted as an operator (like `"/`, `"\`, `—`, `+`, `=`, etc.) with a backslash(`\`). In **setgenfilter()**, the filter criteria is put into a string and parsed to pick out numbers and operators for calculations. If the number or operator in the filter needs to be interpreted literally, it needs to be preceded by a backslash(`\`). For example to filter a table with the following records:

```
1st Base
1st Love
2nd Base
3rd Base
```

and retrieve only those that start with "1st," the filter would look like the following:

```
filter = "\1st.."
```

One backslash for the number and another to indicate the first backslash is not an escape sequence.

Note

- If you use **setGenFilter** on a UIObject in a running report, the filter does not take effect until you run the report again. For example, the following code runs a report and sets a filter; however, the filter has no effect until the report switches to design mode and then back into run mode.

```
method pushButton(var eventInfo Event)
    var
        reOrders      Report
        daCriteria     DynArray[] AnyType
    endVar

    reOrders.open("orders")

    daCriteria["OrderNo"] = " 1234"

    ; Assume the report contains a table frame bound to the Orders table.
    ; This statement has no effect because the report is in run mode.
    reOrders.ORDERS.setGenFilter(daCriteria)

    reOrders.design()
    reOrders.run() ; Now the filter takes effect.
endMethod
```

Example

The following example uses the **pushButton** method for a button named *balanceDueBtn* uses **setGenFilter** to filter a table frame on a form. This code filters the *ORDERS* table frame to display only those orders with a positive balance due.

```
;balanceDueBtn :: pushButton
method pushButton(var eventInfo Event)
  var
    dyn          DynArray[] String
    stField, stData String
  endVar

  stField = "Balance Due"
  stData = " 0"
  dyn[stField] = stData

  ORDERS.setGenFilter(dyn) ; ORDERS is a detail table frame.
endmethod
```

setPosition method**UIObject**

Sets the position of an object.

Syntax

```
setPosition ( const x LongInt, const y LongInt, const w LongInt, const h LongInt)
```

Description

setPosition sets the position of an object on the screen. Variables *x* and *y* specify the coordinates of an object's upper-left corner (in twips). Variables *w* and *h* specify the object's width and height (in twips). If the object is not specified, *self* is implied.

This method does not work when the UIObjects are forms. To set the position of a form, use **setPosition** (Form type).

You can also set and examine an object's position and size using the Position and Size properties.

```
self.Position = Point(100, 150)
self.Size = Point(2000, 2500)
```

The following code performs that same function as the previous code:

```
self.setPosition(100, 150, 2000, 2500)
```

For ObjectPAL, the screen is a two-dimensional grid. The origin (0, 0) is located at the upper-left corner of an object's container, with positive *x* values extending to the right, and positive *y* values extending down.

For dialog boxes and for the Paradox desktop application, the position is specified relative to the entire screen. For forms, reports, and table windows, the position is specified relative to the Paradox desktop.

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, respectively, 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 retrieves the current position of the ellipse using **getPosition** and moves it 100 twips to the right using **setPosition**.

The following code is attached to *toggleButton*'s **pushButton** method:

setProperty 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
```

The following code is attached to *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
```

setProperty method

UIObject

Sets a property to a specified value.

Syntax

```
setProperty ( const propertyName String, const propertyValue AnyType )
```

Description

setProperty sets an object's *propertyName* property to *propertyValue*. If the object does not have a *propertyName* property, or if *propertyValue* is invalid, this method fails.

setProperty is especially useful when *propertyName* is a variable; otherwise, you can access the property directly using the following code:

```
aBox.Color = Red
```

Example

The following example creates a dynamic array that's indexed by property names and contains property values. The array is filled using the array's index as the argument to the **getProperty** command. The method changes one of the property values and resets the object's properties using 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

```

setRange method/procedure

UIObject

Specifies a range of records to include in a field, table frame, or multi-record object. This method replaces **setFilter** that was included in earlier versions of Paradox. Code that calls **setFilter** executes as before.

Syntax

1. **setRange** ([const exactMatchVal AnyType] * [, const minVal AnyType, const maxVal AnyType]) Logical
2. **setRange** (rangeVals Array[] AnyType) Logical

Description

setRange specifies a range of records to include in a field, table frame, or multi-record object.

setRange compares the criteria that you specify with values in the corresponding fields of a table's index. If the active record cannot be committed or if the table is not indexed, this method fails. If you call **setRange** without any arguments the range criteria is reset to include the entire table.

In Syntax 1, you must specify values in *minVal* and *maxVal* to set a range based on the value of the first field of the index. For example, the following code determines values in the first field of each record's index:

```
tblObj.setRange(14, 88)
```

If a value is less than 14 or greater than 88, its corresponding record is excluded from the range. To specify an exact match on the first field of the index, assign *minVal* and *maxVal* the same value. For example, the following code excludes all values except 55:

```
tblObj.setRange(55, 55)
```

To set a range based on the values of more than one field specify exact matches *except* for the last one in the list. For example, the following statement looks for exact matches on Corel and Paradox, and on values ranging from 100 to 500, inclusive, for the third field:

```
tblObj.setRange("Corel", "Paradox", 100, 500)
```

In Syntax 2, you can pass an array of values to specify the range criteria. The following table displays the number of array items and their corresponding range specifications:

Number of array items	Range specification
No items (empty array)	No items resets range criteria to include the entire table.
One item	One item specifies a value for an exact match on the index's first field.
Two items	Two items specifies a range for the index's first field.

setTimer method

Three items	The first item specifies an exact match for the index's first field; items 2 and 3 specify a range for the index's second field.
More than three items	For an array of size n, specify exact matches on the index's first n-2 fields. The last two array items specify a range for the index's n-1 field.

Example 1

For the following 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 limits the number of records included in the LINEITEM object, including only 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.setRange(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
```

Example 2

The following example calls **setRange** with a criteria array that contains more than three items. The following code instructs a table frame to display orders from a person with a specific first name, middle initial, and last name. This table frame displays only those orders that range from 100 to 500 items. This example assumes that the *PartsOrd* table is indexed on the FirstName, MiddleInitial, LastName, and Qty fields.

```
; setQtyRange::pushButton
method pushButton(var eventInfo Event)
var
    arRangeInfo Array[5] AnyType
endVar

arRangeInfo[1] = "Frank"      ; FirstName (exact match)
arRangeInfo[2] = "P."        ; MiddleInitial (exact match)
arRangeInfo[3] = "Corel"     ; LastName (exact match)
arRangeInfo[4] = 100         ; Minimum qty value
arRangeInfo[5] = 500         ; Maximum qty value

PartsOrd.setRange(arRangeInfo) ; PartsOrd is a table frame
endMethod
```

setTimer method

UIObject

Starts an object's timer.

Syntax

```
setTimer ( const milliseconds LongInt [ , const repeat Logical ] )
```


Description

setTimer starts an object's timer. The timer interval (in milliseconds) is specified using *milliseconds*. The optional argument *repeat* specifies whether the timer automatically repeats. If *repeat* is set to True or omitted, the timer repeats; otherwise, the timer event is sent once. **setTimer** is attached to an object's **open** method, and the object's response is defined in its **timer** method.

Note

- Although Windows allows a maximum of 16 timers for all applications, Paradox has no 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, respectively, 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 retrieves the ellipse's position using **getPosition** and moves it 100 twips to the right using **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
```

skip method**UIObject**

Moves forward or backward through a specified number of records.

Syntax

```
skip ( const nRecords LongInt ) Logical
```

Description

skip moves forward or backward through a specified number of records. If you attempt to move beyond the limits of the table, **skip** fails.

switchIndex method

Specifying a positive value for *nRecords* moves forward through the table, specifying a negative value moves backward, and setting *nRecords* to 0 leaves the table as it is.

Note

- Setting *nRecords* = 0 is the same as **currRecord**
- Setting *nRecords* = —1 is the same as **priorRecord**
- Setting *nRecords* = 1 is the same as **nextRecord**

Example

The following example fills a table with records from the *Orders* table. Assume that the table *SampOrd* already exists with the same structure as *Orders*. The *createSampling* button exists on a form along with a table frame that is bound to *Orders*. *CreateSampling*'s **pushButton** method is shown below. The code moves the cursor 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
```

switchIndex method

UIObject

Specifies another index to use for viewing a table's records.

Syntax

1. **switchIndex** ([const *indexName* String] [, const *stayOnRecord* Logical]) Logical
2. **switchIndex** ([const *indexFileName* String] [, const *tagName* String [, const *stayOnRecord* Logical]]) Logical

Description

switchIndex specifies an index file to use with a table. In Syntax 1, *indexName* specifies an index to use with a Paradox table. If you omit *indexName*, the table's primary index is used.

Syntax 2 is for dBASE tables. *indexFileName* can specify an .NDX file or an .MDX file, and optional argument *tagName* specifies an index tag in a production index file (.MDX).

In both syntaxes, if optional argument *stayOnRecord* is set to Yes, this method maintains the active record after the index switch. If it is set to No, the first record in the table becomes the active record. If omitted, *stayOnRecord* is set to No by default.

Example

The following example assumes 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 primary 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
; quotes around "Name" distinguish field name from property name
endMethod
```

twipsToPixels method**UIObject**

Converts screen coordinates from twips to pixels.

Syntax

```
twipsToPixels ( const twips Point ) Point
```

Description

twipsToPixels converts the screen coordinates specified in *twips* from twips to pixels. A pixel is a dot on the screen, and a twip is a device-independent unit equal to 1/1440 of a logical inch (1/20 of a printer's point).

Example

See the **pixelsToTwips** example.

unDeleteRecord**UIObject**

Restores the current record in a dBASE table.

Syntax

```
unDeleteRecord ( ) Logical
```

Description

unDeleteRecord restores the current record of a dBASE table. This operation is successful if **showDeleted** has been set to True, if the record is deleted, and if the table object is in Edit mode.

unlockRecord method

Example

See the **unDeleteRecord** (Tcursor type) example.

unlockRecord method

UIObject

Removes a write lock from the active record.

Syntax

```
unlockRecord ( ) Logical
```

Description

unlockRecord returns True if it successfully removes an explicit write lock on the active record; otherwise, it returns False.

Note

- The Locked property is a read-only property. You can determine whether an object is locked, but you cannot lock or unlock an object.

Example

See the **recordStatus** example.

view method

UIObject

Displays the value of an object in a dialog box.

Syntax

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

Description

view displays the value of an object in a dialog box. Paradox suspends method execution until you close the dialog box. You can specify, in *title*, a title for the dialog box in the title string. If you omit *title*, the dialog box's title becomes the value's data type.

This method works only with the following UIObjects:

- buttons as checkboxes or buttons
- unbound fields only as lists or 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

The following example assumes that a form contains a table frame named *CUSTOMER* that is bound to the *Customer* table, and a button. The following code is attached to the button's **pushButton** method. This code creates an array of seven UIObjects and views 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

```

wasLastClicked method

UIObject

Determines whether an object received the last mouse click.

Syntax

```
wasLastClicked ( ) Logical
```

Description

wasLastClicked returns True if an object received the last mouse click; otherwise, it returns False. This method is only used with objects in the active form.

Example

The following example attaches code to the **mouseUp** method for an object named *boxOne*. If *boxOne* received the click, the message appears. If *boxOne* was sent a **mouseUp** event from another object, the method beeps.

The following code is attached to *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

```

The following code is attached to *sendAClick*'s **mouseUp** method:

```

; sendAClick::mouseUp
method mouseUp(var eventInfo MouseEvent)
boxOne.mouseUp(eventInfo) ; when boxOne's mouseUp gets this,
                           ; it will beep
endMethod

```

wasLastRightClicked method

UIObject

Determines whether an object received the last right-mouse click.

Syntax

```
wasLastRightClicked ( ) Logical
```

Description

wasLastRightClicked returns True if an object received the last right-mouse click; otherwise, it returns False. This method is only used with objects in the active form.

wasLastRightClicked method

Example

The following example is attached to the **mouseRightUp** method for an object named *circleOne*. If the ellipse received a right-click, the specified message displays. If the ellipse was sent a **mouseRightUp** event from another object, the code displays an alternate message.

The following code is attached to *circleOne*'s **mouseRightUp** 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
    msgInfo("Sent Right-click", "Invoked indirectly") ; method invoked indirectly
endIf
endMethod
```

The following is attached to the **mouseRightUp** method for an object named *sendARightClick*. When this object receives a right-click, it will send the event to *circleOne*.

The following code is attached to *sendARightClick*'s **mouseRightUp** method:

```
; sendARightClick::MouseRightUp
Method mouseRightUp(var eventInfo MouseEvent)
circleOne.mouseRightUp(eventInfo) ; when circleOne
; gets this it will trigger the second message box

endMethod
```

ValueEvent

ValueEvent methods control field value changes. In fact, the **changeValue** built-in event method is the only method triggered by a ValueEvent. This means that the built-in **newValue** method is not called with a ValueEvent; instead, **newValue** takes an Event.

The built-in **changeValue** method is called when the a field value is about to change. **changeValue** allows you to determine whether you want to post the value. The built-in **newValue** method reports when a field has already received a new value. Fields defined as buttons or lists behave differently. The built-in **newValue** method differs from the **newValue** method for the ValueEvent type.

The ValueEvent type includes several derived methods from the Event type.

Methods for the ValueEvent type

Event	←	ValueEvent
errorCode		newValue
getTarget		setNewValue
isFirstTime		
isPreFilter		
isTargetSel		
reason		
setErrorCode		
setReason		

newValue method

ValueEvent

Returns a new, unposted value for a ValueEvent.

Syntax

```
newValue ( ) AnyType
```

Description

newValue returns the new, unposted value for a ValueEvent. Because the new value is not yet assigned to a field, the following two statements might return different values:

```
field.Value  
eventInfo.newValue()
```

Note

- **newValue** differs from the built-in **newValue** method.

Example

In the following example, the **changeValue** method for the `creditLimit` field compares the old value with the new value. If the difference between the old and new values is greater than 25 per cent, **changeValue** blocks the change. Assume that `creditLimit` is an unbound field on a form, and that the form has at least one other field.

```
; 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
```

setNewValue method

ValueEvent

Specifies a value to set for a ValueEvent.

Syntax

```
setNewValue ( const newValue AnyType )
```

Description

setNewValue specifies a value to set for a ValueEvent. Ensure that the data type of the value is consistent with the field's type.

Example

The following example assumes that a form contains a field named *authorAbbrToName*, and at least one other field. When the user types an author abbreviation and moves off the field, **changeValue** fills in the full author name.

setNewValue method

```
; authorAbbrToName::changeValue
method changeValue(var eventInfo ValueEvent)
var
    abbrValue,
    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 = "AC" : fullValue = "Agatha Christie"
    case abbrValue = "SP" : fullValue = "Sara Paretsky"
    case abbrValue = "MHC" : fullValue = "Mary Higgins Clark"
    case abbrValue = "FK" : fullValue = "Faye Kellerman"
    case abbrValue = "SG" : fullValue = "Susan Grafton"
    case abbrValue = "AF" : fullValue = "Antonia Fraser"
    otherwise : fullValue = "Author Unknown"
endswitch

eventInfo.setNewValue(fullValue)
endMethod
```


A

ObjectPAL constants

ActionClasses constants

Constant	Data type	Description
DataAction	SmallInt	Data actions are for navigating in a table and for tasks such as record locking and record posting.
EditAction	SmallInt	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 are for moving within a field object.
SelectAction	SmallInt	Select actions are equivalent to Move actions.

ActionDataCommands constants

Constant	Data type	Description
DataArriveRecord	SmallInt	Indicates a change to the active record (e.g., navigation, editing, network refresh, and scrolling)
DataBegin	SmallInt	Moves to the first record in the table associated with the given UIObject. This constant forces recursive action (DataUnlockRecord) if the active record has been modified. If an error is encountered, it calls the error method. This constant is invoked by the First Record button, or Record, First.
DataBeginEdit	SmallInt	Used to enter Edit mode on the form. This constant is invoked by F9, the Edit icon, or View, Edit Data.
DataBeginFirstField	SmallInt	Moves to the first field in the first record of the table associated with the given UIObject. This constant is invoked by CTRL + HOME.

DataCancelRecord	SmallInt	Discards changes to a record. Succeeds by default, but user could block it. This constant is 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 active record. If an error is encountered, this constant calls the error method. This action is irreversible except for dBASE tables. This constant is invoked by Record, Delete or CTRL + DELETE.
DataDesign	SmallInt	Switches from running the form to the Form Design window. This constant is invoked by F8.
DataDitto	SmallInt	Copies into the current field the value of the corresponding field in the prior record. This constant is invoked by CTRL + D.
DataEnd	SmallInt	Moves to the final record in the table associated with the given UIObject. DataEnd forces a recursive action (DataUnlockRecord) if the active record has been modified. If an error is encountered, this constant will call the error method. This constant is invoked by the Last Record button.
DataEndEdit	SmallInt	Exits Edit mode on the form. This constant is invoked by (2nd) F9, Edit Data button on Toolbar, or View, View Data.
DataEndLastField	SmallInt	Moves to the last field of the last record of the table associated with a UIObject. This constant is invoked by CTRL + HOME.
DataFastBackward	SmallInt	Moves backward one set of records (where a set is defined as the number of rows in a table frame or MRO). This constant is invoked by Record, Previous Set, SHIFT + F11 or Previous Record Set button.
DataFastForward	SmallInt	Moves forward one set of records (where a set is defined as the number of rows in a table frame or MRO). This constant is invoked by Record, Next Set, SHIFT + F11 or Next Record Set button.
DataHideDeleted	SmallInt	Alters the mode of the form so that deleted records will be hidden (available only for dBASE tables). This constant is invoked by deselecting View, Show Deleted.

DataInsertRecord	SmallInt	Inserts a new (blank) record before the active record. The new record has a locked record state, and does not exist in the underlying table until the record is eventually modified and unlocked. This constant is invoked by Record, Insert, or INSERT. Note that records created this way can be discarded by using DataDeleteRecord or DataCancelRecord before they have been unlocked. If you Move off such a record without making changes, DataCancelRecord to discard it.
DataLockRecord	SmallInt	Locks the active record. If an error is encountered, this constant calls the error method. This constant is invoked by F5.
DataLookup	SmallInt	Invokes lookup table for the current field, to accept user's choice of a new value, and, if appropriate, to update all corresponding fields governed by lookup. DataLookup is available only for fields that have been defined as lookup fields. This constant is invoked by CTRL + SPACEBAR.
DataLookupMove	SmallInt	Allows the user to choose a new master record for this detail. This constant is invoked by Record, Move Help or CTRL + SHIFT + SPACEBAR.
DataNextRecord	SmallInt	Moves to the next sequential record in the table associated with the UIObject. DataNextRecord forces a recursive action (DataUnlockRecord) if the active record has been modified. If an error is encountered, this constant calls the error method. This constant is invoked by Record, Next, the 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. This constant is invoked by PAGEDOWN.
DataPostRecord	SmallInt	Posts a record. DataPostRecord is 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 lies with that record (meaning it will still be the active record). This constant is invoked by CTRL + F5 or Record, Post/Keep Locked.
DataPrint	SmallInt	Prints a Form or Table window. This constant is invoked by File, Print or the Print button.

DataPriorRecord	SmallInt	Moves (if possible) to the previous record in the table associated with the UIObject. DataPriorRecord forces recursive action(DataUnlockRecord) if the active record has been modified. If an error is encountered, this constant calls the error method. This constant is invoked by Record, Previous, the 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-recordform). DataPriorSet forces a recursive action (DataUnlockRecord) if the active record has been modified. If an error is encountered, this constant calls the error method. This constant is invoked by PAGEUP.
DataRecalc	SmallInt	Forces an object and all objects it contains to refetch and recompute all their data. This constant is invoked by CTRL + F3.
DataRefresh	SmallInt	Notifies users about a refresh of a value in a record displayed on the screen.
DataRefreshOutside	SmallInt	Notifies users about a refresh of a value in a record not displayed on the screen.
DataSaveCrosstab	SmallInt	Writes given crosstab to CROSSTAB.DB. Different from EditSaveCrosstab, which brings up a dialog box asking user the name of the crosstab table to create.
DataSearch	SmallInt	Opens a dialog box to allow the user to search for a specific value within a specified field. This constant is invoked by Record, Locate, Value, or CTRL + Z.
DataSearchNext	SmallInt	Searches for the next record containing the value last specified in response to the last DataSearch action. This onstant is invoked by Record, Locate Next, or CTRL + A.
DataSearchRecord	SmallInt	Opens a dialog box to allow the user to search for a record by specifying the record number. This constant is invoked by Record, Locate, Record Number.
DataSearchReplace	SmallInt	Opens a dialog box to allow the user to search for a specific value within a specified field and to replace it with a different value. This constant is invoked by Record, Locate, and Replace, or CTRL + 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 the status line will reflect their state. This constant is invoked by View, Show Deleted.

DataTableView	SmallInt	Opens a Table View of the master table of a form. This constant is invoked by F7, the Table View Button or View, Table View.
DataToggleDeleted	SmallInt	Reverses the state of show deleted records for dBASE tables.
DataToggleDeleteRecord	SmallInt	Reverses the deleted state of records in dBASE tables.
DataToggleEdit	SmallInt	Reverses the Edit state of the form. DataToggleEdit recursively calls DataBeginEdit or DataEndEdit as appropriate. This constant is invoked by F9, or the Edit Data button.
DataToggleLockRecord	SmallInt	Reverses the lock state of the active record. DataToggleLockRecord recursively uses DataLockRecord or DataUnlockRecord as appropriate. If an error is encountered, this constant calls the error method.
DataUnDeleteRecord	SmallInt	Marks previously deleted record as undeleted (for dBASE tables)
DataUnlockRecord	SmallInt	Commits the record modifications to the table and then (if successful) to unlock the record. If an error is encountered, this constant calls the error method. This constant is invoked by Record, Unlock or SHIFT + F5.

ActionEditCommands constants

Constant	Data type	Description
EditCommitField	SmallInt	Writes the current field's modifications to record buffer (without leaving field)
EditCopySelection	SmallInt	Copies selected area of text to Clipboard. This constant is invoked by Edit, Copy or CTRL + Ins.
EditCopyToFile	SmallInt	Invokes a dialog box to copy selection to a file. This constant is invoked by Edit, Copy To.
EditCutSelection	SmallInt	Copies selected area of text to Clipboard and deletes it. This constant is invoked by Edit, Cut or CTRL + DELETE.
EditDeleteBeginLine	SmallInt	Deletes from the current position to the beginning of the line
EditDeleteEndLine	SmallInt	Deletes from the current position to the end of the line
EditDeleteLeft	SmallInt	Deletes one character position to the left. This constant is invoked by Backspace in Field View.
EditDeleteLeftWord	SmallInt	Deletes up to and including the beginning of the word to the left of the current character position

EditDeleteLine	SmallInt	Deletes the line on which the current position is found
EditDeleteRight	SmallInt	Deletes one character position to the right. This constant is invoked by Del in Field View.
EditDeleteRightWord	SmallInt	Deletes up to and including the end of the word to the right of the current character position
EditDeleteSelection	SmallInt	Deletes the currently selected area of text. This constant is invoked by Edit, Delete.
EditDeleteWord	SmallInt	Deletes the word around the current position. This constant is invoked by CTRL + BACKSPACE.
EditDropDownList	SmallInt	Drops down the pick list associated with a drop-down edit field. This constant is invoked by ALT + the Down Arrow key or clicking edit field'sList icon.
EditEnterFieldView	SmallInt	Enters Field View for the current field (allowing arrow keys to move around within the field). Begins by moving the current position to the end of field and unhighlighting it. This constant is invoked by F2, View, Field View, or the Field View button.
EditEnterMemoView	SmallInt	Enters Memo View on memos or OLE fields. This constant is invoked by SHIFT + F2 or View, Memo View.
EditEnterPersistFieldView	SmallInt	Enters Persistent Field View, meaning arrow keys always move within character positions within a field, even when moving to new fields. This constant is invoked by CTRL + F2 or View, Persistent Field View.
EditExitFieldView	SmallInt	Exits Field View (meaning the arrow keys will move between fields again) and highlights entire field. This constant is invoked by F2, View, Field View, or the Field View button.
EditExitMemoView	SmallInt	Exits Memo View on memos or OLE fields, meaning Enter and TAB will once again move between fields. This constant is invoked by SHIFT + F2 or View, Memo View.
EditExitPersistField View	SmallInt	Exits Persistent Field View, meaning arrow keys move between fields. This constant is invoked by CTRL + F2 or View, Persistent Field View.
EditHelp	SmallInt	Invokes the Help subsystem. This constant is invoked by F1.
EditInsertBlank	SmallInt	Inserts a blank character at the current position
EditInsertLine	SmallInt	Inserts a blank line at the current position
EditInsertObject (5.0)	SmallInt	Inserts a linked or embedded object into the current field (used only by OLE fields)

EditLaunchServer	SmallInt	Invokes the server application appropriate for the current field (used only by OLE fields)
EditPaste	SmallInt	Pastes from the Clipboard to the current position (replacing the active selection if appropriate). This constant is invoked by SHIFT + INSERT or Edit, Paste.
EditPasteFromFile	SmallInt	Invokes a dialog box, allowing user to select file to insert at the current position. This constant is invoked by Edit, Paste From.
EditPasteLink (5.0)	SmallInt	Pastes an object from the Clipboard and establishes a link to the underlying file (used only by OLE fields). This constant is invoked by Edit, Paste Link.
EditProperties	SmallInt	Invokes the property inspection menu for the given object. Only unbound field objects, bound graphic fields, and bound formatted memo fields support this. This constant is invoked by mouse right-click, Properties, Current Object, or F6.
EditReplace	SmallInt	Toggles overstrike mode in a field object
EditSaveCrosstab (5.0)	SmallInt	Invokes a dialog box to allow user to save a crosstab. This constant is invoked by Edit, Save Crosstab.
EditTextSearch	SmallInt	Invokes a dialog box to allow user to search and replace text within the current field. This constant is invoked by Edit, Search Text.
EditToggleFieldView	SmallInt	Reverses the current state of Field View. EditToggleFieldView recursively calls EditEnterFieldView or EditExitFieldView. This constant is invoked by F2, the Field View button, or Edit, Field View.
EditUndoField	SmallInt	Discards the current field's modifications and reverts to value in the active record buffer. This constant is invoked by ESC.

ActionFieldCommands constants

Constant	Data type	Description
FieldBackward	SmallInt	Moves one field backward in tab order. This will search for the prior UIObject marked as a Tab Stop in left-right/top-down order. This constant is invoked by SHIFT + TAB.

FieldDown	SmallInt	Moves to field below the current field, whether in Field View or not. This constant is invoked by ALT + the Down Arrow key.
FieldEnter	SmallInt	Used to commit modifications to a field (if any) and to move one field forward in tab order. This constant is invoked by ENTER.
FieldFirst	SmallInt	Moves to the first field within a record. This constant is invoked by ALT + HOME.
FieldForward	SmallInt	Moves one field forward in tab order. This will search for the next UIObject marked as a Tab Stop in left-right/top-down order. This constant is invoked by Tab.
FieldGroupBackward	SmallInt	Moves one super tab group backward (e.g., between different table frames on the same form). This constant is invoked by F3.
FieldGroupForward	SmallInt	Moves one super tab group forward (e.g., between different table frames on the same form). This constant is invoked by F4.
FieldLast	SmallInt	Moves to the last field within a record. This constant is invoked by ALT + END or by END (when not in Field View).
FieldLeft	SmallInt	Moves to the field left of the current field. This constant is invoked by ALT + the Down Arrow key.
FieldNextPage	SmallInt	Moves to the next sequential page in multi-page form. This constant is invoked by View, Page, Next or SHIFT + F4.
FieldPriorPage	SmallInt	Moves to the prior page in multi-page form. This constant is invoked by View, Page, Previous or SHIFT + F3.
FieldRight	SmallInt	Moves to the field right of the current field, whether in Field View or not. This constant is invoked by ALT + the Right Arrow key.
FieldRotate	SmallInt	Used to rotate columns within a table frame. This constant is invoked by CTRL + R.
FieldUp	SmallInt	Moves to the field above the current field, whether in Field View or not. This constant is invoked by ALT + the Up Arrow key.

ActionMoveCommands constants

Constant	Data type	Description
MoveBegin	SmallInt	Moves to the beginning of the document in Memo view; otherwise, it moves to the first field in the first record of table. This constant is invoked by CTRL + HOME.
MoveBeginLine	SmallInt	Moves to the beginning of the line in Memo view; otherwise, it moves to the first field in the record. This constant is invoked by HOME.
MoveBottom	SmallInt	Moves to the bottom line of the text region in Memo view. Otherwise, it moves to the last record in table.
MoveBottomLeft	SmallInt	Moves to the beginning of the last line on screen in Memo view
MoveBottomRight	SmallInt	Moves to the end of the last line on screen in Memo view. This constant is invoked by CTRL + PAGEDOWN.
MoveDown	SmallInt	Moves down as appropriate. In Memo View, it moves down one line on multi-line fields. Otherwise, it moves to the next Tab Stop object below the active object. Table frame objects move to the next record. This constant is invoked by the Down Arrow key.
MoveEnd	SmallInt	Moves to the end of the document in Memo view; otherwise, it moves to the last field in the last record of table. This constant is invoked by CTRL + END.
MoveEndLine	SmallInt	Moves to the end of the line in Memo view; otherwise, it moves to the last field in the record. This constant is invoked by END.
MoveLeft	SmallInt	Moves left as appropriate. In Memo View, it moves one character position left; otherwise, it moves to the next Tab Stop object right of the active object. This constant is invoked by the Left Arrow key.
MoveLeftWord	SmallInt	Moves the cursor to the beginning of the word to the left of the current insertion point in Memo view. This constant is invoked by CTRL + the Left Arrow key.
MoveRight	SmallInt	Moves right as appropriate. In Memo View, it moves one character position right; otherwise, it moves to the next Tab Stop object right of the active object. This constant is invoked by the Right Arrow key.

MoveRightWord	SmallInt	Moves the cursor to the beginning of the word to the right of the current insertion point. This constant is invoked by CTRL + the Right Arrow key.
MoveScrollDown	SmallInt	Scrolls the image down (effectively moving viewing area up) by appropriate amount. Active fields scroll by even lines of text. Tables move to a new record. In Memo View, scroll toward the bottom of the text. The cursor remains on the same line of the display region unless the last line of the text is visible, in which case the cursor moves down one line until the last line is reached. This constant is invoked by CTRL + the Down Arrow key.
MoveScrollLeft	SmallInt	Scrolls the image right (effectively moving viewing area to the right) by appropriate amount. Active fields scroll roughly one character position. Tables move to a new column.
MoveScrollPageDown	SmallInt	Scrolls the image down (effectively moving viewing area up) by the logical size of the object (e.g., the complete page of the document). This constant is invoked by PAGEDOWN.
MoveScrollPageLeft	SmallInt	Scrolls the image left (effectively moving viewing area right) by the logical size of the object (e.g., the complete page of the document).
MoveScrollPageRight	SmallInt	Scrolls the image right (effectively moving viewing area left) by the logical size of the object (e.g., the complete page of the document).
MoveScrollPageUp	SmallInt	Scrolls the image up (effectively moving viewing area down) by the logical size of the object (e.g., the complete page of the document). This constant is invoked by PAGEUP.
MoveScrollRight	SmallInt	Scrolls the image right (effectively moving viewing area to the left) by appropriate amount. Active fields scroll roughly one character position. Tables move to a new column.
MoveScrollScreenDown	SmallInt	Scrolls the image down (effectively moving viewing area up) by the size of viewing area (e.g., the size of the field). In Memo View, moves down in the document by the height of the display area.
MoveScrollScreenLeft	SmallInt	Scrolls the image left (effectively moving viewing area right) by the size of viewing area (e.g., the size of the field).
MoveScrollScreenRight	SmallInt	Scrolls the image right (effectively moving viewing area left) by the size of viewing area (e.g., the size of the field).

MoveScrollScreenUp	SmallInt	Scrolls the image up (effectively moving viewing area down) by the size of viewing area (e.g., the size of the field). In Memo View, moves up in the document by the height of the display area.
MoveScrollUp	SmallInt	Scroll the 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 cursor stays at the same line position unless the top line of the document is visible, in which case the cursor moves up one line if it can. This constant is invoked by CTRL + the Up Arrow key.
MoveTop	SmallInt	Moves the cursor to the first line of text visible in the display region in Memo view; otherwise, it moves to the first record in table.
MoveTopLeft	SmallInt	Moves to the top left of the display region in Memo view; otherwise, it moves to top left field. This constant is invoked by CTRL + PAGEUP.
MoveTopRight	SmallInt	Moves to the top right of the display region in Memo view; otherwise, it moves to top right field.
MoveUp	SmallInt	Moves up as appropriate. In Memo View, it moves up one line on multi-line fields; otherwise, it moves to the next Tab Stop object above the active object. Table frame objects move to the prior record. This constant is invoked by the Up Arrow key.

ActionSelectCommands constants

Constant	Data type	Description
SelectBegin	SmallInt	In Memo View, it selects from the current position to the beginning of the document; otherwise, it selects from the current position to the first field in the first record of table. This constant is invoked by SHIFT + CTRL + HOME.
SelectBeginLine	SmallInt	In Memo View, it selects from the current position to the beginning of the line; otherwise, it selects from the current position to the first field in the record. This constant is invoked by SHIFT + HOME.
SelectBottom	SmallInt	In Field View and Memo View, select from the current position to bottom of the display region; otherwise, it selects from the current position to the last record in table.

SelectBottomLeft	SmallInt	In Memo View, it selects from the current position to the beginning of the last line in the display region. This constant is invoked by SHIFT + CTRL + PAGEUP.
SelectBottomRight	SmallInt	In Memo View, it selects from the current position to the end of the last line in the display region. This constant is invoked by SHIFT + CTRL + PAGEDOWN.
SelectDown	SmallInt	Selects down as appropriate. In Field View or Memo View, it selects down one line on multi-line fields. Cannot extend selection across fields in forms. Table frame objects select to the next record. This constant is invoked by SHIFT + the Down Arrow key.
SelectEnd	SmallInt	In Field View or Memo View, it selects from the current position to the end of the document; otherwise, it selects from the current position to the last field in the last record of table. This constant is invoked by SHIFT + CTRL + END.
SelectEndLine	SmallInt	In Field View or Memo View, it selects from the current position to the end of the line; otherwise, it selects from the current position to the last field in the record. This constant is invoked by SHIFT + END.
SelectLeft	SmallInt	Selects left as appropriate. In Field View or Memo View, it selects one character position left; otherwise, it selects the next Tab Stop object right of the active object. This constant is invoked by SHIFT + the Left Arrow key.
SelectLeftWord	SmallInt	In Field View or Memo View, if the cursor is between words, it selects the word to the left of the cursor. If the cursor is within a word, it selects to the beginning of that word. This constant is invoked by SHIFT + CTRL + the Left Arrow key.
SelectRight	SmallInt	Selects right as appropriate. In Field View or Memo View, it selects one character position right. This constant is invoked by SHIFT + the Right Arrow key.
SelectRightWord	SmallInt	In Field View or Memo View, it selects to the beginning of the next word. If the cursor precedes one or more spaces or tabs, selection only includes those spaces or tabs. This constant is invoked by SHIFT + CTRL + the Right Arrow key.
SelectScrollDown	SmallInt	Selects the image down (effectively moving viewing area up) by appropriate amount. Active fields select even lines of text. Tables select a new record. This constant is invoked by SHIFT + CTRL + the Down Arrow key.

SelectScrollLeft	SmallInt	Selects the image on left (effectively moving viewing area to the right) by appropriate amount. Active fields select roughly one character position. Tables select to a new column.
SelectScrollPageDown	SmallInt	Selects the image down (effectively moving viewing area up) by the logical size of the object (e.g., the complete page of the document).
SelectScrollPageLeft	SmallInt	Selects the image left (effectively moving viewing area right) by the logical size of the object (e.g., the complete page of the document).
SelectScrollPageRight	SmallInt	Selects the image right (effectively moving viewing area left) by the logical size of the object (e.g., the complete page of the document).
SelectScrollPageUp	SmallInt	Selects the image up (effectively moving viewing area down) by the logical size of the object (e.g., the complete page of the document).
SelectScrollRight	SmallInt	Selects the image on right (effectively moving viewing area to the left) by appropriate amount. Active fields select roughly one character position. Tables select a new column.
SelectScrollScreenDown	SmallInt	Selects the image down (effectively moving viewing area up) by the size of viewing area (e.g., the size of the field). This constant is invoked by SHIFT + PAGEDOWN.
SelectScrollScreenLeft	SmallInt	Selects the image left (effectively moving viewing area right) by the size of viewing area (e.g., the size of the field).
SelectScrollScreenRight	SmallInt	Selects the image right (effectively moving viewing area left) by the size of viewing area (e.g., the size of the field).
SelectScrollScreenUp	SmallInt	Selects the image up (effectively moving viewing area down) by the size of viewing area (e.g., the size of the field). This constant is invoked by SHIFT + PAGEUP.
SelectScrollUp	SmallInt	Moves the image up (effectively moving viewing area down) by appropriate amount. Active fields move by even lines of text.
SelectSelectAll	SmallInt	Selects the entire field
SelectTop	SmallInt	In Field View or Memo View, it selects from the current position to the top of the display region; otherwise, it selects from the current position to the first record in table.

SelectTopLeft	SmallInt	In Field View or Memo View, it selects from the current position to the beginning of screen; otherwise, it selects from the current position to the top left field. This constant is invoked by SHIFT + CTRL + PAGEUP.
SelectTopRight	SmallInt	In Field View or Memo View, it selects from the current position to the end of the top line of the screen; otherwise, it selects from the current position to the top right field. This constant is invoked by SHIFT + CTRL + PAGEDOWN.
SelectUp	SmallInt	Selects up as appropriate. In Field View or Memo View, it selects up one line on multi-line fields; otherwise, it selects the next Tab Stop object above the active object. Table frame objects select to the prior record. This constant is invoked by SHIFT + the Up Arrow key.

AggModifiers constants

Constant	Data type	Description
CumulativeAgg	SmallInt	A cumulative summary that keeps a running total that extends from the start of the report to the end of the current group.
RegularAgg	SmallInt	A normal summary that considers all non-null values in the set, including duplicates.
UniqueAgg	SmallInt	A unique summary that counts only the unique non-null values in the set. Duplicates are ignored.

BrowserOptions constants

Constant	Data type	Description
BrowseOptCreatePrompt	LongInt	Prompts the user for permission to create a file
BrowseOptFileMustExist	LongInt	Specifies that the user can type only names of existing files
BrowseOptNoNetButton	LongInt	Hides the network button on the dialog box
BrowseOptPathMustExist	LongInt	Specifies that the user can type only valid paths and filenames

ButtonStyles constants

Constant	Data type	Description
BorlandButton	SmallInt	Gives a button the 3D look of buttons in Corel products

Windows3dButton	SmallInt	Gives a button the same look as 3D buttons in other Windows products
WindowsButton	SmallInt	Gives a button the same look as flat buttons in other Windows products

ButtonTypes constants

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

Color constants

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

The following constants are dependent on your system's display settings

Constant	Data type	Description
clbackground	LongInt	Takes the defined color for the desktop background
clActiveCaption	LongInt	Takes the defined color for an active message box title bar
clInactiveCaption	LongInt	Takes the defined color for an inactive message box title bar
clMenu	LongInt	Takes the defined color for a menu
clMenuText	LongInt	Takes the defined color for the Menu text
clWindow	LongInt	Takes the defined color for a window background (message box background)
clWindowFrame	LongInt	Takes the defined color for a window frame (message box frame)
clWindowText	LongInt	Takes the defined color for the window text (message box text)
clActiveBorder	LongInt	Takes the defined color for an active window's border
clInactiveBorder	LongInt	Takes the defined color for an inactive window's border
clAppWorkspace	LongInt	Takes the defined color for an application's workspace
clHighlight	LongInt	Takes the defined color for a selected item
clHighlightText	LongInt	Takes the defined color for the font of a selected item
clBtnText	LongInt	Takes the defined color for 3D object's text (button text)
clInactiveCaptionText	LongInt	Takes the defined color for the text of an inactive title bar
clInfobk	LongInt	Takes the defined color for a tooltip window
clInfoText	LongInt	Takes the defined color for a tooltip window's text
clDesktop	LongInt	Takes the defined color for the desktop background
cl3dFace	LongInt	Takes the defined color for the 3D object's face
clBtnFace	LongInt	Takes the defined color for the 3D object's face
clGrayText	LongInt	Takes the defined color from the disabled text
clCaptionText	LongInt	Takes the text color in caption, size box, and scroll bar arrow box
cl3dShadow	LongInt	Takes the Dark shadow color for 3D display elements
cl3dHighlight	LongInt	Takes the dark shadow color for 3D display elements
cl3dHilight	LongInt	Takes the dark shadow color for 3D display elements
clBtnHighlight	LongInt	Takes the dark shadow color for 3D display elements

cl3dDkShadow	LongInt	Takes the shadow color for 3D display elements (for edges facing the light source)
clBtnShadow	LongInt	Takes the shadow color for 3D display elements (for edges facing the light source)
cl3dLight	LongInt	Takes the light color for 3D display elements

Compatibility constants

Constant	Data type	
peAliasPathNonExistant	SmallInt	
peIXForeignKeyError	SmallInt	
peIXInva	SmallInt	
peNonExistantAlias	SmallInt	
pePathNonExistant	SmallInt	

CompleteDisplay constants

Constant	Data type	Description
DisplayAll	SmallInt	Specifies CompleteDisplay for all field objects in the form
DisplayCurrent	SmallInt object	Specifies CompleteDisplay for the current field object

DataTransferCharset constants

Constant	Data type	Description
dtANSI	SmallInt	Specifies the ANSI character set
dtOEM	SmallInt	Specifies the OEM character set

DataTransferDelimitCode constants

Constant	Data type	Description
dtDelimAllFields	SmallInt	Specifies to delimit all fields
dtDelimJustText	SmallInt	Specifies to delimit just text fields

DataTransferFileType constants

Constant	Data type	Description
dtI23V1	SmallInt	Specifies Lotus I23 (.WKS)
dtI23V2	SmallInt	Specifies Lotus I23 (.WK1)
dtASCIIFixed	SmallInt	Specifies ASCII Fixed (BDE)
dtASCIIVar	SmallInt	Specifies ASCII Delimited
dtAuto	SmallInt	Automatically determine file type based on file extension
dtdBase3	SmallInt	Specifies Export to dBASE III + compatible
dtdBase4	SmallInt	Specifies Export to dBASE IV compatible
dtdBase5	SmallInt	Export to dBASE 5 compatible, Import any dBASE
dtdBaseAny	SmallInt	Import (or Export) any dBASE version
dtExcel4	SmallInt	Specifies Excel Version 3,4 (.XLS)
dtExcel5	SmallInt	Specifies Excel Version 5 (.XLS)
dtParadox3	SmallInt	Export to Paradox 3 compatible
dtParadox4	SmallInt	Export to Paradox 4 compatible
dtParadox5	SmallInt	Export to Paradox 5 compatible
dtParadox7	SmallInt	Export to Paradox 7 compatible
dtParadoxAny	SmallInt	Import (or Export) any Paradox version
dtQPW1	SmallInt	Specifies Quattro Pro Windows I, 5 (.WB1)
dtQPW6	SmallInt	Specifies Quattro Pro Windows 6 (.WB2)
dtQPW7	SmallInt	Specifies Quattro Pro Windows 95 (.WB3)
dtQuattro	SmallInt	Specifies Quattro DOS (.WKQ)
dtQuattroPro	SmallInt	Specifies Quattro Pro DOS (.WQ1)

DateRangeTypes constants

Constant	Data type	Description
ByDay	SmallInt	Group report records by day
ByMonth	SmallInt	Group report records by month
ByQuarter	SmallInt	Group report records by quarter (3 months)
ByWeek	SmallInt	Group report records by week

ByYear

SmallInt

Group report records by year

DesktopPreferenceTypes constants

Constant	Data type	Description
prefStartUpExpert	SmallInt	Run the Startup Expert each time Paradox loads (Experts page)
prefTitleName	SmallInt	Title (General page)
prefExpertDefault	SmallInt	Run the experts when creating objects on documents (Experts page)
prefBackgroundName	SmallInt	Background bitmap (General page)
prefTileBitmap	SmallInt	Tile bitmap (General page)
prefSaveOnExit	SmallInt	Desktop state: Save on exit (General page)
prefRestoreDesktop	SmallInt	Desktop state: Restore on startup (General page)
prefSystemFont	SmallInt	Default system font (General page)
prefScreenPageDesk	SmallInt	On-screen size: Size to desktop (Forms/Reports page)
prefScreenPageWidth	SmallInt	On-screen size: Width (Forms/Reports page)
prefScreenPageHeight	SmallInt	On-screen size: Height (Forms/Reports page)
prefFormOpen	SmallInt	Open default: Open forms in design mode (Forms/Reports page)
prefReportOpen	SmallInt	Open default: Open reports in design mode (Forms/Reports page)
prefWarnOnDirChange	SmallInt	Don't show warning prompts when changing directories (Advanced page)
prefBitmapButtons	SmallInt	Changes to Corel-style buttons
prefAltKeyPadChars	SmallInt	Always use ALT + numeric keypad for character entry (Advanced page)
prefExpandBranchs	SmallInt	Indicate expandable directory branches (Advanced page)
prefScrollBarsInForms	SmallInt	Use scroll bars in form windows by default (Advanced page)
prefBlankAsZeroName	SmallInt	Treat blank fields as zeros (Database page)
prefRefreshRate	SmallInt	Refresh rate (seconds) (Database page)
prefExpertsOnCreate	SmallInt	Always use expert (New forms/reports)
prefUserLevel	SmallInt	ObjectPAL level (Developer Preferences: General page)

prefDeveloperMenu	SmallInt	Show developer menus (Developer Preferences: General page)
prefEnableControlBreak	SmallInt	Debugger settings: Enable CTRL + Break (Developer Preferences, General page)
Section = prefQbeSection		
prefAuxOpts	SmallInt	Generate auxiliary tables
prefSqlRunMode	SmallInt	Queries against remote tables (Query)
prefDefCheck	SmallInt	Default QBE check type
prefSqlconstrained	SmallInt	SQL answer constraints
Section = prefProjViewSection		
prefOpenOnStartup	SmallInt	Open Project Viewer on startup (Project Viewer settings: General page)

DeviceType constants

Constant	Data type	Description
Display	SmallInt	
Printer	SmallInt	

ErrorReasons constants

Constant	Data type	Description
ErrorCritical	SmallInt	Displays a message in a modal dialog box
ErrorWarning	SmallInt	Displays a message in the status area

EventErrorCodes constants

Constant	Data type	Description
Can_Arrive	SmallInt	Grants permission to arrive at an object
Can_Depart	SmallInt	Grants permission to leave an object
CanNotArrive	SmallInt	Refuses permission to arrive at an object (blocks the move)
CanNotDepart	SmallInt	Refuses permission to leave an object (blocks the move)

Errors constants

Constants	Data type	Description
ValCheckMayNotBeEnforced	SmallInt	Validity check will not be enforced.
peARYFixedSizeArray	SmallInt	The '%0ds' operation is not allowed on fixed-size arrays.
peARYIndexOutOfBounds	SmallInt	The specified array index is out of bounds. The index is %0dl, and the array limit is %ldl.
peARYNoMemory	SmallInt	Not enough memory to allocate or grow the array.
peARYRangeTooLarge	SmallInt	The starting and ending indexes are not valid for this array. The starting index is %0dl, ending index is %ldl, and the array size is %2dl.
peARYTooLarge	SmallInt	You cannot allocate an array with a size larger than 64k.
peAccessDisabled	SmallInt	Access to table disabled because of previous error.
peAccessError	SmallInt	Invalid file access.
peActionNotSupported	SmallInt	Action not supported for this object
peActiveIndex	SmallInt	Index is being used to order table.
peActiveTrans	SmallInt	A Transaction is currently active.
peActiveTransaction	SmallInt	A user transaction is already in progress.
peAliasInUse	SmallInt	The alias '%0ds' is in use.
peAliasIsServer	SmallInt	Alias is a server.
peAliasMismatch	SmallInt	The destination table of the rename has a conflicting alias.
peAliasNotDefined	SmallInt	The alias '%0ds' has not been defined.
peAliasNotOpen	SmallInt	Alias is not currently opened.
peAliasPathNonExistent	SmallInt	The path for the alias '%0ds' does not exist.
peAliasProjectConflict	SmallInt	The Public Alias being added '%0ds' is already a Project Alias.
peAliasPublicConflict	SmallInt	The Project Alias being added '%0ds' is already a Public Alias.
peAlias_X_Db	SmallInt	The alias '%0ds' and the Database '%ldl' do not match.
peAllFieldsReadOnly	SmallInt	All fields are read only.
peAlreadyLocked	SmallInt	Record already locked by this session.
peArgumentNumber	SmallInt	'%0ds' failed because it has the wrong number of arguments supplied.

peArgumentTypeInvalid	SmallInt	A method which takes an indeterminate number of arguments has an argument which is not a valid type.
peBOF	SmallInt	At beginning of table.
peBadISep	SmallInt	Bad Date Separator
peBadITSep	SmallInt	Bad Time Separator
peBad2Sep	SmallInt	Bad Date Separator
peBad2Tsep	SmallInt	Bad Time Separator
peBad3Sep	SmallInt	Bad Date Separator
peBad3Tsep	SmallInt	Bad Time Separator
peBad4Sep	SmallInt	Bad Date Separator
peBad4Tsep	SmallInt	Bad Time Separator
peBad5Sep	SmallInt	Bad Date Separator
peBad5Tsep	SmallInt	Bad Time Separator
peBad6Tsep	SmallInt	Bad Time Separator
peBadAMPM	SmallInt	Bad AM-PM Specification
peBadAlias	SmallInt	Unknown alias.
peBadArgument	SmallInt	'%0ds' failed because argument %ldi was not legal.
peBadArrayResize	SmallInt	Could not resize a dynamic array.
peBadBlobHeader	SmallInt	Blob has invalid header.
peBadCharsInAlias	SmallInt	Illegal characters in alias
peBadConstantGroup	SmallInt	The constant group '%0ds' was not found.
peBadDate	SmallInt	Bad Date Specification
peBadDay	SmallInt	Bad Day Specification
peBadDriverType	SmallInt	Invalid driver name.
peBadField	SmallInt	Invalid field.
peBadFieldType	SmallInt	Field '%0ds' has a badly formed type '%lds'.
peBadFileFormat	SmallInt	Cannot interpret file. It could be corrupt.
peBadHour	SmallInt	Bad Hour Specification
peBadLinkIndex	SmallInt	Index used to join tables is no longer valid
peBadLogical	SmallInt	Bad Logical Specification
peBadMinutes	SmallInt	Bad Minute Specification

peBadMonth	SmallInt	Bad Month Specification
peBadObject	SmallInt	The method, '%0ds', is not allowed on a '%lds' object.
peBadSeconds	SmallInt	Bad Seconds Specification
peBadTable	SmallInt	Invalid table.
peBadTime	SmallInt	Bad Time Specification
peBadTypeArray	SmallInt	Trying to do a copyToArray or copyFromArray with an array that does not correspond, has unassigned elements or is empty.
peBadVersion	SmallInt	The ObjectPAL version used in this form is incompatible with this version of Paradox. You must recompile from source.
peBadWeekday	SmallInt	Bad Day of Week Specification
peBadXtabAction	SmallInt	Action is not supported in a crosstab
peBadYear	SmallInt	Bad Year Specification
peBigXtab	SmallInt	Crosstab or Query contains too many fields.
peBlankField	SmallInt	The field is blank.
peBlankTableName	SmallInt	A blank table name was provided.
peBlankValue	SmallInt	Value is illegal or blank.
peBlobFileMissing	SmallInt	BLOB file is missing.
peBlobModified	SmallInt	BLOB has been modified.
peBlobNotOpened	SmallInt	BLOB not opened.
peBlobOpened	SmallInt	BLOB already opened.
peBlobReaderror	SmallInt	Problem reading data from .MB file on disk.
peBlobVersion	SmallInt	BLOB file version is too old.
peBracketMismatch	SmallInt	Mismatched brackets.
peBreak	SmallInt	Stopped program at your request.
peBufferSizeError	SmallInt	Buffer size error ??
peBufferTooSmall	SmallInt	Buffer is too small.
peCFunction	SmallInt	Found problem in a CFunction operation.
peCancel	SmallInt	User selected Cancel.
peCancelDatabaseOpen	SmallInt	Canceled open database operation.
peCancelPassword	SmallInt	Cancelled password entry

peCannotClose	SmallInt	Cannot close index.
peCannotCloseAlias	SmallInt	Alias currently in use.
peCannotCopy	SmallInt	Cannot copy selection to Clipboard.
peCannotCopyTo	SmallInt	Unable to copy to file.
peCannotCut	SmallInt	Cannot cut selection to Clipboard.
peCannotCutTo	SmallInt	Unable to cut to file.
peCannotDelete	SmallInt	Unable to delete.
peCannotDeleteLine	SmallInt	Unable to delete line.
peCannotDitto	SmallInt	Cannot duplicate field.
peCannotEdit	SmallInt	You cannot modify this field.
peCannotEditField	SmallInt	This field cannot be edited.
peCannotEditRefresh	SmallInt	Operation not valid during refresh.
peCannotExitField	SmallInt	Unable to exit field.
peCannotExitRecord	SmallInt	Unable to exit record.
peCannotInsert	SmallInt	Cannot insert record here.
peCannotInsertText	SmallInt	Unable to insert text.
peCannotLoadDriver	SmallInt	Cannot load driver.
peCannotLoadLanguageDriver	SmallInt	Cannot load language driver.
peCannotLock	SmallInt	Cannot lock record.
peCannotLockServerDependent	SmallInt	Cannot lock record dependent on server.
peCannotLookupFill	SmallInt	Unable to fill field from lookup table.
peCannotLookupFillCorr	SmallInt	Unable to fill corresponding fields from lookup table.
peCannotLookupMove	SmallInt	Unable to fill field from master table.
peCannotMakeQuery	SmallInt	Cannot create query from the selected file.
peCannotMove	SmallInt	Cannot move in that direction.
peCannotOpenClip	SmallInt	Could not open Clipboard.
peCannotOpenTable	SmallInt	Unable to open table.
peCannotOrderRange	SmallInt	Unable to set Order/Range.
peCannotPaste	SmallInt	Cannot paste from Clipboard into the selected object.
peCannotPasteFrom	SmallInt	Unable to paste from file.

peCannotPasteLink	SmallInt	Unable to paste link.
peCannotPerformAction	SmallInt	Unable to perform action.
peCannotPutField	SmallInt	The value is not legal in this field.
peCannotPutRecord	SmallInt	Record contains illegal field values.
peCannotRotate	SmallInt	Cannot rotate columns.
peCannotUndelete	SmallInt	Cannot undelete record.
peCantDropPrimary	SmallInt	The Primary Index can not be dropped since another index is maintained on the table.
peCantLoadLibrary	SmallInt	Cannot load an IDAPI service library.
peCantOpenTable	SmallInt	Could not open table '%0ds'. Engine error %ldx.
peCantSearchField	SmallInt	Unable to search in this field.
peCantSetFilter	SmallInt	A %0ds cannot be done on %lds %2ds because it is an expression index.
peCantShowDeleted	SmallInt	Table does not show deleted records.
peCantToggleToTable	SmallInt	Cannot toggle to table view.
peCfgCannotWrite	SmallInt	Cannot write to Engine configuration file.
peCfgMultiFile	SmallInt	Cannot initialize with different configuration file.
peClientsLimit	SmallInt	Too many clients.
peCompatErr	SmallInt	An error was triggered in the '%0ds' procedure.
peConstantNotFound	SmallInt	The constant name was not found.
peConversion	SmallInt	Could not convert data of type '%0cc' to type %lcc'. The types are mismatched or the values are incompatible.
peCopyLinkedTables	SmallInt	Copy linked tables?
peCopyOverSelf	SmallInt	Cannot copy a file over itself. Use rename instead.
peCorruptLockFile	SmallInt	Corrupt lock file.
peCreateErr	SmallInt	An error was triggered in a Create operation.
peCreateWarningRange	SmallInt	The %0ds of field %lds is outside range %2di - %3di. Setting it to %4di.
peCursorLimit	SmallInt	Too many open cursors.
peDBLimit	SmallInt	Too many open databases.
peDDEAllocate	SmallInt	DDE: Buffer allocation failed.
peDDEExecute	SmallInt	DDE: Execute server command failed.

peDDEInitiate	SmallInt	DDE: Specified DDE server is not responding.
peDDENoLock	SmallInt	DDE: Could not lock memory.
peDDENotOpened	SmallInt	DDE: Session not opened. Use Open.
peDDEPoke	SmallInt	DDE: Send data (poke) request failed.
peDDERequest	SmallInt	DDE: Could not receive data.
peDDETimeOut	SmallInt	DDE: Time out while waiting for data.
peDDEUnassigned	SmallInt	DDE: Server and Topic were not assigned. Use Open.
peDataLoss	SmallInt	Character(s) not supported by Table Language.
peDataTooLong	SmallInt	Data is too long for field.
peDatabaseErr	SmallInt	An error was triggered in the "%0ds" method on an object of Database type.
peDeadlock	SmallInt	A deadlock was detected.
peDeliveredDocument	SmallInt	Cannot modify this document.
peDependentMustBeEmpty	SmallInt	Cannot make this master a detail of another table if its details are not empty.
peDestMustBeIndexed	SmallInt	Destination must be indexed.
peDetailRecExistsEmpty	SmallInt	Master has detail records. Cannot empty it.
peDetailRecordsExist	SmallInt	Master has detail records. Cannot delete or modify.
peDetailTableExists	SmallInt	Detail table(s) exist.
peDetailTableOpen	SmallInt	Detail table is open.
peDiffSortOrder	SmallInt	Different sort order.
peDifferentPath	SmallInt	Tables in different directories
peDifferentTables	SmallInt	Cannot set cursor of one table to another.
peDirBusy	SmallInt	Directory is busy.
peDirInUseByOldVer	SmallInt	Directory in use by earlier version of Paradox.
peDirLocked	SmallInt	Directory is locked.
peDirNoAccess	SmallInt	Cannot access directory.
peDirNotPrivate	SmallInt	Directory is not private.
peDiskError	SmallInt	A disk error occurred: %0rs
peDivideByZero	SmallInt	Cannot divide by zero.
peDriveNotFound	SmallInt	The drive, "%0ds", either is invalid or not ready.

peDriverLimit	SmallInt	Too many active drivers.
peDriverNotLoaded	SmallInt	Driver not loaded.
peDriverUnknown	SmallInt	The driver type '%0ds' is unknown.
peDuplicateAlias	SmallInt	Duplicate alias name.
peDuplicateMoniker	SmallInt	Table alias is already in use.
peDynamicBind	SmallInt	The data type %00cc does not support dynamic binding.
peEOF	SmallInt	At end of table.
peEditObjRequired	SmallInt	Method requires an edit object.
peEmbedDataProblem	SmallInt	Object to be embedded violates data constraints when placed in container.
peEmbedNotAllowed	SmallInt	Chosen container cannot embed or disembed other objects.
peEmbedWontFit	SmallInt	Object to be embedded falls outside edges of container.
peEmptyClipboard	SmallInt	Cannot paste -- Clipboard is empty.
peEmptyTable	SmallInt	The table is empty.
peEndOfBlob	SmallInt	End of BLOB.
peEngineQueryMismatch	SmallInt	Query and Engine DLLs are mismatched.
peEnumErr	SmallInt	An error was triggered in an Enum.
peExpressionIllegal	SmallInt	Cannot use an expression for linking in this data model.
peExtInvalid	SmallInt	The destination table of the rename has an extension mismatch.
peFS_CREATEERR	SmallInt	Could not create file. Protection or access error.
peFS_WRITEOPENERR	SmallInt	Could not open output file. Protection or access error.
peFailNoError	SmallInt	You have called fail() without any error code or error message.
peFailedDatabaseOpen	SmallInt	Could not open database.
peFailedMethod	SmallInt	The method '%0ds' failed.
peFailedStdDB	SmallInt	Could not open standard database. Engine error %0dx.
peFamFileInvalid	SmallInt	Corrupt family file.
peFieldsBlank	SmallInt	Field is blank.
peFieldLimit	SmallInt	Too many fields in Table Create.
peFieldMultiLinked	SmallInt	Field(s) linked to more than one master.

peFieldMustBeTrimmed	SmallInt	Field will be trimmed, cannot put master records into PROBLEM table.
peFieldNotCurrent	SmallInt	The specified field is not the current field.
peFieldNotInEdit	SmallInt	Must be in Field View to search.
peFieldNotInLookupTable	SmallInt	Field value out of lookup table range.
peFieldValueErr	SmallInt	Could not get a field's value.
peFileBusy	SmallInt	Table is busy.
peFileCorrupt	SmallInt	Corrupt file - other than header.
peFileCreate	SmallInt	%00ds:%02ds=The file, '%3ds', could not be created. %4rs
peFileDeleteFail	SmallInt	File Delete operation failed.
peFileExists	SmallInt	File already exists.
peFileIsDirectory	SmallInt	File name is a directory name.
peFileLocked	SmallInt	File is locked.
peFileNoAccess	SmallInt	Cannot access file.
peFileNotFound	SmallInt	The file, '%0ds', does not exist.
peFilterErrAt	SmallInt	The filter has an error in field '%0di' at position '%ldi'.
peFixedType	SmallInt	You cannot change the type of a typed variable.
peFmlMemberNotFound	SmallInt	Could not find family member.
peForeignKeyErr	SmallInt	Master record missing.
peFormClosed	SmallInt	You have tried to access a document that is not open.
peFormCompileError	SmallInt	Form has PAL syntax errors. Reopening in design window.
peFormCompileErrors	SmallInt	The design object has compile errors and will not run.
peFormInvalidName	SmallInt	%0lds is not a valid name for a %0ds.
peFormInvalidOptions	SmallInt	Invalid WinStyle combination for opening the design object.
peFormNotAttached	SmallInt	You have tried to access a document that is not open.
peFormOpenFailed	SmallInt	The design object, '%0ds', could not be opened.
peFormQueryOpen	SmallInt	Cannot open query
peFormQueryViewMismatch	SmallInt	Query needs to be saved and/or re-executed
peFormTableOpen	SmallInt	Cannot open table
peFormTableReadOnly	SmallInt	This table is read-only

peFormWriteError	SmallInt	Could not write to file.
peFunctionNotFound	SmallInt	Function not found in service library.
peGeneralErr	SmallInt	Unknown error.
peGeneralSQL	SmallInt	General SQL error.
peGroupLocked	SmallInt	Key group is locked.
peHasOpenCursors	SmallInt	Table(s) open. Cannot perform this operation.
peHeaderCorrupt	SmallInt	Corrupt table/index header.
peXBadExponent	SmallInt	Invalid or missing exponent
peXBadSign	SmallInt	Invalid numeric sign
peXCantParseText	SmallInt	Cannot parse the input file: either a line is too long or has no end of line character, or no fields were found.
peXExportNoFields	SmallInt	No fields can be written to the destination table.
peXExtraCharacters	SmallInt	Number has extra characters at end
peXFieldNotInTable	SmallInt	Fixed length specification references a field not contained in the export table
peXFieldPostError	SmallInt	Unknown problem with posting field
peXForeignKeyError	SmallInt	Foreign key violation
peXInvDelFields	SmallInt	DelimitFields must be either DTDelimJustText or DTDelimAllFields.
peXInvDelimiter	SmallInt	A delimiter must be either a single character or an empty string (for none).
peXInvExtension	SmallInt	Extension not valid for this file type.
peXInvSeparator	SmallInt	A separator must be a single character.
peXInvalidCharSet	SmallInt	Character set must be either DTOEM or DTANSI.
peXInvalidDBFFFieldType	SmallInt	Field type is invalid, expecting an exportable dBase field type: C, F, N, D, or L
peXInvalidFieldSpec	SmallInt	Field specification is invalid
peXInvalidPDXFieldType	SmallInt	Field type is invalid, expecting an exportable Paradox field type: A, #, N, D, S, \$, I, L, @, or T
peXLookupTableError	SmallInt	Value does not reside in the lookup table
peXMaxValueError	SmallInt	Value too large
peXMinMaxValueError	SmallInt	Value out of range (too large or too small)

pelXMinValueError	SmallInt	Value too small
pelXMultipleSigns	SmallInt	Numbers cannot contain multiple signs
pelXNeedDiffFile	SmallInt	Cannot export a table to itself.
pelXNoDigits	SmallInt	Number does not contain any digits
pelXNotASpreadsheet	SmallInt	Operation invalid, Source file is not a spreadsheet.
pelXNotExportable	SmallInt	Don't know how to Export to this file type.
pelXNotImportable	SmallInt	Don't know how to Import from this file type.
pelXPBlockRange	SmallInt	Selected range of cells is too wide to import.
pelXPConversion	SmallInt	String conversion error on line %ld.
pelXPDataTooSparse	SmallInt	Data in the selected block is too sparse to import.
pelXPDbClose	SmallInt	Unable to close database.
pelXPDbOpen	SmallInt	Unable to open database.
pelXPExcelFileType	SmallInt	Not a supported Excel file version.
pelXPExcelIndexRecord	SmallInt	Excel Index record not found.
pelXPExportTable	SmallInt	Table selected to load is not a valid Export Specification table.
pelXPFieldCount	SmallInt	An error occurred while parsing the specification table.
pelXPFieldDesc	SmallInt	Unable to get table field descriptions.
pelXPFileClose	SmallInt	Could not close the file.
pelXPFileCreate	SmallInt	Could not create the file.
pelXPFileName	SmallInt	Not a valid file name.
pelXPFileOpen	SmallInt	Could not open the file.
pelXPFileRead	SmallInt	Could not read from file.
pelXPFileWrite	SmallInt	Could not write to file.
pelXPGetField	SmallInt	Unable to get table field.
pelXPGetProp	SmallInt	Unable to get table properties.
pelXPHome	SmallInt	Unable to set table cursor to top.
pelXPImportTable	SmallInt	Table selected to load is not a valid Import Specification table.
pelXPInputFile	SmallInt	Input file is incorrect.
pelXPInsertRecord	SmallInt	Unable to insert table record.

pelXPNextRecord	SmallInt	Unable to get next table record.
pelXPPageName	SmallInt	Invalid page name.
pelXPPassword	SmallInt	The input file is password protected.
pelXPPutField	SmallInt	Unable to put table field.
pelXPRange	SmallInt	Invalid cell range.
pelXPRecordCount	SmallInt	Unable to get table record count.
pelXPRecordInit	SmallInt	Unable to initialize table record.
pelXPRecordLength	SmallInt	Invalid record length.
pelXPRecordSize	SmallInt	Record size is limited to 32000 characters.
pelXPSkip	SmallInt	Errors encountered during import, records were skipped.
pelXPTableName	SmallInt	Not a valid table name.
pelXPTblClose	SmallInt	Unable to close database table.
pelXPTblCreate	SmallInt	Unable to create database table.
pelXPTblLock	SmallInt	Unable to lock database table.
pelXPTblOpen	SmallInt	Unable to open database table.
pelXRecordPostError	SmallInt	Unknown problem with posting record
pelXRequiredFldError	SmallInt	Required value has not been provided
pelXSigDigitLoss	SmallInt	Probably losing significant digits
pelXSignLoss	SmallInt	Possible loss of correct sign
pelXTooManyFields	SmallInt	Cannot export a table with more than 255 fields.
pelXTypeMismatch	SmallInt	Source field doesn't match destination field type
pelXUnexpectedType	SmallInt	Source field was an unexpected type which can't be translated
pelXUnsupportedTransfer	SmallInt	Unsupported transfer: Source or Destination must be a table.
pelXUserCancel	SmallInt	Operation canceled by user.
pelXFormedCalcField	SmallInt	Incorrect expression syntax.
pelXIllegalAliasProperty	SmallInt	The property '%0ds' is not associated with the alias '%1ds'.
pelXIllegalCharacter	SmallInt	Illegal character.
pelXIllegalConversion	SmallInt	Cannot convert data of type '%0cn' to %lcc.
pelXIllegalIndexName	SmallInt	Trying to create a Paradox index '%0ds' which must be named the same as the field '%1ds'.

pelIllegalIndexName1	SmallInt	Trying to create a Paradox index '%0ds' which can't be named the same as a field.
pelIllegalOpForInMem	SmallInt	Can't perform an '%0ds' on an InMemory TCursor.
pelIllegalOperator	SmallInt	An illegal PAL operator was found.
pelIllegalTableName	SmallInt	'%0ds' is not a valid table name.
pelIllegalXtabSpec	SmallInt	Crosstab specification is not allowed.
pelInUse	SmallInt	Cannot delete object(s) that are in use.
pelInappropriateFieldType	SmallInt	The specified field type is invalid.
pelInappropriateSubType	SmallInt	The specified field subtype is invalid.
pelIncompatibleDataType	SmallInt	Trying to store incompatible data type.
pelIncompatibleDataTypes	SmallInt	Data types are different when compared.
pelIncompatibleRecStructs	SmallInt	Incompatible record structures.
pelIncompleteExponent	SmallInt	Incomplete exponent.
pelIncompletePictureMatch	SmallInt	Field is not complete.
pelIncompleteSymbol	SmallInt	Incomplete symbol.
pelIncompleteXtab	SmallInt	Incomplete crosstab specification.
pelIncorrectParmFormat	SmallInt	The parameter is not formatted correctly: %0ds.
pelIndexCorrupt	SmallInt	Corrupt index.
pelIndexDoesntExist	SmallInt	Index does not exist.
pelIndexErr	SmallInt	An error was triggered in an Index operation.
pelIndexExists	SmallInt	Index already exists.
pelIndexFailed	SmallInt	Index could not be created.
pelIndexLimit	SmallInt	Too many indexes on table.
pelIndexNameRequired	SmallInt	Index name required.
pelIndexOpen	SmallInt	Index is open.
pelIndexOutOfdate	SmallInt	Index is out of date.
pelIndexReadOnly	SmallInt	Index is read only.
pelIndexStartFailed	SmallInt	Could not start Index.
pelInfiniteInsert	SmallInt	Infinite record insertion attempted
pelInterfaceVer	SmallInt	Interface mismatch. Engine version different.
pelInternal	SmallInt	An unexpected error occurred.

peInternalLimit	SmallInt	Some internal limit (see context).
peInvalidAttribute	SmallInt	Invalid File Attributes: %0ds
peInvalidBlobHandle	SmallInt	Invalid BLOB handle in record buffer.
peInvalidBlobLen	SmallInt	Invalid BLOB length.
peInvalidBlobOffset	SmallInt	Invalid offset into the BLOB.
peInvalidBookmark	SmallInt	Bookmarks do not match table.
peInvalidCallbackBufLen	SmallInt	Invalid callback buffer length.
peInvalidCfgParam	SmallInt	Invalid configuration parameter.
peInvalidChar	SmallInt	Invalid character.
peInvalidColumn	SmallInt	The specified column number is invalid.
peInvalidDBSpec	SmallInt	Invalid database alias specification.
peInvalidDataBase	SmallInt	Database not opened.
peInvalidDataTypeCompare	SmallInt	Cannot compare data types - Memo, Bitmap, OLE.
peInvalidDate	SmallInt	Invalid Date.
peInvalidDesc	SmallInt	Invalid descriptor.
peInvalidDescNum	SmallInt	Invalid descriptor number.
peInvalidDir	SmallInt	Invalid directory.
peInvalidDrive	SmallInt	Could not access drive.
peInvalidExpression	SmallInt	The filter expression is not valid
peInvalidExpressionInFld	SmallInt	The filter expression in field %s is not valid.
peInvalidExpressionWStr	SmallInt	The filter expression is not valid, %s
peInvalidFieldDesc	SmallInt	Invalid field descriptor.
peInvalidFieldName	SmallInt	Invalid field name.
peInvalidFieldType	SmallInt	Invalid field type.
peInvalidFieldXform	SmallInt	Invalid field transformation.
peInvalidFileExt	SmallInt	Invalid file extension.
peInvalidFileExtension	SmallInt	Invalid file extension for this operation: %0ds
peInvalidFileName	SmallInt	Invalid file name.
peInvalidFilter	SmallInt	Invalid Filter
peInvalidFormat	SmallInt	Invalid format.

peInvalidHandle	SmallInt	Invalid handle to the function.
peInvalidIndexCreate	SmallInt	Invalid index create request
peInvalidIndexDelete	SmallInt	Invalid index delete request
peInvalidIndexDesc	SmallInt	Invalid index descriptor.
peInvalidIndexName	SmallInt	Invalid index/tag name.
peInvalidIndexStruct	SmallInt	Invalid array of index descriptors.
peInvalidIndexType	SmallInt	Invalid index type
peInvalidIsolationLevel	SmallInt	'%0ds' is not a valid isolation level.
peInvalidKey	SmallInt	Cannot evaluate Key or Key does not pass filter condition.
peInvalidKeyword	SmallInt	Invalid use of keyword.
peInvalidLanguageDriver	SmallInt	Invalid language Driver.
peInvalidLinkExpr	SmallInt	Invalid linked cursor expression.
peInvalidMasterTableLevel	SmallInt	Master table level is incorrect.
peInvalidMode	SmallInt	Invalid mode.
peInvalidModifyRequest	SmallInt	Invalid modify request.
peInvalidOperationForTableType	SmallInt	Can't perform %0ds for table of type %1ds.
peInvalidOptParam	SmallInt	Invalid optional parameter.
peInvalidOption	SmallInt	Invalid option.
peInvalidParam	SmallInt	Invalid parameter.
peInvalidParameter	SmallInt	%00ds:%01ds:%02ds: The value of the parameter, '%3ds', is not legal. %4rs
peInvalidPassword	SmallInt	Invalid password given.
peInvalidPath	SmallInt	Invalid path.
peInvalidPrefferedFile	SmallInt	The specified file name is invalid.
peInvalidProperty	SmallInt	The specified property is invalid.
peInvalidQuery	SmallInt	Query not opened.
peInvalidRecStruct	SmallInt	Invalid record structure.
peInvalidRecordNumber	SmallInt	Invalid number of records.
peInvalidRefIntgDesc	SmallInt	Cannot change this RINTDesc.
peInvalidRefIntgStruct	SmallInt	Invalid array of referential integrity descriptors.
peInvalidRestrTableOrder	SmallInt	Invalid ordering of tables during restructure.

peInvalidRestructureOperation	SmallInt	invalid restructure operation.
peInvalidRow	SmallInt	The specified row is invalid.
peInvalidSQL	SmallInt	SQL object not opened.
peInvalidSession	SmallInt	The first argument is not a session or the session is not open.
peInvalidSessionHandle	SmallInt	Invalid session handle.
peInvalidSysData	SmallInt	Corrupt system configuration file.
peInvalidTCursor	SmallInt	TCursor not opened.
peInvalidTable	SmallInt	Invalid table.
peInvalidTableCreate	SmallInt	Invalid table create request
peInvalidTableDelete	SmallInt	Invalid table delete request
peInvalidTableLock	SmallInt	'%0ds' is not a valid Table lock.
peInvalidTableName	SmallInt	Invalid table name.
peInvalidTableVar	SmallInt	Table variable not attached.
peInvalidTime	SmallInt	Invalid Time.
peInvalidTimeStamp	SmallInt	Invalid Datetime
peInvalidTranslation	SmallInt	Translate Error. Value out of bounds.
peInvalidUserPassword	SmallInt	Unknown user name or password.
peInvalidValChkStruct	SmallInt	Invalid array of validity check descriptors.
peKeyFieldTypeMismatch	SmallInt	Foreign and primary key do not match.
peKeyOrRecDeleted	SmallInt	Record/Key deleted.
peKeyViol	SmallInt	Key violation.
peLDNotFound	SmallInt	Could not find language driver.
peLanguageDriveMisMatch	SmallInt	Language Drivers of Table and Index do not match
peLinkWontFit	SmallInt	Link information will not fit in field.
peLinkedTableProtected	SmallInt	A table linked by referential integrity requires password to open.
peListTooBig	SmallInt	Maximum number of items in a list is 2500.
peLiveQueryDead	SmallInt	Live answer set forced to disk for this operation
peLocateErr	SmallInt	An error was triggered in a Locate operation.
peLocateFailed	SmallInt	Could not perform locate operation.
peLockFileLimit	SmallInt	Lock file has grown too large.

peLockInvalid	SmallInt	Trying to '%0ds' (un)lock table '%lds' by name which can't be done in PAL.
peLockTimeout	SmallInt	Lock time out.
peLocked	SmallInt	Record locked by another user.
peLookupSrchFailed	SmallInt	Unable to find lookup value.
peLostExclusiveAccess	SmallInt	Exclusive access was lost.
peLostTableLock	SmallInt	Table lock was lost.
peMailAddressFail	SmallInt	Unable to display the address book.
peMailBadAddressIndex	SmallInt	Invalid address index: %d.
peMailBadFileIndex	SmallInt	Invalid attachment index: %d
peMailFileClose	SmallInt	Error closing attachment %s.
peMailFileOpen	SmallInt	Unable to open the attachment file %s.
peMailFileWrite	SmallInt	Unable to write the temporary attachment file %s.
peMailInvalidEditFields	SmallInt	Invalid specification of address list to edit: %d.
peMailLogoffFail	SmallInt	Unable to complete the mail system logoff.
peMailLogonFail	SmallInt	Unable to complete the mail system logon.
peMailMAPI_AccessDenied	SmallInt	MAPI: Access to mail system denied
peMailMAPI_AmbigRecip	SmallInt	MAPI: Mail recipient information is ambiguous
peMailMAPI_AmbiguousRecipient	SmallInt	MAPI: Mail recipient information is ambiguous
peMailMAPI_AttachmentNotFound	SmallInt	MAPI: The specified attachment was not found.
peMailMAPI_AttachmentOpenFailure	SmallInt	MAPI: One or more attachments could not be located.
peMailMAPI_AttachmentWriteFailure	SmallInt	MAPI: An attachment could not be written to a temporary file. Check directory permissions.
peMailMAPI_BadRecip	SmallInt	MAPI: One or more recipients were unknown. No dialog box was displayed.
peMailMAPI_BadReciptype	SmallInt	MAPI: The type of a recipient was not MAIL_ADDRT0, MAIL_ADDRCC, or MAIL_ADDRBC.
peMailMAPI_DiskFull	SmallInt	MAPI: The disk was full.
peMailMAPI_Failure	SmallInt	MAPI returned an unspecified error. Check your addresses, attachments, and/or your MAPI configuration.
peMailMAPI_InsufficientMemory	SmallInt	MAPI: There was insufficient memory to proceed.

peMailMAPI_InvalidEditfields	SmallInt	MAPI: The value of the numberOfLists parameter was outside the range of 0 to 4.
peMailMAPI_InvalidMessage	SmallInt	MAPI: An invalid message ID was provided.
peMailMAPI_InvalidRecipients	SmallInt	MAPI: One or more of the recipients in the address list was not valid.
peMailMAPI_InvalidRecips	SmallInt	MAPI: One or more of the recipients in the address list was not valid.
peMailMAPI_InvalidSession	SmallInt	MAPI: Invalid session handle - only one logon allowed per MAIL variable.
peMailMAPI_LoginFailure	SmallInt	MAPI: There was no default sign-in, and the user failed to sign in successfully when the sign-in dialog box was displayed.
peMailMAPI_MessageInUse	SmallInt	MAPI: Can not modify this message, it's in use.
peMailMAPI_NetworkFailure	SmallInt	MAPI: encountered a Network failure.
peMailMAPI_NoMessages	SmallInt	MAPI: Couldn't find a matching message.
peMailMAPI_NoRecip	SmallInt	MAPI: No name specified for an address.
peMailMAPI_NotSupported	SmallInt	MAPI: The operation was not supported by the underlying messaging system.
peMailMAPI_TextTooLarge	SmallInt	MAPI: The text in the message was too large to be sent.
peMailMAPI_TooManyFiles	SmallInt	MAPI: Too many file attachments were contained in the message. No mail was read.
peMailMAPI_TooManyRecipients	SmallInt	MAPI: There were too many recipients of the message. No mail was read.
peMailMAPI_TooManySessions	SmallInt	MAPI: Too many sessions open at once.
peMailMAPI_TypeNotSupported	SmallInt	MAPI: undocumented error occurred.
peMailMAPI_UnknownRecipient	SmallInt	MAPI: The recipient did not appear in the address list.
peMailNoMAPI	SmallInt	Unable to load the MAPI subsystem (MAPI32.DLL).
peMailNoMAPIFunction	SmallInt	Unable to load the MAPI function %s.
peMailNoMemory	SmallInt	Insufficient memory to complete this operation.
peMailResolveFail	SmallInt	Unable to resolve the specified mail addresses.
peMailSendFail	SmallInt	Mail send operation failed.
peMailUserCancel	SmallInt	The user cancelled this operation.
peMasterExists	SmallInt	Link to master table already defined.

peMasterReferenceErr	SmallInt	Self referencing referential integrity must be entered one at a time with no other changes to the table
peMasterTableOpen	SmallInt	Master table is open.
peMatchNotFound	SmallInt	"%" was not found.
peMathError	SmallInt	An arithmetic error occurred during '%0ds' execution. Reason: '%lrs'.
peMaxValErr	SmallInt	Maximum validity check failed.
peMemoCorrupt	SmallInt	Corrupt Memo/BLOB file.
peMethodNotFound	SmallInt	The method, '%0mn' is not visible from the object, '%0lun'.
peMethodNotValid	SmallInt	The method is not valid for the object.
peMinValErr	SmallInt	Minimum validity check failed.
peMisMatchedOperands	SmallInt	Cannot perform '%0ds' between %lcc and %2cc.
peMismatchArgs	SmallInt	Mismatch in the number of arguments
peModifiedSinceOpen	SmallInt	The disk file has been modified since it was loaded
peMultiLevelCascade	SmallInt	Multi-level cascade is not supported.
peMultipleInit	SmallInt	Attempt to re-initialize Engine.
peMultiplePoints	SmallInt	Only one decimal point is allowed.
peMultipleSigns	SmallInt	Only one sign is allowed.
peMultipleUniqRecs	SmallInt	Multiple records found, but only one was expected.
peMustUseBaseOrder	SmallInt	Must use baseorder for this operation.
peNA	SmallInt	Operation not applicable.
peNameNotUnique	SmallInt	Name not unique in this context.
peNameReserved	SmallInt	Name is reserved.
peNan	SmallInt	Cannot format a NAN.
peNeedExclusiveAccess	SmallInt	Table cannot be opened for exclusive use.
peNeedRestructure	SmallInt	Need to do (hard) restructure.
peNetFileLocked	SmallInt	Cannot lock network file.
peNetFileVersion	SmallInt	Wrong .NET file version.
peNetInitErr	SmallInt	Network initialization failed.
peNetMultiple	SmallInt	Directory is controlled by other .NET file.
peNetUnknown	SmallInt	Unknown network error.

peNetUserLimit	SmallInt	Network user limit exceeded.
peNo1Sep	SmallInt	Date Separator missing.
peNo1TSep	SmallInt	Time Separator missing.
peNo2Sep	SmallInt	Date Separator missing.
peNo2TSep	SmallInt	Time Separator missing.
peNo3Sep	SmallInt	Date Separator missing.
peNo3TSep	SmallInt	Time Separator missing.
peNo4Sep	SmallInt	Date Separator missing.
peNo4TSep	SmallInt	Time Separator missing.
peNo5Sep	SmallInt	Date Separator missing.
peNo5TSep	SmallInt	Time Separator missing.
peNo6TSep	SmallInt	Time Separator missing.
peNoAMPM	SmallInt	AM-PM Specification missing.
peNoActiveTransaction	SmallInt	No active transaction to commit or rollback.
peNoArguments	SmallInt	'%0ds' failed because it has no arguments supplied.
peNoAssocIndex	SmallInt	No associated index.
peNoCallback	SmallInt	No callback function.
peNoConfigFile	SmallInt	Cannot find Engine configuration file.
peNoCurrRec	SmallInt	No current record.
peNoDMChangeInRun	SmallInt	Cannot modify the Data Model in Run Mode.
peNoDay	SmallInt	Day Specification missing.
peNoDayOrMonthSpec	SmallInt	Format is display only. Need day or month.
peNoDestRecord	SmallInt	Trying to store into a nonexistent record.
peNoDetailRoom	SmallInt	Insufficient room for detail records of %s.
peNoDiskSpace	SmallInt	Insufficient disk space.
peNoFamilyRights	SmallInt	Insufficient family rights for operation.
peNoFieldRights	SmallInt	Insufficient field rights for operation.
peNoFieldRoom	SmallInt	Could not fit field %s in layout.
peNoFileHandles	SmallInt	Not enough file handles.
peNoHour	SmallInt	Hour Specification missing.

peNoHourSpec	SmallInt	Format is display only. Need hour.
peNoKeyField	SmallInt	No key field in this table.
peNoLock	SmallInt	The table(tcursor) is not '%0ds' locked.
peNoLockedRecord	SmallInt	The record is not locked
peNoLogical	SmallInt	Logical Specification missing.
peNoLookup	SmallInt	Lookup not available on this field.
peNoLookupMove	SmallInt	No master lookup available for this field.
peNoMemoView	SmallInt	Memo editing is not allowed on this field.
peNoMemory	SmallInt	Insufficient memory for this operation.
peNoMinutes	SmallInt	Minute Specification missing.
peNoMonth	SmallInt	Month Specification missing.
peNoMultConnect	SmallInt	Multiple connections not supported.
peNoNumber	SmallInt	Missing number.
peNoPage	SmallInt	Invalid page.
peNoPictureMatch	SmallInt	Invalid character(s) in this field.
peNoProperty	SmallInt	The property is not valid for the given object.
peNoRecordNos	SmallInt	The table does not support record numbers.
peNoRecords	SmallInt	Table is empty.
peNoSearchField	SmallInt	Active object is not a field or has a value that cannot be searched.
peNoSeconds	SmallInt	Seconds Specification missing.
peNoSelect	SmallInt	Must be in Field View (F2) to select.
peNoSelection	SmallInt	There is no object selected to cut or copy.
peNoSeqnums	SmallInt	Table does not support sequence numbers.
peNoServerAnsTable	SmallInt	The answer table cannot be on a server.
peNoSession	SmallInt	Database information is missing from Desktop.
peNoSoftDeletes	SmallInt	The table does not support soft deletes.
peNoSortField	SmallInt	No field identified on Sort from table '%0ds'.
peNoSrcRecord	SmallInt	Trying to read from a nonexistent record.
peNoSuchFile	SmallInt	File does not exist.
peNoSuchFilter	SmallInt	Filter handle is invalid

peNoSuchIndex	SmallInt	Index does not exist.
peNoSuchTable	SmallInt	Table does not exist.
peNoTableName	SmallInt	Specify the table to be associated for TCursor.
peNoTableRights	SmallInt	Insufficient table rights for operation. Password required.
peNoTableSupport	SmallInt	Table does not support this operation.
peNoTempFile	SmallInt	Could not create temporary table.
peNoTempTableSpace	SmallInt	Temporary table resource limit.
peNoTextTable	SmallInt	Unrecognized table type
peNoTransaction	SmallInt	No user transaction is currently in progress.
peNoUniqueRecs	SmallInt	Table does not support this operation because it is not uniquely indexed.
peNoWeekday	SmallInt	Day of Week Specification missing.
peNoWorkPrivAlias	SmallInt	Cannot change the path of the default working or private directories.
peNoYear	SmallInt	Year Specification missing.
peNonExistentAlias	SmallInt	Invalid alias.
peNotABlob	SmallInt	Field is not a BLOB.
peNotAValidField	SmallInt	The field number or name is not in the table.
peNotAllowedFieldType	SmallInt	Field '%0ds' of type '%lds' is not a valid type for a sort or index operation.
peNotAllowedInPlace	SmallInt	This operation is not allowed while in place.
peNotCoEdit	SmallInt	Table needs to be in Edit mode to perform operation.
peNotCurSession	SmallInt	Operation must be performed on the current session.
peNotEnoughRights	SmallInt	Cannot perform operation '%0ds' on '%lds' because of insufficient rights.
peNotField	SmallInt	Field '%0ds' is not a field in table '%lds'.
peNotFieldNum	SmallInt	Field '%0di' is not a field in table '%lds'.
peNotImplemented	SmallInt	Not implemented yet.
peNotInEditMode	SmallInt	Not in Edit mode. Press F9 to edit data.
peNotInRunMode	SmallInt	Document is not in run mode.
peNotIndexed	SmallInt	Table is not indexed.
peNotInitialized	SmallInt	Engine not initialized.

peNotLiveView	SmallInt	Not a live query view.
peNotLocked	SmallInt	Object not locked.
peNotNumericField	SmallInt	'%0ds' is not a numeric field in '%ltn'.
peNotOleField	SmallInt	This field is not an OLE field.
peNotOnANetwork	SmallInt	Not on a network. Not logged in or wrong network driver.
peNotOnThatNet	SmallInt	Not on the network.
peNotOpenIndex	SmallInt	'%0ds' requires that the table '%lds' be opened on an index.
peNotSameSession	SmallInt	Attempt to mix objects from different sessions.
peNotSuffSQLRights	SmallInt	Insufficient SQL rights for operation.
peNotSupported	SmallInt	Capability not supported.
peNotSupportedFiltered	SmallInt	Operation not supported on filtered record set
peNotValidSearchField	SmallInt	Illegal field type for '%0ds' in '%lds' for locate.
peNullFieldName	SmallInt	Field has no name.
peOSAccessDenied	SmallInt	Permission denied.
peOSArgListTooLong	SmallInt	Argument list is too long.
peOSBadFileNo	SmallInt	Bad file number.
peOSCrossDevLink	SmallInt	Cross-device link.
peOSDriveNotReady	SmallInt	Drive not ready.
peOSExecFmt	SmallInt	Execution format error.
peOSFileExist	SmallInt	File already exists.
peOSInt24Fail	SmallInt	Critical DOS Error.
peOSInvalidAccCode	SmallInt	Invalid access code.
peOSInvalidArg	SmallInt	Invalid argument.
peOSInvalidData	SmallInt	Invalid data.
peOSInvalidEnviron	SmallInt	Invalid environment.
peOSInvalidFormat	SmallInt	Invalid format.
peOSInvalidFunc	SmallInt	Invalid function number.
peOSInvalidMemAddr	SmallInt	Invalid memory block address.
peOSLockViol	SmallInt	Lock violation.
peOSMathArg	SmallInt	Math argument.

peOSMemBlocksDestroyed	SmallInt	Memory blocks destroyed.
peOSNetErr	SmallInt	Operating system network error.
peOSNoDevice	SmallInt	Device does not exist.
peOSNoFATEntry	SmallInt	File or directory does not exist.
peOSNoMemory	SmallInt	Not enough memory.
peOSNoMoreFiles	SmallInt	No more files.
peOSNoPath	SmallInt	Path not found.
peOSNotSameDev	SmallInt	Not same device.
peOSOutOfRange	SmallInt	Result is too large.
peOSRemoveCurDir	SmallInt	Attempt to remove current directory.
peOSShareViol	SmallInt	Share violation.
peOSTooManyOpenFiles	SmallInt	Too many open files. You may need to increase MAXFILEHANDLE limit in IDAPI configuration.
peOSUnknown	SmallInt	Unknown internal operating system error.
peObjImplicitlyDropped	SmallInt	Object implicitly dropped.
peObjImplicitlyModified	SmallInt	Object implicitly modified.
peObjMayBeTruncated	SmallInt	Object may be truncated.
peObjNotFound	SmallInt	Could not find object.
peObjectDisabled	SmallInt	Object disabled.
peObjectImplicitlyTruncated	SmallInt	Object implicitly truncated.
peObjectNotFound	SmallInt	You have tried to reference the object named '%0ds' from the object named '%1un'. The referenced object could not be found. The name is either incorrect or the object is not visible from '%2ds'.
peObjectTreeTooBig	SmallInt	Too many objects for object tree.
peOldVersion	SmallInt	Older version (see context).
peOle2AccessMethod	SmallInt	Error accessing the method '%0ds'.
peOle2AccessProperty	SmallInt	Error accessing the property '%0ds'.
peOle2BadConnection	SmallInt	The OLE connection is no longer valid. Perhaps the server application was closed.
peOle2BadParameterCount	SmallInt	An OLE method was called with wrong parameter count.
peOle2BadPropertyIndex	SmallInt	Bad Index used to access the OLE property '%0ds'.

peOle2NoInterface	SmallInt	Server '%0ds' has no programmable interface.
peOle2NoOLEType	SmallInt	Cannot convert '%0cc' to an OLE type.
peOle2NoObjectPalType	SmallInt	Error converting OLE type to ObjectPAL type.
peOle2NoReturnValue	SmallInt	The OLE method '%0ds' has no return value.
peOle2NoTypeInfo	SmallInt	OLE server has no type information.
peOle2NoTypeLibrary	SmallInt	Server '%0ds' does not have a registered type library.
peOle2NoValueByReference	SmallInt	Cannot pass value property by reference.
peOle2NotAServer	SmallInt	OleAuto only bound to type information, no actual server is activated.
peOle2NotCollection	SmallInt	OLE object is not a collection object.
peOle2NotImplemented	SmallInt	OLE functionality is not implemented.
peOle2OcxUnavailable	SmallInt	OCX is unavailable for access.
peOle2OpenServer	SmallInt	Error opening server '%0ds'.
peOle2OpenTypeLibrary	SmallInt	Failed to open OLE type library.
peOle2ParameterOverflow	SmallInt	Overflow during conversion of argument %0di of OLE method '%lds'.
peOle2PropertyMismatch	SmallInt	Type mismatch in access of OLE property '%0ds'.
peOle2PropertyOverflow	SmallInt	Overflow during conversion of OLE property '%0ds'.
peOle2ReturnValueOverflow	SmallInt	Overflow during conversion of return value of OLE method '%0ds'.
peOle2TypeLibraryNotExist	SmallInt	OLE server has no type library.
peOle2TypeMismatch	SmallInt	Type mismatch in argument %0di, in call of the OLE method '%lds'.
peOle2TypeRegistryErr	SmallInt	Incorrect type library registration for server '%0ds'.
peOle2UnknownException	SmallInt	Unknown exception fault in OLE server.
peOle2UnknownMethod	SmallInt	Attempt to call a Method '%0ds' unknown to OLE server.
peOle2UnknownProperty	SmallInt	Attempt to access a Property '%0ds' unknown to OLE server.
peOle2UnknownServerName	SmallInt	Attempt to open unknown server '%0ds'.
peOpenBlobLimit	SmallInt	Too many open BLOBs.
peOpenDetailFailed	SmallInt	Detail Table Open operation failed.
peOpenErr	SmallInt	Cannot open file
peOpenLookupFailed	SmallInt	Lookup Table Open operation failed.

peOpenMasterFailed	SmallInt	Master Table Open operation failed.
peOpenTableLimit	SmallInt	Too many open tables.
peOpenedByPal	SmallInt	This table view was opened by ObjectPal.
peOperatorNotAllowed	SmallInt	Operation ‘%0ds’ is not allowed on the data type %lcc.
peOptRecLockFailed	SmallInt	Couldn’t perform the edit because another user changed the record.
peOptRecLockRecDel	SmallInt	Couldn’t perform the edit because another user deleted or moved the record.
peOsAlreadyLocked	SmallInt	Record already locked by this workstation.
peOsNotLocked	SmallInt	Record not locked.
peOsUnknownSrvErr	SmallInt	Error from NOVELL file server.
peOutOfHandles	SmallInt	Out of internal file handles ??
peOutOfRange	SmallInt	Number is out of range.
peOverflow	SmallInt	Overflow. The source data is numerically too large (positive or negative) to store in the destination.
pePart1Sep	SmallInt	Incomplete Date Separator .
pePart1Tsep	SmallInt	Incomplete Time Separator .
pePart2Sep	SmallInt	Incomplete Date Separator .
pePart2Tsep	SmallInt	Incomplete Time Separator .
pePart3Sep	SmallInt	Incomplete Date Separator .
pePart3Tsep	SmallInt	Incomplete Time Separator .
pePart4Sep	SmallInt	Incomplete Date Separator .
pePart4Tsep	SmallInt	Incomplete Time Separator .
pePart5Sep	SmallInt	Incomplete Date Separator .
pePart5Tsep	SmallInt	Incomplete Time Separator .
pePart6Tsep	SmallInt	Incomplete Time Separator .
pePartAMPM	SmallInt	Incomplete AM-PM Specification.
pePartDay	SmallInt	Incomplete Day Specification.
pePartHour	SmallInt	Incomplete Hour Specification.
pePartLogical	SmallInt	Incomplete Logical Specification.
pePartMinutes	SmallInt	Incomplete Minute Specification.

pePartMonth	SmallInt	Incomplete Month Specification.
pePartSeconds	SmallInt	Incomplete Seconds Specification.
pePartWeekday	SmallInt	Incomplete Day of Week Specification.
pePartYear	SmallInt	Incomplete Year Specification.
pePasswordLimit	SmallInt	Too many passwords.
pePasswordRequired	SmallInt	Password required
pePasteNeedPage	SmallInt	You can paste page only from Clipboard before a selected page.
pePastePage	SmallInt	Clipboard object can be pasted only into a Form.
pePathNonExistent	SmallInt	The path '%0ds' does not exist.
pePathNotFound	SmallInt	The path, '%0ds', does not exist.
pePdxIOTable	SmallInt	Paradox 1.0 tables are not supported.
pePdx35IdDriver	SmallInt	Needs Paradox 3.5-compatible language driver.
pePdxDriverNotActive	SmallInt	Paradox driver not active.
pePictureErr	SmallInt	The field value fails picture validity check.
pePrecisionExceeded	SmallInt	Number is out of range for the given type.
pePrimaryKeyRedefine	SmallInt	Cannot redefine primary key.
pePrnInvalidDriver	SmallInt	Invalid printer driver.
pePrnNoDriver	SmallInt	Cannot find printer driver.
pePrnNoMemory	SmallInt	Insufficient memory.
pePrnNoPrinter	SmallInt	No printer installed or Windows cannot print.
pePropertyAccess	SmallInt	Cannot access property.
pePropertyBadValue	SmallInt	Attempted to assign an illegal value to the property.
pePropertyGet	SmallInt	An error occurred when trying to get the property named '%0ds' of the object named '%1un' of type '%2uc'.
pePropertyNotFound	SmallInt	You have tried to access the property named '%0up' which does not belong to the object named '%1un' of type '%2uc'.
pePropertySet	SmallInt	An error occurred when setting the property named '%0ds' of the object named '%1un' of type '%2uc'.
pePublicAliasExists	SmallInt	Alias(es) already defined -- discarding new ones.
peQBETerminated	SmallInt	QBE terminated by user.peQBEbadFileName.
peQryAmbOutPr	SmallInt	obsolete

peQryAmbSymAs	SmallInt	obsolete
peQryAmbigJoAsy	SmallInt	obsolete
peQryAmbigJoSym	SmallInt	obsolete
peQryAmbigOutEx	SmallInt	Ambiguous use of ! (inclusion operator).
peQryAseToPer	SmallInt	A SET operation cannot be included in its own grouping.
peQryAveNumDa	SmallInt	Only numeric and date/time fields can be averaged.
peQryBadExprI	SmallInt	Invalid expression.
peQryBadFieldOr	SmallInt	Invalid OR expression.
peQryBadFormat	SmallInt	Query format is not supported.
peQryBadVName	SmallInt	obsolete
peQryBitmapErr	SmallInt	bitmap
peQryBlobErr	SmallInt	General BLOB error.
peQryBlobTerm	SmallInt	Operation is not supported on BLOB fields.
peQryBuffTooSmall	SmallInt	Buffer too small to contain query string.
peQryCalcBadR	SmallInt	CALC expression cannot be used in INSERT, DELETE, CHANGETO and SET rows.
peQryCalcType	SmallInt	Type error in CALC expression.
peQryCancExcept	SmallInt	Query canceled.
peQryChNamBig	SmallInt	Cannot perform operation on CHANGED table together with a CHANGETO query.
peQryChgToIti	SmallInt	CHANGETO can be used in only one query form at a time.
peQryChgToChg	SmallInt	Cannot modify CHANGED table.
peQryChgToExp	SmallInt	A field can contain only one CHANGETO expression.
peQryChgToIns	SmallInt	A field cannot contain more than one expression to be inserted.
peQryChgToNew	SmallInt	obsolete
peQryChgToVal	SmallInt	CHANGETO must be followed by the new value for the field.
peQryChkmrkFi	SmallInt	Checkmark or CALC expressions cannot be used in FIND queries.
peQryChunkErr	SmallInt	chunk
peQryColum255	SmallInt	More than 255 fields in ANSWER table.

peQryConAftAs	SmallInt	AS must be followed by the name for the field in the ANSWER table.
peQryDBExcept	SmallInt	Unexpected Database Engine error.
peQryDelItime	SmallInt	DELETE can be used in only one query form at a time.
peQryDelAmbig	SmallInt	Cannot perform operation on DELETED table together with a DELETE query.
peQryDelFrDel	SmallInt	Cannot delete from the DELETED table.
peQryEgFieldType	SmallInt	Example element is used in two fields with incompatible types or with a BLOB.
peQryEmpty	SmallInt	Query string is empty.
peQryExaminOr	SmallInt	Cannot use example elements in an OR expression.
peQryExprTyps	SmallInt	Expression in this field has the wrong type.
peQryExtraCom	SmallInt	Extra comma found.
peQryExtraOro	SmallInt	Extra OR found.
peQryExtraQro	SmallInt	One or more query rows do not contribute to the ANSWER.
peQryFatalExcept	SmallInt	Unexpected exception.
peQryFindIAtt	SmallInt	FIND can be used in only one query form at a time.
peQryFindAnsT	SmallInt	FIND cannot be used with the ANSWER table.
peQryGrpNoSet	SmallInt	A row with GROUPBY must contain SET operations.
peQryGrpStRow	SmallInt	GROUPBY can be used only in SET rows.
peQryIdfPerli	SmallInt	Use only one INSERT, DELETE, SET or FIND per line.
peQryIdfinlco	SmallInt	Use only INSERT, DELETE, SET or FIND in leftmost column.
peQryInAnExpr	SmallInt	Syntax error in expression.
peQryInsITime	SmallInt	INSERT can be used in only one query form at a time.
peQryInsAmbig	SmallInt	Cannot perform operation on INSERTED table together with an INSERT query.
peQryInsDelCh	SmallInt	INSERT, DELETE, CHANGETO and SET rows may not be checked.
peQryInsExprR	SmallInt	Field must contain an expression to insert (or be blank).
peQryInsToIns	SmallInt	Cannot insert into the INSERTED table.
peQryIsArray	SmallInt	Variable is an array and cannot be accessed.
peQryLabelErr	SmallInt	Label

peQryLinkCalc	SmallInt	Rows of example elements in CALC expression must be linked.
peQryLngvName	SmallInt	Variable name is too long.
peQryLongExpr	SmallInt	Expression is too long.
peQryLongQury	SmallInt	Query may take a long time to process.
peQryMemExcept	SmallInt	Not enough memory to finish operation.
peQryMemVProc	SmallInt	Reserved word or one that can't be used as a variable name.
peQryMisSrtQu	SmallInt	Missing right quote.
peQryMisngCom	SmallInt	Missing comma.
peQryMisngRpa	SmallInt	Missing).
peQryNIY	SmallInt	Feature not implemented yet in query.
peQryNamTwice	SmallInt	Cannot specify duplicate column names.
peQryNoAnswer	SmallInt	This DELETE, CHANGE or INSERT query has no ANSWER.
peQryNoChkmar	SmallInt	Query has no checked fields.
peQryNoDefOcc	SmallInt	Example element has no defining occurrence.
peQryNoGroups	SmallInt	No grouping is defined for SET operation.
peQryNoPatter	SmallInt	Cannot use patterns in this context.
peQryNoQryToPrep	SmallInt	Attempted to prepare an empty query.
peQryNoSuchDa	SmallInt	Date does not exist.
peQryNoValue	SmallInt	Variable has not been assigned a value.
peQryNonsense	SmallInt	Query makes no sense.
peQryNotHandle	SmallInt	Function called with bad query handle.
peQryNotParse	SmallInt	Query was not previously parsed or prepared.
peQryNotPrep	SmallInt	Query is not prepared. Properties unknown.
peQryOnlyCons	SmallInt	Invalid use of example element in summary expression.
peQryOnlySetR	SmallInt	Incomplete query statement. Query only contains a SET definition.
peQryOutSensl	SmallInt	Example element with ! makes no sense in expression.
peQryOutTwicl	SmallInt	Example element cannot be used more than twice with a ! query.
peQryPaRowCnt	SmallInt	Row cannot contain expression.

peQryPersePar	SmallInt	obsolete
peQryProcPlsw	SmallInt	obsolete
peQryPwlnsrts	SmallInt	No permission to insert or delete records.
peQryPwModrts	SmallInt	No permission to modify field.
peQryQbeFieldFound	SmallInt	Field not found in table.
peQryQbeNoFence	SmallInt	Expecting a column separator in table header.
peQryQbeNoFenceT	SmallInt	Expecting a column separator in table.
peQryQbeNoHeaderT	SmallInt	Expecting column name in table.
peQryQbeNoTab	SmallInt	Expecting table name.
peQryQbeNumCols	SmallInt	Expecting consistent number of columns in all rows of table.
peQryQbeOpentab	SmallInt	Cannot open table.
peQryQbeTwice	SmallInt	Field appears more than once in table.
peQryQualnDel	SmallInt	DELETE rows cannot contain quantifier expression.
peQryQualnIns	SmallInt	Invalid expression in INSERT row.
peQryQxFieldCount	SmallInt	Query extended syntax field count error.
peQryQxFieldSymNotFound	SmallInt	Field name in sort or field clause not found.
peQryQxTableSymNotFound	SmallInt	Table name in sort or field clause not found.
peQryRaglnIns	SmallInt	Invalid expression in INSERT row.
peQryRaglnSet	SmallInt	Invalid expression in SET definition.
peQryRefresh	SmallInt	Refresh exception during query.
peQryRegister	SmallInt	Lock failure.
peQryRestartQry	SmallInt	Query must be restarted.
peQryRowUsErr	SmallInt	row use
peQrySetExpec	SmallInt	SET keyword expected.
peQrySetVAmbI	SmallInt	Ambiguous use of example element.
peQrySetVBadI	SmallInt	obsolete
peQrySetVDefI	SmallInt	obsolete
peQrySumNumbe	SmallInt	Only numeric fields can be summed.
peQrySyntErr	SmallInt	QBE syntax error.
peQryTableIsWP3	SmallInt	Table is write protected.

peQryTokenNot	SmallInt	Token not found.
peQryTwoOutrl	SmallInt	Cannot use example element with ! more than once in a single row.
peQryTypeMIsM	SmallInt	Type mismatch in expression.
peQryUnknownAnsType	SmallInt	Unknown answer table type.
peQryUnrelQl	SmallInt	Query appears to ask two unrelated questions.
peQryUnusedSt	SmallInt	Unused SET row.
peQryUseInsDe	SmallInt	INSERT, DELETE, FIND, and SET can be used only in the leftmost column.
peQryUseOfChg	SmallInt	CHANGETO cannot be used with INSERT, DELETE, SET or FIND.
peQryVarMustF	SmallInt	Expression must be followed by an example element defined in a SET.
peQueryErr	SmallInt	An error was triggered in the '%0ds' method on an object of Query type.
peQueryView	SmallInt	This table view is a query view.
peREGExpressionTooLarge	SmallInt	Matching error: Expression is too big.
peREGInvalidBracketRange	SmallInt	Matching error: Invalid bracket range.
peREGNestedSQP	SmallInt	Matching error: Nested operand.
peREGOperandEmpty	SmallInt	Matching error: Operand is empty.
peREGSPQFollowsNothing	SmallInt	Matching error: *, +, ? must follow an expression.
peREGTooManyParens	SmallInt	Matching error: Too many parentheses.
peREGTrailingBackSlash	SmallInt	Matching error: Trailing back slash.
peREGUnmatchedBrackets	SmallInt	Matching error: Unmatched brackets.
peREGUnmatchedParens	SmallInt	Matching error: Unmatched parentheses.
peReadErr	SmallInt	Read failure.
peReadOnlyDB	SmallInt	Database is read only.
peReadOnlyDir	SmallInt	This directory is read only.
peReadOnlyField	SmallInt	Trying to modify read-only field.
peReadOnlyProperty	SmallInt	You do not have write access to this property. It is read only. It cannot be modified.
peRecAlreadyLocked	SmallInt	This record is already locked by another module in this session.

peRecLockFailed	SmallInt	Cannot post record out of range.
peRecLockLimit	SmallInt	Too many record locks on table.
peRecMoved	SmallInt	Record moved because key value changed.
peRecNotFound	SmallInt	Could not find record.
peRecTooBig	SmallInt	Record size is too big for table.
peRecordAlreadyLocked	SmallInt	Warning: Record already locked in this session.
peRecordGroupConflict	SmallInt	Conflicting record lock in this session.
peRecordIsDeleted	SmallInt	Record is deleted.
peRecordIsNotDeleted	SmallInt	Record is not deleted.
peRecordNotLocked	SmallInt	This record is not locked so the operation is invalid.
peRefIntgReqIndex	SmallInt	Referential integrity fields must be indexed.
peReqOptParamMissing	SmallInt	Optional parameter is required.
peReqSameTableTypes	SmallInt	Operation requires the same table types.
peReq_WLock_TC	SmallInt	The operation you are trying to perform requires write-lock access to the table which could not be achieved.
peReq_XLock_TC	SmallInt	The operation you are trying to perform requires exclusive-lock access to the table which could not be achieved.
peReqdErr	SmallInt	Field value required.
peRequiredField	SmallInt	This field cannot be blank.
peRequiresPDOxtable	SmallInt	The table created must be a Paradox table type.
peReservedOsName	SmallInt	Name is reserved by DOS.
peSKCantInstallHook	SmallInt	Cannot do sendKeys() while another sendKeys() is already playing.
peSKInvalidCount	SmallInt	The repeat count is not correct.
peSKInvalidKey	SmallInt	The key name is not correct.
peSKMissingCloseBrace	SmallInt	Missing closing brace.
peSKMissingCloseParen	SmallInt	Missing closing parenthesis.
peSKStringTooLong	SmallInt	The keys string is too long.
peSQL	SmallInt	An error was triggered in the "%0ds" method on an object of SQL type.
peSQLCommLost	SmallInt	Lost communication with SQL server.

peSameTable	SmallInt	Table Names the same.
peSearchColReqd	SmallInt	Non-blob column in table required to perform operation.
peSerNumLimit	SmallInt	Serial number limit (Paradox).
peServerNoMemory	SmallInt	NOVELL server out of memory.
peServerPathIllegal	SmallInt	A server alias does not have a path.
peSessionErr	SmallInt	An error was triggered in the '%0ds' method on an object of Session type.
peSessionsLimit	SmallInt	Too many sessions.
peShareNotLoaded	SmallInt	SHARE not loaded. It is required to share local files.
peSharedFileAccess	SmallInt	Not initialized for accessing network files.
peSortErr	SmallInt	An error was triggered in a Sort operation.
peSortFailed	SmallInt	Sort from '%0ds' to '%1ds' could not be performed.
peSortStartFailed	SmallInt	Could not start Sort.
peSrvAccessDenied	SmallInt	Access to requested file denied.
peSrvCannotGetLock	SmallInt	Unable to acquire lock.
peSrvCapacityLimit	SmallInt	Internal catalog size exceeded.
peSrvCopyFailed	SmallInt	Could not copy to the specified file.
peSrvDeleteFailed	SmallInt	Could not delete the specified file.
peSrvDiskError	SmallInt	Error occurred while accessing file from disk.
peSrvFileDoesNotExist	SmallInt	Could not find requested file.
peSrvFormat	SmallInt	Error writing file. Record not tagged.
peSrvGraphicPasteFailed	SmallInt	Unable to paste graphic.
peSrvInvalidCount	SmallInt	File is corrupt. Record tag error.
peSrvInvalidExtension	SmallInt	Invalid file extension for this file type.
peSrvInvalidHandle	SmallInt	Internal invalid handle.
peSrvInvalidName	SmallInt	File name is invalid.
peSrvMemoryAllocation	SmallInt	Out of memory.
peSrvNameTooLong	SmallInt	File name is too long.
peSrvNoReadRights	SmallInt	No read access to file.
peSrvNoWriteRights	SmallInt	File does not exist or is write protected.
peSrvNotSameDevice	SmallInt	Rename not allowed to a different device.

peSrvOCXControlCreateError	SmallInt	Error creating ActiveX control. Please check installation.
peSrvOCXControlNotFound	SmallInt	Specified ActiveX control not found in registry or in registered location. Please check installation.
peSrvOleActivateFailed	SmallInt	Cannot activate OLE server.
peSrvOleCantUpdateNow	SmallInt	Unable to update OLE object.
peSrvOleInsertObjectFailed	SmallInt	Unable to insert OLE object.
peSrvOlePasteFailed	SmallInt	Unable to paste OLE object.
peSrvOlePasteLinkFailed	SmallInt	Unable to paste link OLE object.
peSrvOpen	SmallInt	File does not exist or is read protected.
peSrvPathNotFound	SmallInt	Could not find requested path.
peSrvRead	SmallInt	Disk error occurred while reading file.
peSrvRenameFailed	SmallInt	Could not rename the specified file.
peSrvTextPasteFailed	SmallInt	Unable to paste text.
peSrvUnknowError	SmallInt	Internal error.
peSrvUseCountLimit	SmallInt	Internal catalog usecount error.
peSrvWrite	SmallInt	Error occurred while writing to file. Check disk space.
peStackOverflow	SmallInt	Stack overflow. Your method or procedures are nested too deeply.
peStringTooLong	SmallInt	String too long. Cannot exceed %0di characters.
peSysCorrupt	SmallInt	Data structure corruption.
peSysFileIO	SmallInt	I/O error on a system file.
peSysFileOpen	SmallInt	Cannot open a system file.
peSysReEntered	SmallInt	System has been illegally re-entered.
peTCursorAttach	SmallInt	Could not attach TCursor to another object.
peTCursorErr	SmallInt	An error was triggered in the "%0ds" method on an object of TCursor type.
peTableCursorLimit	SmallInt	Too many cursors per table.
peTableEncrypted	SmallInt	Encrypted dBASE tables not supported.
peTableErr	SmallInt	An error was triggered in the "%0ds" method of an object of Table type.
peTableExists	SmallInt	Table already exists.
peTableFull	SmallInt	Table is full.

peTableInUse	SmallInt	Table is in use.
peTableLevelChanged	SmallInt	Table level changed.
peTableMismatch	SmallInt	Tables are not equivalent.
peTableOpen	SmallInt	Table is open.
peTableProtected	SmallInt	'%0ds' is already protected. Did not provide password.
peTableReadOnly	SmallInt	Table is read only.
peTableSQL	SmallInt	SQL replicas not supported.
peTableViewTableReadOnly	SmallInt	Table is read only.
peTableLockLimit	SmallInt	Too many table locks.
peTblNotImplemented	SmallInt	'%0ds' has not been implemented.
peTblUtilInUse	SmallInt	Cannot perform utility while table is in use.
peTextWontFit	SmallInt	Text will not fit in field.
peTooFewSeries	SmallInt	Surface chart needs two or more series.
peTooManyTables	SmallInt	Crosstab or Query uses too many tables.
peToolsRead	SmallInt	Could not read the style sheet.
peToolsWrite	SmallInt	Could not save the style sheet.
peTransactionImbalance	SmallInt	Transaction mismatch -- cannot commit changes
peTransactionNA	SmallInt	Transactions are not supported by this database.
peUIObjectErr	SmallInt	An error was triggered in the '%0ds' method on an object of UIObject type.
peUnassigned	SmallInt	You have tried to use an unassigned variable. A variable must be assigned a value before you can use it.
peUnboundXtab	SmallInt	Crosstab has no defining table.
peUnknownDB	SmallInt	Unknown database.
peUnknownDBType	SmallInt	Unknown database type.
peUnknownDataBase	SmallInt	The database or alias supplied for opening a TCursor was not known.
peUnknownDriver	SmallInt	Driver not known to system.
peUnknownExtension	SmallInt	Cannot recognize file extension.
peUnknownFieldName	SmallInt	The specified field name is invalid.
peUnknownFieldNum	SmallInt	The specified field number is invalid.

peUnknownFile	SmallInt	Cannot open file.
peUnknownIndex	SmallInt	%0ds %1ds is an unknown index.
peUnknownNetType	SmallInt	Network type unknown.
peUnknownTableType	SmallInt	Unknown table type.
peUnknownVersion	SmallInt	Cannot read file -- version is too high.
peUnlockFailed	SmallInt	Unlock failed.
peUnsupportedOption	SmallInt	This printer does not support the setting: %0ds.
peUntranslatableCharacters	SmallInt	Character(s) not supported by Table Language.
peUpdateNoIndex	SmallInt	The Add or Sub routines require an indexed destination table in order to do updates.
peUseCount	SmallInt	Table has too many users.
peValFieldModified	SmallInt	Validity check field modified.
peValFileCorrupt	SmallInt	Corrupt or missing .VAL file.
peValFileInvalid	SmallInt	.VAL file is out of date.
peValidateData	SmallInt	Should field constraints be checked?
peVendInitFail	SmallInt	Vendor initialization failed.
peWorkStationSessionLimit	SmallInt	Too many sessions from this workstation.
peWriteErr	SmallInt	Write failure.
peWriteOnlyProperty	SmallInt	You do not have read access to this property. It is write only. It cannot be read.
peWrongDriverName	SmallInt	Wrong driver name.
peWrongDriverType	SmallInt	Wrong driver type.
peWrongDriverVer	SmallInt	Wrong driver version.
peWrongObjectVersion	SmallInt	Object could not be read. Continuing read.
peWrongSysVer	SmallInt	Wrong system version.
peWrongTable	SmallInt	Preferred report is not for this table. Generating a default report.
peXtabAnswerError	SmallInt	Error in crosstab ANSWER table.
peXtabNotRunning	SmallInt	Cannot save the table while the crosstab is not running.

ExecuteOptions constants

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 active window remains active.
ExeShowNoActivate	SmallInt	Displays the Application Window at its most recent size and position. The active window remains active.
ExeShowNormal	SmallInt	Activates and displays a window

FieldDisplayTypes constants

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

FileBrowserFileTypes constants

Constant	Data type	Description
fbAllTables	LongInt	All table types supported by Paradox (*.db, *.dbf, etc.)
fbASCII	LongInt	Delimited Text files (*.txt)
fbBitmap	LongInt	Bitmap graphics (*.bmp)

fbDBase	LongInt	dBASE tables (*.dbf)
fbDLL (8)	LongInt	Dynamic Link Libraries (*.dll)
fbDM (5.0)	LongInt	Data model files (*.dm)
fbExcel	LongInt	Excel worksheets (*.xls)
fbFiles	LongInt	All files (*.*)
fbForm	LongInt	Paradox forms (*.fsl, *.fdl)
fbGraphic	LongInt	Graphic files (*.bmp, *.eps, *.gif, *.pcx, *.tif)
fbHTML (8)	LongInt	HTML files (*.htm)
fbHTMLTemplate (8)	LongInt	HTML template files (*.htt)
fbIni	LongInt	Initialization files (*.ini)
fbLibrary	LongInt	ObjectPAL libraries (*.lsl, *.ldl)
fbLotus1	LongInt	Lotus 1-2-3 version 1 worksheets (*.wks)
fbLotus2	LongInt	Lotus 1-2-3 version 2 worksheets (*.wkl)
fbOCX (8)	LongInt	ActiveX Controls (*.OCX)
fbParadox	LongInt	Paradox tables (*.db)
fbQuattro	LongInt	Quattro worksheets (*.wkq)
fbQuattroPro	LongInt	Quattro Pro worksheets (*.wql)
fbQuattroProWindows	LongInt	Quattro Pro for Windows notebooks (*.wbl)
fbQuery	LongInt	Query files (*.qbe)
fbReport	LongInt	Paradox reports (*.rsl, *.rdl)
fbScreenStyle (5.0)	LongInt	Form style sheets (*.ft)
fbScript	LongInt	ObjectPAL scripts (*.ssl, *.sdl)
fbSimpleText (8)	LongInt	Text files (*.txt)
fbSQL (5.0)	LongInt	SQL files (*.sql)
fbTable	LongInt	All table types supported by Paradox (*.db, *.dbf, etc.)
fbTableView	LongInt	Paradox table view files (*.tv)
fbText	LongInt	All text files (*.txt, *.pxt, *.rtf)

FontAttributes constants

Constant	Data type	Description
FontAttribBold	SmallInt	bold
FontAttribItalic	SmallInt	italic
FontAttribNormal	SmallInt	normal
FontAttribStrikeOut	SmallInt	strike out
FontAttribUnderline	SmallInt	underline

FrameStyles constants

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
Windows3dFrame (5.0)	SmallInt	Uses the default Windows 3D frame style
Windows3dGroup (5.0)	SmallInt	Uses the default Windows 3D group border

General constants

Constant	Data type	Description
No	Logical	False
Off	Logical	False
On	Logical	True

Pi	Number	3.14159265358979323846
Yes	Logical	True

GraphBindTypes constants

Constant	Data type	Description
Graph1DSummary	SmallInt	Specifies a one-dimensional summary chart and enables summary operators
Graph2DSummary	SmallInt	Specifies a two-dimensional summary chart and enables summary operators and group-by specification
GraphTabular	SmallInt	Specifies a tabular chart (default)

GraphicMagnification constants

Constant	Data type	Description
Magnify100	SmallInt	Displays the chart at its actual size
Magnify200	SmallInt	Displays the chart at twice its actual size
Magnify25	SmallInt	Displays the chart at a quarter of its actual size
Magnify400	SmallInt	Displays the chart at four times its actual size
Magnify50	SmallInt	Displays the chart at half its actual size
MagnifyBestFit	SmallInt	Resizes the chart as necessary to fit the chart in the frame

GraphLabelFormats constants

Constant	Data type	Description
GraphHideY	SmallInt	Hides the Y-value (2-D and 3-D Pie and Column charts only)
GraphPercent	SmallInt	Displays the Y-value as a percent (2-D and 3-D Pie and Column charts only)
GraphShowY	SmallInt	Displays the Y-value in the units used in the table (2-D and 3-D Pie and Column charts only)

GraphLabelLocation constants

Constant	Data type	Description
LabelAbove	SmallInt	
LabelBelow	SmallInt	

LabelBottom	SmallInt
LabelCenter	SmallInt
LabelLeft	SmallInt
LabelMiddle	SmallInt
LabelRight	SmallInt
LabelTop	SmallInt

GraphLegendPosition constants

Constant	Data type	Description
LegendCenter	SmallInt	Displays the legend centered below the chart
LegendLeft	SmallInt	Displays the legend to the left of the chart

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

GraphMarkerSize constants

Constant	Data type	Description
MarkerSize0	SmallInt	
MarkerSize10	SmallInt	
MarkerSize14	SmallInt	
MarkerSize18	SmallInt	
MarkerSize2	SmallInt	
MarkerSize24	SmallInt	
MarkerSize3	SmallInt	
MarkerSize36	SmallInt	
MarkerSize6	SmallInt	
MarkerSize8	SmallInt	

GraphTypeOverride

Constant	Data type	Description
GraphArea	SmallInt	Displays specified series as an area chart
GraphBar	SmallInt	Displays specified series as a bar chart
GraphDefault	SmallInt	Displays specified series in the default chart type
GraphLine	SmallInt	Displays specified series as a line chart

GraphTypes

Constant	Data type	Description
Graph2DArea	SmallInt	2-dimensional area chart
Graph2DBar	SmallInt	2-dimensional bar chart
Graph2DColumns	SmallInt	2-dimensional column chart
Graph2DLine	SmallInt	2-dimensional line chart
Graph2DPie	SmallInt	2-dimensional pie chart
Graph2DRotatedBar	SmallInt	2-dimensional rotated bar chart
Graph2DStackedBar	SmallInt	2-dimensional stacked bar chart
Graph3DArea	SmallInt	3-dimensional area chart

Graph3DBar	SmallInt	3-dimensional bar chart
Graph3DColumns	SmallInt	3-dimensional column chart
Graph3DPie	SmallInt	3-dimensional pie chart
Graph3DRibbon	SmallInt	3-dimensional ribbon chart
Graph3DRotatedBar	SmallInt	3-dimensional rotated bar chart
Graph3DStackedBar	SmallInt	3-dimensional stacked bar chart
Graph3DStep	SmallInt	3-dimensional step chart
Graph3DSurface	SmallInt	3-dimensional surface chart
GraphXY	SmallInt	XY chart

IdRanges

Constant	Data type	Description
UserAction	SmallInt	Minimum value for a user-defined action constant
UserActionMax	SmallInt	Maximum value for a user-defined action constant
UserError	SmallInt	Minimum value for a user-defined error constant
UserErrorMax	SmallInt	Maximum value for a user-defined error constant
UserMenu	SmallInt	Minimum value for a user-defined menu ID constant
UserMenuMax	SmallInt constant	Maximum value for a user-defined menu ID

Keyboard constants

Constant	Data type	Description
VK_ADD	SmallInt	Add key
VK_APPS	SmallInt	Application property inspection 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	Down Arrow 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
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	Left Arrow 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	Right Arrow key
VK_SELECT	SmallInt	Select key
VK_SEPARATOR	SmallInt	Separator key
VK_SHIFT	SmallInt	SHIFT key
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	Up Arrow key

KeyboardState constants

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

Note

- If you want to combine the states you can add them together. For example, sending `mouseDown(15, 15, Shift+Control)` will emulate the Shift and Control keys being pressed during a `mouseDown` event.

LibraryScope constants

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

LineEnd constants

Constant	Data type	Description
ArrowBothEnds	SmallInt	Adds arrows to both ends of a line (only if <code>LineType = StraightLine</code>)
ArrowOneEnd	SmallInt	Adds an arrow to the terminal end of a line (only if <code>LineType = StraightLine</code>)
NoArrowEnd	SmallInt	Displays a line without arrows

LineStyle constants

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

LineThickness constants

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
LWidthHalfPoint	SmallInt point	Specifies a thickness of one half of a printer's point

LineType constants

Constant	Data type	Description
CurvedLine	SmallInt	Specifies a curved (elliptical) line
StraightLine	SmallInt	Specifies a straight line

MailAddressTypes constants

Constant	Data type	Description
MailAddrTo	SmallInt	Specifies this address goes on the To line
MailAddrCC	SmallInt	Specifies this address goes on the CC line
MailAddrBC	SmallInt	Specifies this person gets a copy of the message, without letting anyone else see it (may not be supported by all mail systems)

MailReadOptions constants

Constant	Data type	Description
MailReadBodyAsFile	SmallInt	Write the message text to a temporary file and add it as the first attachment in the attachment list
MailReadEnvelopeOnly	SmallInt	Read the message header only. Do not copy file attachments to temporary files nor read message text. (Setting MailReadEnvelopeOnly enhances performance.)
MailReadPeek	SmallInt	Do not mark the message as read

Marking a message as read affects its appearance in the user interface and generates a read receipt. If the mail system you are using does not support this, then MailReadPeek is ignored, and the message will be marked as read.

MailReadSuppressAttachments SmallInt

Read mail message header and text, but do not copy file attachments. This option is ignored if using MailReadEnvelopeOnly. Specifying to MailReadSuppressAttachments enhances performance.

MenuChoiceAttributes constants

Constant	Data type	Description
MenuChecked	SmallInt	Inserts a check mark before the menu item
MenuDisabled	SmallInt	Specifies that a menu item cannot be selected. Menu stays open
MenuEnabled	SmallInt	Specifies that a menu item can be selected. Menu closes
MenuGrayed	SmallInt	Displays a menu item in gray characters (dimmed)
MenuHilited	SmallInt	Highlights a menu item
MenuNotChecked	SmallInt	Displays a menu item without a check mark
MenuNotGrayed	SmallInt	Displays a menu item normally (not dimmed)
MenuNotHilited	SmallInt	Displays a menu item without a highlight

MenuCommands constants

Constant	Data type	Description
MenuAddPage	SmallInt	Adds a page to a form
MenuAddWatch	SmallInt	Program, Add Watch
MenuAlignBar	SmallInt	Toggles the Align option from View, Toolbars
MenuAlignBottom	SmallInt	Format, Alignment, Align Bottom
MenuAlignCenter	SmallInt	Format, Alignment, Align Center
MenuAlignMiddle	SmallInt	Format, Alignment, Align Middle
MenuAlignLeft	SmallInt	Format, Alignment, Align Left
MenuAlignRight	SmallInt	Format, Alignment, Align Right
MenuAlignTop	SmallInt	Format, Alignment, Align Top
MenuBuild (4.5)	SmallInt	Reports when the desktop is building a form's menu

MenuCanClose	SmallInt	Asks for permission to continue after choosing Close (Control menu)
MenuChangedPriv (5.0)	SmallInt	Reports when the private directory has been changed. Forms that remain open after the change can use this information to make adjustments, as needed.
MenuChangedWork (5.0)	SmallInt	Reports when the working directory has been changed. Forms that remain open after the change can use this information to make adjustments, as needed.
MenuChangingPriv (5.0)	SmallInt	Reports when the private directory is about to change. Setting the error code to a nonzero value allows a form to stay open after the change; setting the error code to zero closes the form before changing the directory.
MenuChangingWork (5.0)	SmallInt	Reports when the working directory is about to change. Setting the error code to a nonzero value allows a form to stay open after the change; setting the error code to zero closes the form before changing the directory.
MenuCheckSyntax	SmallInt	Program, Check Syntax
MenuCompile	SmallInt	Program, Compile
MenuCompileWithDebug	SmallInt	Program, Compile With Debug
MenuControlClose	SmallInt	Close (Control menu)
MenuControlKeyMenu	SmallInt	Control menu was invoked by a pressing a key
MenuControlMaximize	SmallInt	Maximize (Control menu)
MenuControlMinimize	SmallInt	Minimize (Control menu)
MenuControlMouseMenu	SmallInt	Control menu was invoked by a mouse click
MenuControlMove	SmallInt	Move (Control menu)
MenuControlNextWindow	SmallInt	Next Window (Control menu)
MenuControlPrevWindow	SmallInt	Prev Window (Control menu)
MenuControlRestore	SmallInt	Restore (Control menu)
MenuControlSize	SmallInt	Size (Control menu)
MenuCopyToolbar	SmallInt	Edit, Copy to Toolbar
MenuDataModel	SmallInt	Format, Data Model
MenuDataModelDesigner	SmallInt	Tools, Data Model Designer
MenuDataModelNew	SmallInt	File, New, Data Model
MenuDeliver	SmallInt	Format, Deliver

MenuDesignBringFront	SmallInt	Format, Order, Bring to Front
MenuDesignDuplicate	SmallInt	Edit, Duplicate
MenuDesignGroup	SmallInt	Format, Group
MenuDesignLayout	SmallInt	Format, Layout
MenuDesignSendBack	SmallInt	Format, Order, Send to Back
MenuDmCommit	SmallInt	Accepts the changes you have made
MenuDmLoad	SmallInt	File, Open
MenuDmRestore	SmallInt	Cancels the change you have made
MenuDmSave	SmallInt	File, Save
MenuDmUnlink	SmallInt	Removes existing links
MenuEditCopy	SmallInt	Edit, Copy
MenuEditCopyTo	SmallInt	Edit, Copy To
MenuEditCut	SmallInt	Edit, Cut
MenuEditDelete	SmallInt	Edit, Delete
MenuEditLinks	SmallInt	For OLE objects only
MenuEditPaste	SmallInt	Edit, Paste
MenuEditUndo	SmallInt	Edit, Undo
MenuExpertsOpen	SmallInt	Tools, Experts
MenuFieldFilter	SmallInt	Right-click Filter on Field object
MenuFieldPicture	SmallInt	Right-click Picture... on unbound fields
MenuFileAliases	SmallInt	Tools, Alias Manager...
MenuFileAutoRefresh	SmallInt	Tools, Settings, Preferences, Database, Refresh Rate
MenuFileExit	SmallInt	File, Exit
MenuFileExport	SmallInt	File, Export
MenuFileImport	SmallInt	File, Import
MenuFileMultiBlankZero	SmallInt	Tools, Settings, Preferences, Treat blank fields as zeros
MenuFileMultiUserDrivers	SmallInt	Tools, Settings, Preferences, BDE, Database driver list
MenuFileMultiUserInfo	SmallInt	Tools, Settings, Preferences, Database
MenuFileMultiUserLock	SmallInt	Tools, Security, Set Locks
MenuFileMultiUserLockInfo	SmallInt	Tools, Security, Display Locks

MenuFileMultiUserRetry	SmallInt	Tools, Settings, Preferences, Database, Set Retry
MenuFileMultiUserUserName	SmallInt	Tools, Settings, Preferences, Database, User name
MenuFileMultiUserWho	SmallInt	Tools, Settings, Preferences, Database, Current user list
MenuFilePrint	SmallInt	File, Print
MenuFilePrinterSetup	SmallInt	File, Printer Setup
MenuFilePrivateDir	SmallInt	Tools, Settings, Preferences, Database, Private directory
MenuFileTableAdd	SmallInt	Tools, Utilities, Add
MenuFileTableCopy	SmallInt	Tools, Utilities, Copy
MenuFileTableDelete	SmallInt	Tools, Utilities, Delete
MenuFileTableEmpty	SmallInt	Tools, Utilities, Empty
MenuFileTableInfoStructure	SmallInt	Tools, Utilities, Info Structure
MenuFileTablePasswords	SmallInt	Tools, Security, Passwords
MenuFileTableRename	SmallInt	Tools, Utilities, Rename
MenuFileTableRestructure	SmallInt	Tools, Utilities, Restructure
MenuFileTableSort	SmallInt	Tools, Utilities, Sort
MenuFileTableSubtract	SmallInt	Tools, Utilities, Subtract
MenuFileWorkingDir	SmallInt	File, Working Directory
MenuFolderOpen	SmallInt	Tools, Project Viewer
MenuFormatBar	SmallInt	Toggles the Text Formatting option from View, Toolbars
MenuFormDesign	SmallInt	View, Design Form
MenuFormEditData	SmallInt	View, Edit Data
MenuFormFieldView	SmallInt	View, Field View
MenuFormFilter	SmallInt	Format, Filter
MenuFormMemoView	SmallInt	View, Memo View
MenuFormNew	SmallInt	Opens a new form
MenuFormOpen	SmallInt	Opens a form
MenuFormOrderRange	SmallInt	Format, Filter
MenuFormPageFirst	SmallInt	View, Page, First
MenuFormPageGoto	SmallInt	View, Page, Go To
MenuFormPageLast	SmallInt	View, Page, Last

MenuFormPageNext	SmallInt	View, Page, Next
MenuFormPagePrevious	SmallInt	View, Page, Previous
MenuFormPersistView	SmallInt	View, Persistent Field View
MenuFormShowDeleted	SmallInt	View, Show Deleted
MenuFormTableView	SmallInt	View, Table View
MenuFormView	SmallInt	View, View Data
MenuFormViewData (5.0)	SmallInt	View, View Data
MenuHelpAbout	SmallInt	An Introduction to the available help
MenuHelpCoach (5.0)	SmallInt	Displays an expert coach to help
MenuHelpContents	SmallInt	Displays the help table of contents
MenuHelpKeyboard	SmallInt	Displays keyboard help
MenuHelpSearch	SmallInt	Displays the ObjectPAL references
MenuHelpSupport	SmallInt	Displays Technical Support info
MenuHelpUsingHelp	SmallInt	Displays an introduction to the help available
MenuHTMLPublish	SmallInt	File, Publish to HTML
MenuInit	SmallInt	Generated by clicking a menu items
MenuInsertObject	SmallInt	Edit, Insert Object (For OLE objects only)
MenuInsOleControl	SmallInt	Tools, Register
MenuLibraryNew	SmallInt	Opens a new library
MenuLibraryOpen	SmallInt	Opens a library
MenuNextWarning	SmallInt	Search, Next Warning
MenuNoteBookAddPage	SmallInt	Insert, Page
MenuNoteBookFirstPage	SmallInt	View, Page, First
MenuNoteBookLastPage	SmallInt	View, Page, Last
MenuNoteBookNextPage	SmallInt	View, Page, Next
MenuNoteBookPriorPage	SmallInt	View, Page, Previous
MenuNoteBookRotate	SmallInt	Format, Rotate Pages
MenuObjectBar	SmallInt	Toggles the Object option from View, Toolbars
MenuOpenProjectView (5.0)	SmallInt	Tools, Project Viewer
MenuPageLayout	SmallInt	Format, Layout

MenuPALSave	SmallInt	Edit, Save Debug State
MenuPasteFrom	SmallInt	Edit, Paste From
MenuPasteLink	SmallInt	Edit, Paste Link
MenuProjAdd	SmallInt	Add Reference button (Standard toolbar)
MenuProjDelete	SmallInt	Remove Reference button (Standard toolbar)
MenuPropertiesBandLabels	SmallInt	View, Band Labels
MenuPropertiesCurrent	SmallInt	Edit, Current Object
MenuPropertiesCurrentDialog	SmallInt	Edit, Current Object
MenuPropertiesDesigner	SmallInt	View, Design Form
MenuPropertiesDesktop	SmallInt	View, View Data
MenuPropertiesExpandedRuler	SmallInt	Expanded Ruler (Designer Preferences options)
MenuPropertiesFormRestoreDefaults	SmallInt	Restores the default settings for a form
MenuPropertiesFormSaveDefaults	SmallInt	Saves the default settings for a form
MenuPropertiesGroupRepeats	SmallInt	Format, Properties, Remove group repeat
MenuPropertiesHorizontalRuler	SmallInt	Horizontal Ruler (Designer Preferences options)
MenuPropertiesMethods	SmallInt	Object Explorer
MenuPropertiesShowGrid	SmallInt	View, Grid
MenuPropertiesSizeandPos	SmallInt	View, Size and Position
MenuPropertiesSizeToFit	SmallInt	Format, Properties
MenuPropertiesSnapToGrid	SmallInt	Format, Snap to Grid
MenuPropertiesStyleSheet	SmallInt	Format, Style Sheet
MenuPropertiesVerticalRuler	SmallInt	Vertical Ruler (Designer Preferences options)
MenuPropertiesWindow	SmallInt	Displays the Window Style
MenuPropertiesZoom100	SmallInt	View, Zoom, 100%
MenuPropertiesZoom200	SmallInt	View, Zoom, 200%
MenuPropertiesZoom25	SmallInt	View, Zoom, 25%
MenuPropertiesZoom400	SmallInt	View, Zoom, 400%
MenuPropertiesZoom50	SmallInt	View, Zoom, 50%
MenuPropertiesZoomBestFit	SmallInt	View, Zoom, Best Fit
MenuPropertiesZoomFitHeight	SmallInt	View, Zoom, Fit Height

MenuPropertiesZoomFitWidth	SmallInt	View, Zoom, Fit Width
MenuQueryNew	SmallInt	Opens a new query
MenuQueryOpen	SmallInt	Opens an existing query
MenuQBEDoJoin	SmallInt	Join Tables button (Query design window toolbar)
MenuQBEPProperties	SmallInt	Query, Properties
MenuQBEShowSQL	SmallInt	View, Show SQL
MenuQBESortAnswer	SmallInt	Query, Properties: Sort
MenuQuickForm	SmallInt	Tools, Quick Form
MenuQuickGraph	SmallInt	Tools, Quick Chart
MenuQuickReport	SmallInt	Tools, Quick Report
MenuQuickXTab	SmallInt	Tools, Quick Crosstab
MenuRecordCancel	SmallInt	Record, Cancel Changes
MenuRecordDelete	SmallInt	Record, Delete
MenuRecordFastBackward	SmallInt	Record, Previous Set
MenuRecordFastForward	SmallInt	Record, Next Set
MenuRecordFirst	SmallInt	Record, First
MenuRecordInsert	SmallInt	Record, Insert
MenuRecordLast	SmallInt	Record, Last
MenuRecordLocateNext	SmallInt	Record, Locate Next
MenuRecordLocateRecordNumber	SmallInt	Record, Locate, Record Number
MenuRecordLocateSearchAndReplace	SmallInt	Record, Locate, and Replace
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
MenuReportAddBand	SmallInt	Insert, Group Band
MenuReportNew	SmallInt	Opens a new report

MenuReportOpen	SmallInt	Opens a report
MenuReportPageFirst	SmallInt	View, Page, First
MenuReportPageGoto	SmallInt	View, Page, Go To
MenuReportPageLast	SmallInt	View, Page, Last
MenuReportPageNext	SmallInt	View, Page, Next
MenuReportPagePrevious	SmallInt	View, Page, Previous
MenuReportPrintDesign	SmallInt	Prints the file
MenuReportRestartOpts	SmallInt	Format, Restart Options
MenuRotatePage	SmallInt	Format, Rotate Pages
MenuSave	SmallInt	File, Save
MenuSaveAs	SmallInt	File, Save As...
MenuSaveCrossTab	SmallInt	Edit, Save Crosstab (must have a defined crosstab on a runtime form)
MenuScriptNew	SmallInt	Opens a new script
MenuScriptOpen	SmallInt	Opens a script
MenuSearchText	SmallInt	Edit, Search Text
MenuSelectAll	SmallInt	Edit, Select All
MenuSetBreakPoint	SmallInt	Program, Toggle Breakpoint
MenuSizeMaxHeight	SmallInt	Format, Size, Maximum Height
MenuSizeMaxWidth	SmallInt	Format, Size, Maximum Width
MenuSizeMinHeight	SmallInt	Format, Size, Minimum Height
MenuSizeMinWidth	SmallInt	Format, Size, Minimum Width
MenuSpaceHorz	SmallInt	Format, Spacing, Horizontal
MenuSpaceVert	SmallInt	Format, Spacing, Vertical
MenuSpellCheckForm	SmallInt	Tools, Spell Checker
MenuSpellCheckView	SmallInt	Tools, Spell Checker
MenuSQLFileNew	SmallInt	Opens a new SQL File
MenuSQLFileOpen	SmallInt	Opens an SQL File
MenuStackPages	SmallInt	Stacks the pages in a form
MenuStepInto	SmallInt	Program, Step Into
MenuStepOver	SmallInt	Program, Step Over

MenuTableNew	SmallInt	Opens a new table
MenuTableOpen	SmallInt	Opens a table
MenuTileHorizontal	SmallInt	Tiles a form's pages side-by-side
MenuTileVertical	SmallInt	Tiles a form's pages, top and bottom
MenuViewBreakPoints	SmallInt	View, Breakpoints
MenuViewDebugger	SmallInt	View, Debugger
MenuViewMethods	SmallInt	View, ObjectPAL Quick Lookup: Types and Methods
MenuViewSource	SmallInt	View, Source
MenuViewStack	SmallInt	View, Call Stack
MenuViewTracer	SmallInt	View, Tracer
MenuViewTypes	SmallInt	View, ObjectPAL Quick Lookup: Types and Methods
MenuViewWatch	SmallInt	View, Watch
MenuWindowArrangeIcons	SmallInt	Window, Arrange Icons
MenuWindowCascade	SmallInt	Window, Cascade
MenuWindowCloseAll	SmallInt	Window, Close All
MenuWindowTile	SmallInt	Window, Tile
MenuWriteAsText	SmallInt	File, Publish As, Text
MenuWriteAsRTF	SmallInt	File, Publish As, RTF

MenuReasons constants

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 Toolbar button

MouseShapes constants

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)

MouseSize	LongInt	Pointer is four-headed arrow pointing North-South-East-West
MouseSizeNWSE	LongInt	Pointer is two-headed arrow pointing Northwest-Southeast
MouseSizeNESW	LongInt	Pointer is two-headed arrow pointing Northeast-Southwest
MouseSizeWE	LongInt	Pointer is two-headed arrow pointing East-West
MouseSizeNS	LongInt	Pointer is two-headed arrow pointing North-South
MouseNo	LongInt	Pointer is the international symbol for NO
MouseHand	LongInt	Pointer is a hand
MouseHelp	LongInt	Pointer is the standard arrow and a question mark
MouseDrag	LongInt	Pointer is the standard document drag and drop
MouseUpArrow	LongInt	Pointer is an arrow pointing up
MouseWait	LongInt	Pointer is an hourglass

MoveReasons constants

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

PageTilingOptions constants

Constant	Data type	Description
StackPages	SmallInt	Pages are stacked one on top of the other
TileHorizontal	SmallInt	Pages are tiled horizontally
TileVertical	SmallInt	Pages are tiled vertically

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
RightDiagonalLinesPattern	SmallInt
ScalesPattern	SmallInt
StaggeredDashesPattern	SmallInt
ThickHorizontalLinesPattern	SmallInt
ThickStripesDownPattern	SmallInt
ThickStripesUpPattern	SmallInt
ThickVerticalLinesPattern	SmallInt
VerticalLinesPattern	SmallInt
VeryHeavyDotPattern	SmallInt
WeavePattern	SmallInt
ZigZagPattern	SmallInt

PrintColor constants

Constant	Data type	Description
prnColor	LongInt	Print in color (color printers only)

prnMonochrome	LongInt	Print in monochrome
---------------	---------	---------------------

PrintDuplex constants

Constant	Data type	Description
prnHorizontal	LongInt	Double-sided printing where the left and right edges of consecutive pages can be bound (also called bind on edge printing)
prnSimplex	LongInt	Single-sided printing
prnVertical	LongInt	Double-sided printing where the top and bottom edges of consecutive pages can be bound (also called bind on top printing)

PrinterOrientation constants

Constant	Data type	Description
prnLandscape	LongInt	Landscape (long) orientation
prnPortrait	LongInt	Portrait (tall) orientation

PrinterSizes constants

Constant	Data type	Description
prn10x14	LongInt	10 by 14 inches
prn11x17	LongInt	11 by 17 inches
prnA3	LongInt	A3 297 x 420 mm
prnA4	LongInt	A4 210 x 297 mm
prnA4Small	LongInt	A4 Small 210 x 297 mm
prnA5	LongInt	A5 148 x 210 mm
prnB4	LongInt	B4 250 x 354
prnB5	LongInt	B5 182 x 257 mm
prnCSheet	LongInt	C size sheet
prnDSheet	LongInt	D size sheet
prnEnv9	LongInt	Envelope #9 3 7/8 x 8 7/8 inches
prnEnv10	LongInt	Envelope #10 4 1/8 x 9 1/2 inches
prnEnv11	LongInt	Envelope #11 4 1/2 x 10 3/8 inches

prnEnvI2	LongInt	Envelope #12 4 3/4 x 11 inches
prnEnvI4	LongInt	Envelope #14 5 x 11 1/2 inches
prnEnvB4	LongInt	Envelope B4 250 x 353 mm
prnEnvB5	LongInt	Envelope B5 176 x 250 mm
prnEnvB6	LongInt	Envelope B6 176 x 125 mm
prnEnvC3	LongInt	Envelope C3 324 x 458 mm
prnEnvC4	LongInt	Envelope C4 229 x 324 mm
prnEnvC5	LongInt	Envelope C5 162 x 229 mm
prnEnvC6	LongInt	Envelope C6 114 x 162 mm
prnEnvC65	LongInt	Envelope C65 114 x 229 mm
prnEnvDL	LongInt	Envelope DL 110 x 220mm
prnEnvItaly	LongInt	Envelope 110 x 230 mm
prnEnvMonarch	LongInt	Envelope Monarch 3.875 x 7.5 inches
prnEnvPersonal	LongInt	6 3/4 Envelope 3 5/8 x 6 1/2 inches
prnESheet	LongInt	E size sheet
prnExecutive	LongInt	Executive 7 1/4 x 10 1/2 inches
prnFanfoldLegalGerman	LongInt	German Legal Fanfold 8 1/2 x 13 inches
prnFanfoldStandardGerman	LongInt	German Std Fanfold 8 1/2 x 12 inches
prnFanfoldUS	LongInt	US Std Fanfold 14 7/8 x 11 inches
prnFolio	LongInt	Folio 8 1/2 x 13 inches
prnLedger	LongInt	Ledger 17 x 11 inches
prnLegal	LongInt	Legal 8 1/2 x 14 inches
prnLetter	LongInt	Letter 8 1/2 x 11 inches
prnLetterSmall	LongInt	Letter Small 8 1/2 x 11 inches
prnNote	LongInt	Note 8 1/2 x 11 inches
prnQuarto	LongInt	Quarto 215 x 275 mm
prnStatement	LongInt	Statement 5 1/2 x 8 1/2 inches
prnTabloid	LongInt	Tabloid 11 x 17 inches

PrinterType constants

Constant	Data type	Description
prnHppCL	SmallInt	
prnPostscript	SmallInt	
prnTTY	SmallInt	
prnUnknown	SmallInt	

PrintQuality constants

Constant	Data type	Description
prnDraft	LongInt	Draft quality (lowest quality, fastest print time)
prnHigh	LongInt	High quality (highest quality, slowest print time)
prnLow	LongInt	Low quality
prnMedium	LongInt	Medium quality

PrintSources constants

Constant	Data type	Description
prnAuto	LongInt	Paper source selected automatically
prnCassette	LongInt	Cassette
prnEnvelope	LongInt	Envelope, automatic feed
prnEnvManual	LongInt	Envelope, manual feed
prnLargeCapacity	LongInt	Large capacity paper source
prnLargeFmt	LongInt	Large format paper source
prnLower	LongInt	Lower paper tray
prnManual	LongInt	Manual feed
prnMiddle	LongInt	Middle paper tray
prnOnlyOne	LongInt	Single paper tray
prnSmallFmt	LongInt	Small format paper source
prnTractor	LongInt	Tractor feed paper
prnUpper	LongInt	Upper paper tray

PublishToFilters constants

Constant	Data type	Description
PublishToRTF	SmallInt	publishes the current report object to Rich Text Format.
PublishToWP9	SmallInt	publishes the current report to WordPerfect9 format.
PublishToWord97	SmallInt	publishes the current report to Microsoft Word97 format.

qbeCheckType constants

Constant	Data type
checkCheck	SmallInt
checkDesc	SmallInt
checkGroup	SmallInt
checkNone	SmallInt
checkPlus	SmallInt

qbeRowOperation constants

Constant	Data type
qbeRowDelete	SmallInt
qbeRowInsert	SmallInt
qbeRowNone	SmallInt
qbeRowSet	SmallInt

QueryRestartOptions constants

Constant	Data type	Description
QueryDefault	SmallInt	Use the options specified interactively using the Query Restart Options dialog box
QueryLock	SmallInt	Lock all other users out of the tables needed while the query is running. If Paradox cannot lock a table, it does not run the query. This is the least polite to other users. You must wait until all the locks can be secured before the query will run.
QueryNoLock	SmallInt	Run the query even if someone changes the data while it's running.

QueryRestart	SmallInt	Start the query over. Specify QueryRestart when you want to make sure you get a snapshot of the data as it existed at some instant. Another user might change the data after the query is completed but before the Answer table is displayed, but at least you got a snapshot. This is just the nature of multi-user work.
--------------	----------	--

RasterOperations constants

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 unchanged 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

RegistryKeyType constants

Constant	Data type	Description
regKeyClassesRoot	LongInt	Alias to HKEY_CLASSES_ROOT in the Registry
regKeyCurrentUser	LongInt	Alias to HKEY_CURRENT_USER in the Registry
regKeyLocalMachine	LongInt	Alias to HKEY_LOCAL_MACHINE in the Registry
regKeyUser	LongInt	Alias to HKEY_USERS in the Registry

ReportOrientation constants

Constant	Data type	Description
PrintDefault	SmallInt	Use the current Windows default orientation

PrintLandscape	SmallInt	Use landscape (long) orientation
PrintPortrait	SmallInt	Use portrait (tall) orientation

ReportPrintPanel constants

Constant	Data type	Description
PrintClipToWidth	SmallInt	Clips (trims) all data that does not fit across the page (within the margins)
PrintHorizontalPanel	SmallInt	Prints additional pages as needed to fit all the data. Each of these pages immediately follows the page it extends.
PrintOverflowPages	SmallInt	Same as PrintHorizontalPanel
PrintVerticalPanel	SmallInt	Creates a secondary page for each page of the report, even if it doesn't overflow

ReportPrintRestart constants

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

RestructureOperations constants

Constant	Data type	Description
RestructureModify	SmallInt	Modify an existing field
RestructureAdd	SmallInt	Add a new field
RestructureDrop	SmallInt	Drop (delete) an existing field

SpecialFieldTypes constants

Constant	Data type	Description
DateField	SmallInt	Displays the current system date

NofFieldsField	SmallInt	Displays the number of fields in the current table
NofPagesField	SmallInt	Displays the number of pages in the current form or report
NofRecsField	SmallInt	Displays the number of records in the current table
PageNumField	SmallInt	Displays the current page number
RecordNoField	SmallInt	Displays the active record number
TableNameField	SmallInt	Displays the name of the current table
TimeField	SmallInt	Displays the current system time

StatusReasons constants

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

TableFrameStyles constants

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
tfSingleLine	SmallInt	Table frame has a box frame
tfTripleLine	SmallInt	Table frame has a triple-box frame

TextAlignment constants

Constant	Data type	Description
TextAlignBottom	SmallInt	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)

TextDesignSizing constants

Constant	Data type	Description
TextFixedSize	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

TextSpacing constants

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

ToolbarBitmap constants

Constant	Data type	Description
BitmapAddBand	SmallInt	System bitmap
BitmapAddTable	SmallInt	System bitmap
BitmapAddToCat	SmallInt	System bitmap
BitmapAlignBottom	SmallInt	System bitmap
BitmapAlignCenter	SmallInt	System bitmap
BitmapAlignLeft	SmallInt	System bitmap
BitmapAlignMiddle	SmallInt	System bitmap
BitmapAlignRight	SmallInt	System bitmap
BitmapAlignTop	SmallInt	System bitmap
BitmapBookTool	SmallInt	System bitmap
BitmapBoxTool	SmallInt	System bitmap
BitmapBringToFront	SmallInt	System bitmap

BitmapButtonTool	SmallInt	System bitmap
BitmapCancel	SmallInt	System bitmap
BitmapChartTool	SmallInt	System bitmap
BitmapChkSyntax	SmallInt	System bitmap
BitmapCoEdit	SmallInt	System bitmap
BitmapCompile	SmallInt	System bitmap
BitmapDataBegin	SmallInt	System bitmap
BitmapDataEnd	SmallInt	System bitmap
BitmapDataModel	SmallInt	System bitmap
BitmapDataNextRecord	SmallInt	System bitmap
BitmapDataNextSet	SmallInt	System bitmap
BitmapDataPriorRecord	SmallInt	System bitmap
BitmapDataPriorSet	SmallInt	System bitmap
BitmapDelTable	SmallInt	System bitmap
BitmapDesignMode	SmallInt	System bitmap
BitmapDoJoin	SmallInt	System bitmap
BitmapDuplicate	SmallInt	System bitmap
BitmapEditAnswer	SmallInt	System bitmap
BitmapEditCopy	SmallInt	System bitmap
BitmapEditCut	SmallInt	System bitmap
BitmapEditPaste	SmallInt	System bitmap
BitmapEllipseTool	SmallInt	System bitmap
BitmapFieldTool	SmallInt	System bitmap
BitmapFilter	SmallInt	System bitmap
BitmapFirstPage	SmallInt	System bitmap
BitmapFldView	SmallInt	System bitmap
BitmapFontAttribBold	SmallInt	System bitmap
BitmapFontAttribItalic	SmallInt	System bitmap
BitmapFontAttribStrikeout	SmallInt	System bitmap
BitmapFontAttribUnderline	SmallInt	System bitmap

BitmapGotoPage	SmallInt	System bitmap
BitmapGraphicTool	SmallInt	System bitmap
BitmapGroup	SmallInt	System bitmap
BitmapHelp	SmallInt	System bitmap
BitmapHSpacing	SmallInt	System bitmap
BitmapLastPage	SmallInt	System bitmap
BitmapLineSpace1	SmallInt	System bitmap
BitmapLineSpace15	SmallInt	System bitmap
BitmapLineSpace2	SmallInt	System bitmap
BitmapLineSpace25	SmallInt	System bitmap
BitmapLineSpace3	SmallInt	System bitmap
BitmapLineSpace35	SmallInt	System bitmap
BitmapLineTool	SmallInt	System bitmap
BitmapLinkDm	SmallInt	System bitmap
BitmapLoadDm	SmallInt	System bitmap
BitmapMaxHeight	SmallInt	System bitmap
BitmapMaxWidth	SmallInt	System bitmap
BitmapMinHeight	SmallInt	System bitmap
BitmapMinWidth	SmallInt	System bitmap
BitmapNextPage	SmallInt	System bitmap
BitmapNextWarn	SmallInt	System bitmap
BitmapObjectTree	SmallInt	System bitmap
BitmapOk	SmallInt	System bitmap
BitmapOleTool	SmallInt	System bitmap
BitmapOpenExpert	SmallInt	System bitmap
BitmapOpenForm	SmallInt	System bitmap
BitmapOpenLibrary	SmallInt	System bitmap
BitmapOpenProject	SmallInt	System bitmap
BitmapOpenQbe	SmallInt	System bitmap
BitmapOpenReport	SmallInt	System bitmap

BitmapOpenScript	SmallInt	System bitmap
BitmapOpenSql	SmallInt	System bitmap
BitmapOpenTable	SmallInt	System bitmap
BitmapOpenTutor	SmallInt	System bitmap
BitmapPageBreak	SmallInt	System bitmap
BitmapPickTool	SmallInt	System bitmap
BitmapPrevPage	SmallInt	System bitmap
BitmapPrint	SmallInt	System bitmap
BitmapQuickForm	SmallInt	System bitmap
BitmapQuickGraph	SmallInt	System bitmap
BitmapQuickReport	SmallInt	System bitmap
BitmapQuickXTab	SmallInt	System bitmap
BitmapRecordTool	SmallInt	System bitmap
BitmapRemoveFromCat	SmallInt	System bitmap
BitmapRestructure	SmallInt	System bitmap
BitmapRun	SmallInt	System bitmap
BitmapSave	SmallInt	System bitmap
BitmapSaveDm	SmallInt	System bitmap
BitmapSendToBack	SmallInt	System bitmap
BitmapSetBreak	SmallInt	System bitmap
BitmapSetOrgin	SmallInt	System bitmap
BitmapSetWatch	SmallInt	System bitmap
BitmapShowSQL	SmallInt	System bitmap
BitmapSortAnswer	SmallInt	System bitmap
BitmapSpeedExit	SmallInt	System bitmap
BitmapSrchNext	SmallInt	System bitmap
BitmapSrchValue	SmallInt	System bitmap
BitmapStepInto	SmallInt	System bitmap
BitmapStepOver	SmallInt	System bitmap
BitmapStop	SmallInt	System bitmap

BitmapTableFrameTool	SmallInt	System bitmap
BitmapTButton	SmallInt	System bitmap
BitmapTComboBox	SmallInt	System bitmap
BitmapTextCenter	SmallInt	System bitmap
BitmapTextJustify	SmallInt	System bitmap
BitmapTextLeft	SmallInt	System bitmap
BitmapTextRight	SmallInt	System bitmap
BitmapTextTool	SmallInt	System bitmap
BitmapTGuage	SmallInt	System bitmap
BitmapTHeader	SmallInt	System bitmap
BitmapTListBox	SmallInt	System bitmap
BitmapTSpinEdit	SmallInt	System bitmap
BitmapViewBreak	SmallInt	System bitmap
BitmapViewCallStack	SmallInt	System bitmap
BitmapViewDebugger	SmallInt	System bitmap
BitmapViewMethods	SmallInt	System bitmap
BitmapViewSource	SmallInt	System bitmap
BitmapViewTracer	SmallInt	System bitmap
BitmapViewTypes	SmallInt	System bitmap
BitmapViewWatch	SmallInt	System bitmap
BitmapVSpacing	SmallInt	System bitmap
BitmapXtabTool	SmallInt	System bitmap

ToolBarButtonType constants

Constant	Data type	Description
ToolBarButtonPush	SmallInt	Specifies a pushbutton type toolbar button
ToolBarButtonRadio	SmallInt	Specifies a radiobutton type toolbar button
ToolBarButtonRepeat	SmallInt	Specifies a repeating pushbutton type toolbar button
ToolBarButtonToggle	SmallInt	Specifies a toggle-action toolbar button

ToolbarClusterID constants

A cluster is a logical aggregation of buttons. There are 13 clusters in the system. Each cluster is always at the same position. The position of the cluster is expressed in 'Button widths' on a horizontal Toolbar. For example, the Mode cluster starts at a distance of 4 button widths from the left.

Constant	Data type	Description
ToolbarFileCluster	SmallInt	Specifies Toolbar cluster 0 (position 0)
ToolbarEditCluster	SmallInt	Specifies Toolbar cluster 1 (position 1, 2, 3)
ToolbarModeCluster	SmallInt	Specifies Toolbar cluster 2 (position 4, 5)
ToolbarToolCluster	SmallInt	Specifies Toolbar cluster 3 (position 6, 7)
ToolbarVCRCluster	SmallInt	Specifies Toolbar cluster 4 (position 8, 9)
ToolbarInterCluster	SmallInt	Specifies Toolbar cluster 5 (position 10, 11, 12, 13)
ToolbarInter2Cluster	SmallInt	Specifies Toolbar cluster 6 (position 14)
ToolbarQuickCluster	SmallInt	Specifies Toolbar cluster 7 (position 15, 16, 17)
ToolbarMiscCluster	SmallInt	Specifies Toolbar cluster 8 (position 18, 19)
ToolbarMisc2Cluster	SmallInt	Specifies Toolbar cluster 9 (position 20, 21)
ToolbarObjectCluster	SmallInt	Specifies Toolbar cluster 10 (position 22)
ToolbarProjectCluster	SmallInt	Specifies Toolbar cluster 11 (position 23)
ToolbarExpertCluster	SmallInt	Specifies Toolbar cluster 12 (position 24)

Note

- ClusterID constants must be set, but they are ignored by the system.

ToolbarState constants

Constant	Data type	Description
ToolbarStateBottom	SmallInt	Specifies a Toolbar docked at screen bottom
ToolbarStateFloatHorizontal	SmallInt	Specifies a floating horizontal Toolbar
ToolbarStateFloatVertical	SmallInt	Specifies a floating vertical Toolbar
ToolbarStateLeft	SmallInt	Specifies a Toolbar docked at screen left
ToolbarStateRight	SmallInt	Specifies a Toolbar docked at screen right
ToolbarStateTop	SmallInt	Specifies a Toolbar docked at screen top

TrackBarStyles constants

Constant	Data type	Description
LineSize	SmallInt	The number of ticks the thumb moves on LineUp and LineDown events
PageSize	SmallInt	The number of ticks the thumb moves on PageUp and PageDown events
TrackBarAutoTic	Logical	Automatically displays tick marks
TrackBarBoth	Logical	Displays ticks on both sides of the trackbar
TrackBarBottom	Logical	Thumb points down, tick marks at bottom (TrackBarHorz only)
TrackBarEnableSelRange	Logical	Enables a selected range within the trackbar, used with a SelStart and SelEnd value, highlights the selection range
TrackBarHorz	Logical	Displays trackbar horizontally
TrackBarLeft	Logical	Thumb, tick marks on left-hand side of trackbar (TrackBarVert only)
TrackBarNoTicks	Logical	Do not display tick marks
TrackBarRight	Logical	Thumb, tickmarks on right-hand side of trackbar (TrackBarVert only)
TrackBarTop	Logical	Thumb points up, tick marks at top (TrackBarHorz only)
TrackBarVert	Logical	Displays trackbar vertically

UIObjectTypes constants

Constant	Data type	Description
BandTool (5.0)	SmallInt	Creates a report band
BoxTool	SmallInt	Creates a box
ButtonTool	SmallInt	Creates a button
ChartTool	SmallInt	Creates a chart
EllipseTool	SmallInt	Creates an ellipse
FieldTool	SmallInt	Creates a field
GraphicTool	SmallInt	Creates a graphic object
LineTool	SmallInt	Creates a line
OleTool	SmallInt	Creates an OLE object

NoteBookTool (7)	SmallInt	Creates a tabbed notebook object
PageBrkTool (5.0)	SmallInt	Creates a page break in a report
PageTool	SmallInt	Creates a new page
RecordTool	SmallInt	Creates a multi-record object
SelectionTool	SmallInt	Allows you to select an object
TableFrameTool	SmallInt	Creates a table frame
TextTool	SmallInt	Creates a text box
XtabTool	SmallInt	Creates a crosstab object

UpdateLink constants

Constants	Data type	Description
OLEUpdateAutomatic	SmallInt	
OLEUpdateManual	SmallInt	

ValueReasons constants

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 (e.g., by choosing a radio button or list item), but the field value has not been committed (e.g., 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.

WindowStyles constants

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 maximize 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

B

Properties and property values

This appendix lists the properties and property values available in ObjectPAL. For more information on each property, including a description of each, see the ObjectPAL online Help file. As well, refer to the ObjectPAL online Help file for a complete listing of all the properties unique to chart objects.

Property	Data type	Values
Alignment property	SmallInt	TextAlignCenter, TextAlignJustify, TextAlignLeft, TextAlignRight
Arrived property	Logical	True, False
AttachedHeader property	Logical	True, False
AutoAppend property	Logical	True, False
AvgCharSize property	Point	>0
BlankRecord property	Logical	True, False
Border property	Logical	True, False
BottomBorder property	LongInt	N/A
Breakable property	Logical	True, False
ButtonType property	SmallInt	CheckBoxType, PushButtonType, RadioButtonType
ByRows property	Logical	True, False
CalculatedField property	Logical	True, False
Caption property	Logical	True, False
CenterLabel property	Logical	True, False
CheckedValue property	String	N/A
Class property	String	Band, Bitmap, Box, Button, Cell, Chart, Crosstab, EditRegion, Ellipse, Field, Form, FormData, Group, Header, Line, List, Multi-record, OLE, Page, Record, Report, ReportPrint, TableFrame, Text

Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Columnar property	Logical	True, False
ColumnPosition property	SmallInt	> 0
ColumnWidth property	LongInt	> 0
CompleteDisplay property	Logical	True, False
ContainerName property	String	N/A
ControlMenu property	Logical	True, False
CurrentColumn property	SmallInt	> 0
CurrentPage property	String	User defined
CurrentRecordMarker.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
CurrentRecordMarker.LineStyle property	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
CurrentRecordMarker.Show property	Logical	True, False
CurrentRow	SmallInt	> 0
CursorColumn property	LongInt	N/A
CursorLine property	LongInt	N/A
CursorPos property	LongInt	N/A
DataSource property	String	user dependent.
Default property	String	N/A
DefineGroup property	Logical	True, False
DeleteColumn property	SmallInt	> 0
Deleted property	Logical	True, False
DeleteWhenEmpty property	Logical	True, False
Design.ContainObjects property	Logical	True, False
Design.PinHorizontal property	Logical	True, False
Design.PinVertical property	Logical	True, False
Design.Selectable property	Logical	True, False
Design.SizeToFit property	Logical	True, False

DesignModified property	Logical	True, False
DesignSizing property	SmallInt	TextFixedSize, TextGrowOnly, TextSizeToFit
DesktopForm property	Logical	True, False
DialogForm property	Logical	True, False
DialogFrame property	Logical	True, False
DisplayType property	SmallInt	CheckBoxField, ComboField, EditField, LabeledField, ListField, RadioButtonField
DrillDown property	Integer	0-creates a single HTML template file for the entire report 1-creates separate HTML template files for each group in the drilldown 2-creates separate HTML template files for each nested group in the drilldown 3-creates separate HTML template files for each item in the drilldown
Editing property	Logical	True, False
Enabled property	Logical	True, False
End property	Point	N/A
FieldName property	String	N/A
FieldNo property	SmallInt	N/A
FieldRights property	String	ReadOnly, ReadWrite, All
FieldSize property	SmallInt	N/A
FieldSqueeze property	Logical	True/False
FieldType property	String	N/A
FieldUnits2 property	SmallInt	N/A
FieldValid property	Logical	True, False
FieldView property	Logical	True, False
First property	String	N/A
FirstRow property	Data type	N/A
FitHeight property	Logical	True, False
FitWidth property	Logical	True, False
FlatLook property	Logical	True, False
FlyAway property	Logical	True, False
Focus property	Logical	True, False

Font.color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Font.Script	String	Varies according to individual system fonts.
Font.Size	SmallInt	> 0
Font.Style property	SmallInt	FontAttribBold, FontAttribItalic, FontAttribNormal, FontAttribStrikeout, FontAttribUnderline
Font.Typeface property	String	Varies according to individual system attributes.
Format.DateFormat property	String	Format specification
Format.LogicalFormat property	String	Format specification
Format.NumberFormat property	String	Format specification
Format.TimeFormat property	N/A	Format specification
Format.TimeStampFormat property	String	Format specification
Frame.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Frame.Style property	SmallInt	Windows3DFrame, Windows3DGroup, DashDotDotFrame, DashDotFrame, DashedFrame, DottedFrame, DoubleFrame, Inside3DFrame, NoFrame, Outside3DFrame, ShadowFrame, SolidFrame, WideInsideDoubleFrame, WideOutsideDoubleFrame
Frame.Thickness property	SmallInt	N/A
FrameObjects property	Logical	True, False
FullName property	String	N/A
FullSize property	Point	N/A
Grid.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Grid.GridStyle property	SmallInt	tf3D, tfDoubleLine, tfNoGrid, tfSingleLine, tfTripleLine
Grid.RecordDivider property	Logical	True, False
GridLines.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
GridLines.ColumnLines property	Logical	True, False
GridLines.HeadingLines property	Logical	True, False

GridLines.LineStyle	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
GridLines.QueryLook property	Logical	True, False
GridLines.RowLines	Logical	True, False
GridLines.Spacing	SmallInt	TextSingleSpacing, TextDoubleSpacing, TextTripleSpacing
GridValue	Point	> 0
GroupObjects	Logical	True, False
GroupRecords	SmallInt	> 0
Header property	String	N/A
HeadingHeight property	LongInt	> 0
Headings property	String	GroupOnly, PageAndGroup
HTMLAction property	String	N/A
HTMLFormParams property	Logical	True, False
HTMLMethod property	String	POST, GET
dHorizontalScrollBar property	Logical	True, False
InactiveColor property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, any user-defined color.
IncludeAllData property	Logical	True, False
IndexField property	Logical	True, False
InsertColumn property	SmallInt	0
InsertField property	Point	N/A
Inserting property	Logical	True, False
Invisible property	Logical	True, False
Justification property	SmallInt	TextAlignTop, TextAlignBottom, TextAlignVCenter, TextAlignLeft, TextAlignRight, TextAlignCenter
KeyField property	Logical	True, False
LabelText property	String	N/A
LeftBorder property	LongInt	N/A
Line.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent

Line.LineStyle property	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
Line.Thickness property	SmallInt	N/A
LineEnds property	SmallInt	ArrowBothEnds, ArrowOneEnd, NoArrowEnd
LineSpacing property	SmallInt	TextDoubleSpacing, TextDoubleSpacing2, TextSingleSpacing, TextSingleSpacing2, TextTripleSpacing
LineSqueeze property	Logical	True/False
LineStyle property	SmallInt	DashDotDotLine, DashDotLine, DashedLine, DottedLine, NoLine, SolidLine
LineType property	SmallInt	CurvedLine, StraightLine
List.Count property	SmallInt	N/A
List.Selection property	SmallInt	N/A
List.Value property	AnyType	N/A
Locked property	Logical	True, False
LookupTable property	String	N/A
LookupType property	String	JustCurrentField, AllCorresponding
Magnification property	SmallInt	Magnify25, Magnify50, Magnify100, Magnify200, Magnify400, MagnifyBestFit
Manager property	String	N/A
MarkerPos property	LongInt	N/A
Margins.Bottom property	LongInt	0
Margins.Left property	LongInt	0
Margins.Right property	LongInt	0
Margins.Top property	LongInt	0
MaximizeButton property	Logical	True, False
Maximum property	String	N/A
MemoView property	Logical	True, False
MinimizeButton property	Logical	True, False
Minimum property	String	N/A
Modal property	Logical	True, False
MouseActivate property	Logical	True, False
NCols property	SmallInt	N/A

NRecords property	LongInt	N/A
NRows property	SmallInt	N/A
Name property	String	N/A
Next property	String	N/A
NextTabStop property	String	N/A
NoEcho property	Logical	True, False
NumberPages property	SmallInt	User defined
Orphan/Widow property	Logical	True/False
OtherBandName property	String	N/A
OverStrike property	Logical	True, False
Owner property	String	N/A
PageSize property	Point	0
PageTiling property	SmallInt	StackPages, TileHorizontal, TileVertical
Pattern.Color property	LongInt	Black, Blue, Brown, DarkBlue, DarkCyan, DarkGray, DarkGreen, DarkMagenta, DarkRed, Gray, Green, LightBlue, Magenta, Red, White, Yellow, Transparent
Pattern.Style property	SmallInt	BricksPattern, CrosshatchPattern, DiagonalCrosshatchPattern, DottedLinePattern, EmptyPattern, FuzzyStripesDownPattern, HeavyDotPattern, HorizontalLinesPattern, LatticePattern, LeftDiagonalLinesPattern, LightDotPattern, MaximumDotPattern, MediumDotPattern, RightDiagonalLinesPattern, ScalesPattern, StaggeredDashesPattern, ThickHorizontalLinesPattern, ThickStripesDownPattern, ThickStripesUpPattern, ThickVerticalLinesPattern, VerticalLinesPattern, VeryHeavyDotPattern, WeavePattern, ZigZagPattern
PersistView property	Logical	True, False
Picture property	String	N/A
PinHorizontal property	Logical	True, False
PinVertical property	Logical	True, False
Position property	Point	N/A
PositionalOrder	SmallInt	> 0
PrecedePageHeader property	Logical	True, False

Prev property	String	N/A
PrinterDocument property	Logical	True, False
PrintOn1stPage property	Logical	True, False
ProgID property	String	N/A
Range property	SmallInt	Varies according to field type.
RasterOperation property	LongInt	MergePaint, NotSourceCopy, NotSourceErase, SourceAnd, SourceCopy, SourceErase, SourceInvert, SourcePaint
ReadOnly property	Logical	True, False
RecNo property	LongInt	N/A
Refresh property	Logical	True, False
RefreshOption property	SmallInt	PrintFromCopy, PrintLock, PrintNoLock, PrintRestart, PrintReturn
RemoveGroupRepeats property	Logical	True, False
RepeatHeader property	Logical	True, False
Required property	Logical	True, False
RightBorder property	LongInt	N/A
RowHeight property	LongInt	> 0
RowNo property	SmallInt	N/A
Scroll property	Point	N/A
SeeMouseMove property	Logical	True, False
Select property	Logical	True, False
SelectedText property	String	N/A
SeqNo property	LongInt	N/A
ShowAllColumns property	Logical	True, False
ShowGrid property	Logical	True, False
Shrinkable property	Logical	True, False
Size property	Point	N/A
SizeToFit property	Logical	True, False
SnapToGrid property	Logical	True, False
SortOrder property	Logical	Ascending, Descending

SpecialField property	SmallInt	DateField, NofFieldsField, NofPagesField, NofRecsField, PageNumField, RecordNoField, TableNameField, TimeField
SquareTabs property	Logical	True, False
StandardMenu property	Logical	True, False
StandardToolBar property	Logical	True, False
Start property	Point	N/A
StartPageNumbers property	SmallInt	> 0
Style property	SmallInt	CorelButton, Windows3dButton, WindowsButton
SummaryModifier property	SmallInt	CumulativeAgg, CumUniqueAgg, RegularAgg, UniqueAgg
TabHeight property	LongInt	User defined value
TabsAcross property	SmallInt	User defined value
TabsOnTop property	Logical	True, False
TabStop property	Logical	True, False
TableName property	String	N/A
Text property	String	N/A
ThickFrame property	Logical	True, False
Thickness property	SmallInt	LWidth10Points, LWidth1Point, LWidth2Points, LWidth3Points, LWidth6Points, LWidthHairline, LWidthHalfPoint
Title property	String	N/A
Tooltip property	Logical	True/False
TooltipText property	String	User defined.
TopBorder property	LongInt	N/A
TopLine property	LongInt	N/A
Touched property	Logical	True, False
Translucent property	Logical	True, False
UncheckedValue property	String	N/A
Value property	String	N/A
VerticalScrollBar property	Logical	True, False
Visible property	Logical	True, False
WideScrollBar property	Logical	True, False

Width property	LongInt	> 0
WordWrap property	Logical	True, False
Xseparation property	LongInt	> 0
Yseparation property	LongInt	> 0

Index

A

- Abort messages 583
- abs 349
- absolute values 349
- access rights 153
 - fields 1003
 - files 153, 164, 174
 - tables 733 - 734
 - fields 738
 - tables 793
- accessRights 153
- acos 349
- action 189
- action commands 826, 907, 911, 913, 915, 917
 - forms 247
 - objects 831 - 832, 875, 887
 - OLE applications 369 - 370
 - forms 189
 - objects 826
 - tables 690
- action constants 690, 826
 - forms 247
 - objects 831 - 832, 875, 887
 - built-in 907, 911, 913, 915, 917
 - forms 189
 - objects 826
 - tables 690
 - user-defined 40
- actionClass 40
- ActionClasses 907
- ActionDataCommands 907
- ActionEditCommands 911
- ActionEvent type 39
 - actionClass 40
 - id 41
 - setId 41 - 43
- ActionFieldCommands 913
- ActionMoveCommands 915
- actions
 - forms 189, 247
 - objects 831 - 832, 875, 887
 - OLE applications 368 - 370
 - identifying 40 - 43
 - objects 826
 - tables 690
 - user-defined 40
- ActionSelectCommands 917
 - constants 917
- activating design objects 831, 892
- activating form windows 191, 243
- activating records 692
- active built-in object variable 37
- ActiveX controls 377 - 378
 - collections 383 - 384
 - invoke 384
 - properties 382
 - registering 386
 - returning type 379 - 380
- add 620 - 621, 696
- addAddress 284
- addAlias 452 - 453
- addArray 305, 394
- addAttachment 285
- addBar 395
- addBreak 306, 396
- addButton 813
- add-in forms 42
- AddInForm 42
- AddInForm type 42
- addLast 51
- addPassword 454
- addPopUp 306, 396 - 397
- addProjectAlias 454 - 455
- address 973
- address books 286
- Address Types constants 973
- addressBook 286
- addressBookTo 286
- addressing letters 284, 286, 288, 290
- addSeparator 398
- addStaticText 307, 399
- addText 308 - 310, 400 - 401
- advancedWildcardsInLocate 456
- advMatch 503 - 504, 799 - 800
- AggModifiers 920
- aggregate constants 920
- alarms 530, 609
- aliases 452 - 455
 - adding 452 - 455, 483
 - deleting 479, 481
 - loading 476
 - password protecting 483
- returning information 458 - 460, 472 - 473
- saving 482
- setting properties 472 - 473, 484
- aliasName 697
- aligning text 1001, 1005
- Alignment 1001
- amortization 358
- angles 350 - 352, 362 - 364
- ANSI characters 505
 - conversions 526 - 528
 - key codes 264, 267
 - text files 94
 - text strings 526
 - text strings 505, 507
- ansiCode 505
- Answer 404
- Answer tables 413
 - adding cursors 434
 - creating 404, 410 - 412, 429 - 430
 - returning field structures 409, 413
 - setting sort order 414, 430
- AnyType type 43
 - blank 44
 - dataType 45
 - fromHex 45
 - isAssigned 46
 - isBlank 47
 - isFixedType 47
 - toHex 48
 - unAssign 48
 - view 49 - 50
- append 52
- appendASCIIFix 84
- appendASCIIVar 85
- appending data
 - setting specifications 85
- appending data to records 94, 112, 118
 - delimited formats 101, 106, 108, 113 - 114, 119
 - fixed formats 84, 86, 107, 111
 - from spreadsheets 109 - 110, 115, 120 - 121
 - getting 94
 - getting specifications 99 - 100, 104, 106
 - setting specifications 88, 112, 116 - 117
 - delimited formats 116, 122
 - specifications 100
- appendRow 405

Index

- appendTable 405
 - Application 49
 - Application Bar
 - displaying 823 - 824
 - displaying 822
 - application constants 963
 - applications
 - closing 561
 - returning active 554
 - shelling to DOS 559
 - type 49
 - aracters xxiv
 - arc cosine 349, 352
 - arc sine 350, 362 - 363
 - arctangent 350 - 351, 364
 - Array type 50
 - indexOf 57
 - insert 58
 - addLast 51
 - append 52
 - contains 53
 - countOf 54
 - empty 54
 - exchange 55
 - fill 55
 - grow 56
 - insertAfter 58
 - insertBefore 59
 - insertFirst 59
 - isResizeable 60
 - remove 60
 - removeAllItems 61
 - removeItem 62
 - replaceItem method 63
 - setSize 63
 - size 64
 - view method 64
 - array types 50
 - arrays. 66, 305, 369 - 370, 394, 549 - 550, 834 - 835
 - data types 50, 138
 - dynamic 138, 140 - 143
 - resizing 58
 - adding items 51 - 52, 58 - 59
 - assigning values 55
 - displaying items 64
 - exchanging items 55
 - matching values 53 - 54, 57 - 59, 63
 - removing items 54, 56, 60 - 62
 - replacing items 63
 - resizing 56, 60, 63 - 64
 - swapping indexes 55
 - Arrived property 1001
 - ASCII characters
 - extended key codes 543
 - ASCII files
 - character 118
 - character sets 94
 - reading 84 - 88, 107 - 108, 111
 - writing to 85, 90 - 91
 - character sets 101
 - asin 350
 - assignment 48, 165, 224, 276, 317, 402, 421, 476, 493, 664, 692, 754, 784, 797, 804, 862
 - assignment operator 8
 - atan 350
 - atan2 351
 - atFirst 698, 827
 - atLast 698, 828
 - attach . 190, 193, 377, 439, 449, 622, 699 - 700, 814, 828 - 830
 - attaching objects 377
 - AttachedHeader 1001
 - attachToKeyViol 701
 - AutoAppend 1001
 - automation servers 377, 382, 386 - 387
 - opening 384 - 385
 - registry 378
 - auxiliary tables 408, 421
 - averages 623, 703
 - AvgCharSize 1001
- ## B
- basic language elements 7
 - BDE System Information dialog box 468, 540
 - beep 530, 609
 - beginning-of-file indicators 804
 - beginTransaction 74 - 75
 - Binary 69 - 70
 - binary data 35
 - binary large objects 65
 - deleting 65
 - formatting 65 - 66
 - getting size 68
 - reading 66 - 67
 - saving 69 - 70
 - writing 69 - 70
 - binary numbers 281, 488
 - comparisons 280 - 283, 488 - 490
 - Binary type 65 - 66
 - clipboardErase 65
 - clipboardHasFormat 65
 - readFromClipboard 66
 - readFromFile 67
 - size 68
 - writeToClipboard 69
 - bitAND 280, 487
 - bitIsSet 281, 488
 - bitmaps
 - adding to Toolbars 813, 819
 - data type 258
 - reading 259 - 260
 - writing 260 - 262
 - magnification 1006
 - toolbar 992
 - bitOR 282, 489
 - bitwise operations 281, 488
 - long integers 280 - 283
 - small integers 488 - 490
 - bitXOR 283, 490
 - blank 44
 - blank records 1001
 - blank values 44, 47, 457, 476
 - blankAsZero 457
 - BlankRecord 1001
 - BLOBs 65
 - copying 66 - 67, 69 - 70
 - deleting 65
 - formatting 65 - 66
 - getting size 68
 - reading 66 - 67
 - saving 69 - 70
 - BMP files 260
 - Boolean values 278
 - conversions 278 - 279
 - formatting 1004
 - Border 1001
 - borders 1001
 - adding 1001, 1004
 - returning 852, 1001
 - size 1001
 - adding 1004
 - color settings 1004
 - constants 965, 991
 - returning 1004
 - size 1005
 - bot 702
 - BottomBorder 1001
 - bounding boxes 852
 - Breakable 1001

- breakApart 506
 - bringToFront 831
 - bringToTop 191
 - broadcastAction 831 - 832
 - BrowserOptions 920
 - buffers 601, 717, 744, 750, 801, 838
 - built-in event methods
 - running 10
 - built-in menus 305, 1003
 - built-in menus 318
 - built-in object variables 37
 - container 37
 - self 37
 - subject 37
 - button constants 920 - 921, 996
 - button objects
 - adding 1001
 - labeling 1001, 1005
 - ButtonStyles 920
 - ButtonType 921, 1001
 - ByRows 1001
- C**
- C and C++ languages 33, 35
 - C language 32
 - using 32
 - C++ language 32
 - using 32
 - calculated fields 4, 744, 851, 1001
 - CalculatedField 1001
 - Cancel messages 585 - 586
 - cancelEdit 703, 833
 - canceled changes 703, 717, 795, 833, 838, 901
 - canLinkFromClipboard 366
 - canReadFromClipboard 367
 - Caption 1001
 - cascading 403
 - cascading menus 306, 323 - 324, 396 - 397
 - case sensitivity 474, 476
 - text 519, 527
 - text strings 474, 516
 - text strings 517
 - cateNextPattern xxix
 - catePriorPattern xxvii
 - cAverage 623, 703
 - cCount 623, 704
 - CD-ROM drives 156, 158, 163, 172
 - ceil 351
 - CenterLabel 1001
 - CHANDLE 35
 - changing data 738, 833, 838, 840, 843, 863, 901, 1003
 - changing field values 49 - 50, 904 - 906
 - changing method definitions 234
 - char 263
 - character codes 507
 - ANSI 267
 - ASCII extended keys 543
 - keypresses 263 - 264, 269 - 271
 - OEM 521
 - text files 94, 118
 - virtual keys 264, 269 - 271, 528
 - ANSI 505, 507
 - text files 101
 - virtual keys 508, 518
 - characters 507
 - echoing 1007
 - overwriting 1007
 - repeating 508
 - charAnsiCode 264
 - Charset 923
 - chart constants 966 - 968
 - checkboxes 1001
 - CheckedValue 1001
 - checkField 406
 - checkmarks (queries) 406 - 408, 414
 - checkRow 407
 - choosing items in lists 1006
 - choosing menu 325, 331
 - choosing menu commands 325 - 332
 - choosing objects 1002 - 1003
 - chr 507
 - chrOEM 507
 - chrToKeyName 508
 - Class 1001
 - classes 40, 1001
 - returning 551
 - saving 847
 - clearCheck 408
 - clearDirLock 153
 - client areas 257
 - Clipboard 65
 - clearing 65 - 66
 - formats 65 - 66
 - graphic objects 259 - 260
 - OLE objects 366 - 367, 375 - 376
 - reading 66, 259, 301, 367, 375, 521
 - writing to 69, 260, 303, 376, 528 - 529
 - clipboardErase 65
 - clipboardHasFormat 65
 - clock 533
 - close 192, 272, 378, 440, 457, 530, 691, 705, 801
 - closing 76, 135, 192, 272, 440, 457, 530, 705, 801
 - applications 561
 - databases 76
 - DDE conversations 135
 - forms 192, 530
 - libraries 272
 - reports 440
 - sessions 457
 - tables 691, 705
 - text files 801
 - clusterID constants 997
 - cMax 624, 705
 - cMin 625, 706
 - cNpv 626, 707
 - code
 - deleting 875 - 877
 - writing 845 - 847, 878
 - comments 7
 - deleting 233
 - disabling 10
 - syntax 2
 - code pages 610 - 611
 - color constants 921, 984
 - colors 857, 891, 921, 1002
 - objects 1002
 - patterns 1007
 - records 1002
 - ellipses 1005
 - field values 1004
 - grids 1004
 - objects 1004
 - text 1004
 - column headings 1004
 - Columnar 1002
 - ColumnPosition 1002
 - columns 1004 - 1005
 - deleting 1002
 - rotating 1002
 - setting current 1002
 - size 1002
 - adding 1004 - 1005
 - counting 1006
 - ColumnWidth 1002
 - commands 305
 - comments keyword 7
 - commit 801
 - commitTransaction 76

Index

- compact 627, 707
- comparison operators 8
- comparisons 516 - 517
 - binary 280 - 281, 283, 488 - 490
 - binary numbers 282
 - floating-point numbers 356
 - text strings 516 - 517
- Compatibility 923
- compileInformation 531
- compiling 225, 250 - 252, 531
- CompleteDisplay 923, 1002
- const keyword 9
- constantNameToValue 532
- constants
 - DesktopPreferenceTypes 574, 603
 - RegistryKeyType 534, 549, 575, 596, 606
 - returning 532, 551
- ActionClasses 907
- ActionDataCommands 907
- ActionEditCommands 911
- ActionFieldCommands 913
- ActionMoveCommands 915
- AggModifiers 920
- BrowserOptions 920
- ButtonStyles 920
- ButtonTypes 921
- color 921
- Compatibility 923
- CompleteDisplay 923
- DataTransferCharset 923
- DataTransferFileType 924
- declaring 9
- DesktopPreferenceTypes 925
- ErrorReasons 926
- Errors 927
- EventErrorCodes 926
- ExecuteOptions 963
- FieldDisplayTypes 963
- FileBrowserFileTypes 963
- FontAttributes 965
- FrameStyles 965
- General 965
- GraphBindTypes 966
- GraphicMagnification 966
- GraphLabelFormats 966
- GraphLabelLocation 966
- GraphLegendPosition 967
- GraphMarkers 967
- GraphMarkerSize 968
- GraphTypeOverRide 968
- GraphTypes 968
- IdRanges 969
- Keyboard 969
- KeyBoardState 971
- LineEnd 972
- LineStyle 972
- LineThickness 973
- Linetype 973
- listed 907
- MailAddressType 973
- MailReadOptions 973
- MenuChoiceAttributes 974
- MenuCommands 974
- MenuReasons 982
- MouseShapes 982
- MoveReasons 983
- PageTilingOptions 983
- PatternStyles 984
- PrintColor 984
- PrinterOrientation 985
- PrinterType 987
- PrintQuality 987
- PrintSources 987
- publishToFilters 988
- qbeCheckType 988
- qbeRowOperation 988
- QueryRestartOption 988
- RasterOperation 989
- RegistryKeyType 989
- ReportPrintPanel 990
- ReportPrintRestart 990
- StatusReasons 991
- TableFrameStyles 991
- TextAlignment 991
- TextDesignConstants 992
- TextSpacing 992
- ToolBarBitmap 992
- ToolBarButtonType 996
- ToolBarState 997
- TrackBar 998
- TrackBarStyles 998
- UIObjectTypes 998
- UpdateLink 999
- user-defined 40, 144, 325
- ValueReasons 999
- WindowStyles 999
- constantValueToName 532
- container built-in object variable 37
- ContainerName 1002
- containers 862
- getting names 1002
- testing validity 862
- getting names 1007
- returning objects 1003, 1007 - 1008
- contains 53, 139, 311
- context-sensitive menus 305
- Control menus 1002
- control structures 7
- ControlMenu 1002
- conversions 526, 532, 586
 - constants 532
 - dates and time 124, 127, 571 - 573
 - memos 300
 - monetary values 71 - 72
 - numbers 278 - 279, 284, 357 - 358, 490 - 491, 572
 - screen coordinates 391, 616, 887, 901
 - strings 519, 525, 527, 571 - 573
 - numbers 45, 48
- convertPointWithRespectTo 833
- copy 154, 628, 708
- copyFromArray 709, 834
- copying 154, 628, 708, 834 - 835
 - BLOBs 66 - 67, 69 - 70
 - data 620 - 621, 709 - 710, 834 - 835
 - files 154
 - objects 836
 - OLE objects 367, 372 - 376
 - tables 536, 620 - 621, 628, 708
 - text 301 - 304, 521, 528 - 529
 - data 711
- copyRecord 710
- copyToArray 711, 835
- copyToToolBar 836
- cos 352
- cosh 352
- cosine 349, 352
- Count 312
- countOf 54
- CPUClockTime 533
- create 193, 273, 449, 628 - 636, 674 - 675, 802, 815, 837
- createAuxTables 408
- createIndex 637 - 638, 712 - 714
- createQBEStrng 409
- critical errors 559, 581
- crosstabs 851, 1002 - 1003
- cSamStd 639, 715
- cSamVar 640, 715
- cStd 641, 716

Index

cSum	642, 717	saving	701, 776, 779, 798, 888, 890	isTable	80
Currency	70 - 72	sorting	542, 683, 790	open	81
Currency type	71 - 72	updating	193 - 194, 208, 539, 782 - 783, 797 - 798	rollBackTransaction	82
currency values		validity checks	760	setMaxRows	82
formatting	567	changing	1003	transactionMethod	83
Currency values	70	displaying	1004	databases	73
conversions	71 - 72	entering	1007	adding tables	197, 203, 212, 535 - 536, 628 - 636
types	70	sorting	1003	assignment	79
current date types		validity checks	1003	closing	76
today	125 - 126	data models	208 - 209	opening	73, 81, 470
CurrentColumn	1002	adding tables	197, 203, 212	removing tables	77, 208, 537, 643
currentPage	440, 1002	attaching to tables	197, 721	returning information	470 - 474
CurrentRecordMarker.Color	1002	linking tables	199, 203 - 207, 211, 215	DataSource	1002
CurrentRecordMarker.LineStyle	1002	object types	73, 404, 491, 620, 695	DataTransfer constants	923 - 924
CurrentRecordMarker.Show	1002	properties	201 - 202, 210, 212	DataTransfer type	89, 93, 97, 104, 106
CurrentRow	1002	querying tables	198	dlgExport	85
currRecord	717, 838	removing tables	208	im	
CursorColumn	1002	returning field values	200	se	
CursorLine	1002	synchronizing	208 - 209	appendASCIIFix procedure	84
CursorPos	1002	writing data	208	appendASCIIVar procedure	85
cursors	695	data transfers	123	dlgImport procedure	85
Answer tables	434	initializing structures	89	exportASCIIFix procedure	90
data models	197, 208 - 209, 721	type	83	exportASCIIVar procedure	91
field objects	1002, 1006	data types	43	exportParadoxDOS	92
SQL tables	497	arrays	50, 138	getAppend method	94
type	695	assigning	224, 276, 317, 402, 692, 804, 862	getDestDelimiter method	96
custom menus	252, 447 - 448	BLOBs	65	getDestType method	99
reusing	1003	dates and time	123, 811	getKeyviol method	100
custom methods	213 - 214	declaring	123, 811	getProblems method	100
calling	849 - 850	graphics	258	getSourceDelimitedFields method	102
changing	234	logical	278	getSourceDelimiter method	103
deleting	875 - 877	numeric	70, 279, 347	getSourceRange method	105
saving	847	OLE	377	getSourceSeparator method	106
tracking	234, 273 - 274, 845 - 846	records	436	loadDestSpec method	111
writing	878	text	300, 502	loadSourceSpec method	111
deleting	233	assigning	48	setAppend method	112
cVar	642, 718	database	610 - 611	setDest	112
		database engine	452	setDestDelimiter method	114
		enabling	452, 476, 478	setDestSeparator method	116
		returning information	468, 540, 594	setKeyviol method	116
		Database type	73	setProblems method	117
		beginTransaction	74 - 75	setSource method	117
		close	76	setSourceDelimiter method	119
		commitTransaction	76	DataTransferCharset	923
		delete	77	DataTransferCharset constants	118
		enumFamily	77	DataTransferDelimitCode constants	95, 114, 119
		getMaxRows	78	DataTransferFileType	924
		isAssigned	79	DataTransferFileTypes	99, 117
		isSQLServer	79	date	123
				date procedure	124

D

data	89, 325
changing	717, 723, 754, 795, 833, 838, 843, 901, 1003
copying	620 - 621, 709 - 710, 834 - 835
deleting	542 - 543, 627, 707, 724, 758, 760, 841 - 842, 865
finding	761 - 767, 867 - 872
matching patterns	761 - 767, 867 - 872
protecting	454, 480, 483, 665, 688, 756

Index

- date types
 - date 124
 - dateVal 125
- DateRangeTypes
 - report constants 924
- dates 123
 - conversions 124, 127, 571 - 573
 - finding leap years 130
 - formatting 123, 567
 - returning 127 - 129, 132 - 134
 - data types 126
 - formatting 1004
- DateTime 127
- DateTime types
 - day 127
 - daysInMonth 128
 - dow 128
 - dowOrd 129
 - doy 129
 - hour 130
 - milliSec 131
 - minute 131
 - month 132
 - second 133
 - year 134
- dateVal 125
- day 127
- daysInMonth 128
- dBASE tables 635 - 636
 - deleting 795, 901
 - deleting records 627, 682, 707
 - exporting 92
 - language drivers 635 - 636
 - moving through 770, 884
 - numeric fields 741, 1003
 - removing records 758, 760, 865, 1002
- DDE 134
- DDE links 134
 - closing 135
 - executing 136
 - opening 136
 - specifying items 137 - 138
 - type 134
- DDE type
 - close 135
 - open 136
 - setItem 137 - 138
 - DDE type 134
- debug 533
- debugging 225, 250 - 252, 533
 - tracing code 613 - 615
- decimal numbers 45, 48
- decimal places 741, 1003
 - dBASE tables 741, 1003
- declarations 2 - 3
 - data types 123, 811
 - data types 47, 126
 - external libraries 23
 - external routines 28
 - prototypes 2
- Default 1002
- default menus 305, 318
- default properties 1002
- DefineGroup 1002
- delays 256, 608, 694
- delayScreenUpdates 193 - 194
- delete 154, 643, 838
- DeleteColumn 1002
- Deleted 1002
- deleteDir 155
- deleteRecord 719, 839
- deleteRegistryKey 534
- DeleteWhenEmpty 1002
- deleting 154, 643, 838
 - aliases 479, 481
 - BLOBs 65
 - columns in tables 1002
 - data 542 - 543, 627, 707, 724, 758, 760, 841 - 842, 865
 - directories 155
 - empty fields 1002
 - files 154
 - indexes from tables 645, 722
 - menu commands 313, 318
 - menus 318
 - methods 875 - 877
 - objects 838
 - passwords 480
 - tables 77, 208, 537, 643
 - Toolbars from 228, 818
 - Toolbars from forms 224, 817, 819
 - array items 54, 56, 60 - 62, 140, 142
 - methods 233
- DelimitCode
 - DataTransfer constants 923
- Delimited ASCII Export dialog 91
- delimited ASCII files 85, 87, 91, 108
 - returning information. 94 - 96, 98, 101 - 102, 106
 - setting specifications 113 - 114, 119
 - returning information 103
 - setting specifications 116, 122
- deliver 195
- delivering
 - forms 195
- derived methods 6
- design 195, 441
- design objects
 - activating 831, 892
 - binding 828 - 830
 - bounding boxes 852
 - color settings 857, 891
 - containers 862
 - copying 836
 - creating 837
 - deleting 838
 - event 875
 - event handling 831 - 832, 865, 887
 - filtering 840, 852, 893 - 894
 - multiple objects 1002
 - pinning objects 1002
 - position 854, 895
 - resizing 1002
 - resizing specific objects 1002
 - returning 845, 848
 - selecting specific objects 1002
 - synchronizing 891
 - types 305
 - viewing 902
 - multiple objects 1003, 1007 - 1008
 - naming 1007
 - scrolling 1005
 - setting coordinates 1007
 - types 825
- design windows
 - forms 248, 1003
 - reports 441, 443, 447, 1003
 - forms 1004
 - reports 1004
- Design.ContainObjects 1002
- Design.PinHorizontal 1002
- Design.PinVertical 1002
- Design.Selectable 1002
- Design.SizeToFit 1002
- designing 195, 221, 441
 - forms 193, 195, 226, 1003
 - reports 441, 1003
- DesignModified 1003
- DesignSizing 1003
- Desktop 49, 1003
- desktop preferences 925

- desktop WindowHandles 544
- DesktopForm 1003
- desktopMenu 535
- detail tables 199, 203 - 207, 211
- device independent 260
- device independent bitmaps 259
- DeviceType 926
- dialog 64, 437 - 438
- dialog boxes 246, 1006
 - opening 246
 - opening forms as 535, 1003
 - position 220, 253
 - system messages 583 - 586
 - displaying field 49 - 50
- DialogForm 1003
- DialogFrame 1003
- DIB 259
- DIB formats 260
- didFlyAway 720
- dimmed objects 1003
- directories
 - creating 169
 - deleting 155
 - paths 163, 171, 175 - 178, 180 - 182, 184 - 185
 - private 170, 175 - 176
 - startup 182
 - validating 165, 170, 185
 - working 177 - 178, 185
 - locking 172
 - type 152
- disableBreakMessage 196
- disableDefault keyword 10
- disablePreviousError 196
- disk drives
 - available space 161 - 162, 183 - 184
 - returning 156, 158, 163, 167 - 168
 - setting default 173
 - type 152
- disk errors 186 - 188
- display constants 923
- Display Managers 188
 - reports 438
 - scripts 448
 - tables 689
 - applications 49
 - forms 188, 1006
- displaying 256, 323 - 324, 403, 821, 823 - 824
 - Application Bar 823 - 824
- field values 437 - 438, 698, 725, 744, 749, 771 - 772, 777, 843, 851, 858, 884 - 885, 889, 899, 1002
 - forms 256
 - graphics 258
 - menu commands 313 - 316, 319 - 322
 - menus 306, 318, 323 - 324, 396 - 397, 403 - 404, 535
 - objects 902
 - tables 693
 - text 1001
 - Toolbars 224, 228, 256, 817 - 819, 821
 - Application Bar 822
 - column heads 1004
 - columns and rows 1004
 - data 1004
 - grids 1004
 - menu commands 317
 - reports 1007
 - text 1004
- DisplayType 1003
- distance 389
- division 357
- dlgAdd 535
- dlgCopy 536
- dlgCreate 536
- dlgDelete 537
- dlgEmpty 537
- dlgExport 85
- dlgImport 85
- dlgImportASCIIFix 86
- dlgImportASCIIVar 87
- dlgImportSpreadsheet 87
- dlgImportTable 88
- dlgNetDrivers 537
- dlgNetLocks 538
- dlgNetRefresh 539
- dlgNetRetry 539
- dlgNetSetLocks 539
- dlgNetSystem 540
- dlgNetUserName 540
- dlgRename 541
- dlgRestructure 541
- dlgSort 542
- dlgSubtract 542
- dlgTableInfo 543
- DLL
 - uses block 28
- DLLs 748
- dmAddTable 197
- dmAttach 197, 721
- dmBuildQueryString 198
- dmEnumLinkFields 199
- dmGet 200
- dmGetProperty 201 - 202
- dmHasTable 203
- dmLinkToFields 203 - 205
- dmLinkToIndex 206 - 207
- dmPut 208
- dmRemoveTable 208
- dmResync 208 - 209
- dmSetProperty 210
- dmUnlink 211
- doDefault keyword 10
- DOS 559
 - executing commands 559
 - returning information 545, 593, 618
 - running programs 559
- dow 128
- dowOrd 129
- doy 129
- DrillDown property 1003
- drivers 537
 - language 432, 574, 610 - 612, 633, 635, 746 - 747
 - system 461 - 467, 537
- drives 156
 - available space 161 - 162, 183 - 184
 - returning 156, 158, 163, 167 - 168
 - setting default 173
 - type 152
- drop-down lists
 - adding items 1003
 - adding items 1002, 1006
 - selecting items 1006
- drop-down menus 306, 323 - 324, 396 - 397, 403
- dropGenFilter 644, 721, 840
- dropIndex 645, 722
- dynamic arrays 138
 - assigning values 141
 - data type 138
 - displaying items 143
 - matching values 139
 - removing items 140, 142
 - resizing 140, 142
- dynamic data exchange 134
 - links 135 - 138
- DynArray 138
- DynArray type
 - contains 139
 - empty 140

Index

getKeys	141	enumDriverTopics	467	eot	736
removeItem	142	enumEngineInfo	468	EPS files	260
size	142	enumEnvironmentStrings	545	error codes	146, 150, 555
		enumEvents	380	clearing from stack	555 - 556
		enumExperts	546	returning	146, 552, 555, 557
		enumFamily	77	setting	150
		enumFieldNames	647, 726, 843	SQL tables	556, 558
		enumFieldNamesInIndex	647, 726	error constants	144, 926
		enumFieldStruct	409, 648 - 649, 727 - 728	error messages	558
		enumFileList	157	clearing from stack	555
		enumFolder	469	returning	552, 557 - 558
		enumFonts	547	error stack	
		enumFormats	547	clearing	555 - 556, 558
		enumFormNames	548	displaying	555, 557 - 559
		enumInbox	289	SQL errors	556, 558
		enumIndexStruct	650 - 651, 729 - 730	errorClear	555
		enumLocks	731, 844	errorCode	146, 555
		enumMethods	381	ErrorEvent	144
		enumObjectNames	845	ErrorEvent type	144
		enumObjects	381	reason	145
		enumOpenDatabases	470	setReason	145
		enumPrinters	548	errorHasErrorCode	556
		enumProperties	382	errorHasNativeErrorCode	556
		enumRefIntStruct	652, 731 - 732	errorLog	557
		enumRegistryKeys	549	errorMessage	558
		enumRegistryValueNames	550	errorNativeCode	558
		enumReportNames	550	errorPop	558
		enumRTLClassNames	551	ErrorReasons	926
		enumRTLConstants	551	errors	
		enumRTLErrors	552	disk	186 - 188
		enumRTLMethods	553	event type	144
		enumSecStruct	653 - 654, 733 - 734	events	146, 150
		enumServerClassNames	369	handling	555 - 558
		enumServerInfo	382	reporting	145, 196, 559
		enumSource	213, 273, 845 - 846	run-time	555
		enumSourcePageList	89	SQL tables	556, 558
		enumSourceRangelist	90	trapping	559, 581
		enumSourceToFile	214, 274, 847	user-defined constants	144
		enumTableLinks	215	run-time	552, 558
		enumTableProperties	735	Errors	927
		enumToolBarNames	816	errorShow	559
		enumUIClasses	847	errorTrapOnWarnings	559
		enumUIObjectNames	216, 441, 848	event constants	926
		enumUIObjectProperties	217, 442, 848	Event type	146
		enumUsers	471	errorCode	146
		enumVerbs	369 - 370	getTarget	147
		enumWindowHandles	554	isFirstTime	148
		enumWindowNames	554	isPreFilter	149
		environment	545, 593, 618	isTargetSelf	149
		eof	804	reason	150
echoing characters	1007				
ecute	xxv				
edit	368, 723, 840				
Edit mode	723, 725, 754, 840, 843, 863, 1003				
enabling	723, 840				
exiting	725, 843				
testing existence	754, 863				
Editing	1003				
editing data	39, 703, 723, 725, 754, 833, 840, 843, 863, 1003				
editing OLE objects	368				
editing text	1003				
electronic mailings	284				
ellipses	973, 1005 - 1006				
e-mail	284				
empty 54, 89, 140, 287, 313, 646, 724, 815, 841 - 842					
empty strings	502, 518				
emptyAddresses	288				
emptyAttachments	288				
emptying tables	537, 646, 665, 755, 863				
Enabled	1003				
enableDefault keyword	11				
enableExtendedCharacters	543				
encryption	454, 483				
adding passwords	454, 483				
removing passwords	480				
testing	665, 688, 756				
end	725, 803, 843, 1003				
endEdit	725, 843				
end-of-file indicators	804				
enumAliasLoginInfo	458				
enumAliasNames	459 - 460				
enumAutomationServers	378				
enumClipboardFormats	66				
enumConstants	379				
enumConstantValues	379				
enumControls	380				
enumDataBaseTables	461				
enumDataModel	212				
enumDesktopWindowHandles	544				
enumDesktopWindowNames	544				
enumDriverCapabilities	461 - 465				
enumDriverInfo	466				
enumDriverNames	467				

- setErrorCode 150
 - setReason 151
 - event types 39, 144, 146, 261, 324, 332, 344, 498, 823, 904
 - EventErrorcodes 926
 - events 146
 - changing field 904
 - changing field values 905 - 906
 - errors 146, 150
 - event type 146
 - handling 148 - 151, 831 - 832, 875, 887
 - keyboard 228 - 230, 261, 865
 - menu 233, 324 - 325
 - mouse 235 - 242, 332, 878 - 882
 - move 344
 - push buttons 890
 - status bar 498
 - targets 147, 149
 - mouse 1006
 - User-defined action constants 40
 - exchange 55
 - execMethod 275, 849 - 850
 - execute 136, 559
 - ExecuteOptions 963
 - executeQBE 410 - 412
 - executeSQL 492
 - executeString 560
 - executing 136, 189, 275, 410 - 412, 492, 559
 - DDE conversations 136
 - DOS commands and programs 559
 - forms 196, 217 - 218, 248, 256, 608
 - methods 275, 694, 849 - 850
 - queries 410 - 412, 416, 423, 432 - 434
 - scripts 449, 451
 - SQL statements 492, 497
 - methods 561
 - existDrive 158
 - exit 561
 - exp 353
 - expert directory 561
 - Experts 546, 595
 - expertsDir 561
 - expertsDir procedure 561
 - exponentials 353, 359 - 360
 - exportASCIIIFix 90
 - exportASCIIVar 91
 - exporting data
 - delimited 94 - 96, 98, 106, 113 - 114, 119
 - delimited formats 91
 - fixed formats 90
 - from 97, 104, 115, 120
 - from spreadsheets 89, 105, 121
 - getting 100
 - getting 100
 - getting specifications 94, 97, 99, 104, 106
 - Paradox DOS formats 92
 - setting 112, 116 - 117
 - setting specifications 112, 117 - 118
 - spreadsheets 93
 - type 83
 - exportParadoxDOS 92
 - exportSpreadsheet 93
 - extended characters 543
- ## F
- families 77, 655, 737
 - familyRights 655, 737
 - fetch limit 78
 - Fetch Limit
 - query 82
 - field 797
 - field constants 963, 965
 - field names 656, 737, 1003
 - returning 647, 656, 726, 737, 843
 - setting 1003
 - field squeeze 1003
 - field translations 636
 - field values
 - assigning 165, 276, 317, 692, 804
 - averaging 623, 703
 - blank 44, 457, 476
 - changing 904 - 906
 - default 1002
 - displaying 437 - 438, 698, 749, 777, 889, 1002
 - event type 904
 - ranges 661, 680, 689, 747, 787, 856, 897
 - returning 623 - 625, 642, 704 - 706, 717, 760, 777
 - searching 761 - 767, 867 - 872
 - true/false 278 - 279
 - truncating 365
 - updating 193 - 194, 208, 539, 782 - 783
 - color settings 1004
 - displaying 49 - 50, 64, 789
 - templates 1007
 - true/false 1004
 - Field View 1003, 1007
 - FieldDisplayTypes 963
 - fieldName 656, 737, 1003
 - fieldNo 656, 738, 1003
 - fieldRights 1003
 - fields
 - access rights 1003
 - deleting empty 1002
 - insertion point 1002
 - naming 1003
 - returning information 657 - 658, 669, 738 - 739, 885, 1003
 - size 739, 1003
 - access rights 738
 - entering data 1007
 - insertion point 1006
 - fieldSize 739, 1003
 - FieldSqueeze 1003
 - fieldType 657 - 658, 740, 1003
 - fieldUnits2 741, 1003
 - FieldValid 1003
 - fieldValue 742 - 743
 - FieldView 1003
 - file 153
 - file attributes 164, 174
 - File Browser 562 - 564
 - constants 963
 - fileBrowser 562 - 564
 - fileBrowserEx 562 - 564
 - FileBrowserFileTypes 963
 - FileBrowserInfo 562 - 564
 - FileBrowserInfo constants 920
 - files 798
 - access rights 153, 164, 174
 - associated forms 219
 - browsing 562 - 564
 - copying 154
 - data transfers 83, 89, 123
 - deleting 154
 - family members 77, 655, 737
 - finding 158 - 160, 162, 170, 562 - 564
 - renaming 171
 - returning information 164, 166, 183, 461, 469
 - type 152
 - FileSystem 152
 - FileSystem type
 - clearDirLock 153
 - copy 154
 - delete 154
 - deletedDir 155
 - drives 156
 - enumFileList 157
 - existDrive 158

Index

- findFirst 158 - 159
- findNext 160
- freeDiskSpace 161 - 162
- fullName 162
- getDir 163
- getDrives 163
- getFileAccessRights 164
- getValidFileExtensions 164
- isAssigned 165
- isDir 165
- isFile 166
- isFixed 167
- isRemote 167
- isRemovable 168
- isValidDir 168 - 169
- makeDir 169
- name 170
- privDir 170
- rename 171
- setDir 171
- setDrive 173
- setFileAccessRights 174
- setPrivDir 175 - 176
- setWorkingDir 177 - 178
- shortName 179
- size 180
- splitFullFileName 180 - 181
- startUpDir 182
- time 183
- totalDiskSpace 183
- totalDiskSpaceEX 184
- windowsDir 184
- windowsSystemDir 185
- workingDir 185
- accessRights method 153
- setDirLock 172
- FileType 924
- fill method 55
- fill patterns 508
 - color settings 1007
 - constants 984
 - defining 1007
 - repeating characters 508
- fill procedure 508
- filters 689
 - design objects 840, 852, 893 - 894
 - tables. 644, 659 - 660, 677 - 678, 721, 745, 785 - 786
- finances 354, 358, 360, 626, 707
 - future values 354
 - periodic payments 358
 - present values 360, 626, 707
- findFirst 158 - 159
- finding data 761 - 767, 867 - 872
- finding matches
 - arrays 53 - 54, 57 - 59, 63, 139
 - enabling wildcards 456, 475
 - filenames 158 - 160, 162, 170
 - records 761 - 767, 867 - 870, 872
 - specifying case sensitivity 476
 - text strings. 503 - 504, 520, 522 - 523, 799 - 800
- findNext 160
- first 383, 1003
- FirstRow 1003
- FitHeight 1003
- FitWidth 1003
- fixed format ASCII files 84, 86, 90, 107
 - loading 111
- Fixed Length ASCII Export dialog 90
- Fixed Length ASCII Import dialog box 86, 107
- fixed-size arrays 50
- flat look 1003
- FlatLook 1003
- floating toolbars 815
- floating Toolbars
 - emptying 815
 - position 816, 820
- floating-point numbers 347
 - comparisons 356
 - conversions 358
 - data 347
- floating—point numbers
 - absolute values 349
 - conversions 357
 - dividing 357
 - fractional part 353
 - random 361
 - rounding 351, 353, 362
 - truncating 365
- floor 353
- FlyAway 1003
- Focus 1003
- focus (objects) 1001, 1003, 1006
 - moving 1007
- folders 469
 - returning information 469
- font constants 965
- Font.Color 1004
- Font.Style 1004
- Font.Typeface 1004
- FontAttributes 965
- fonts 547, 965, 1001
- for keyword 11
- forceRefresh 744, 851
- forEach keyword 12
- Form 188, 248
- Form Design windows 195, 223, 226, 228, 231, 248
- form letters 284
 - adding attachments 285, 291
 - addressing 284, 286, 288, 290
 - clearing contents 287
 - deleting attachments 288
 - returning content 292 - 293
 - sending 294 - 295, 297
 - setting content 298 - 299
 - type 284
- Form type 188, 212
 - attach 190
 - bringToTop 191
 - close 192
 - create 193
 - deliver 195
 - design 195
 - disableBreakMessage 196
 - disablePreviousError 196
 - dmAddTable 197
 - dmAttach 197
 - dmBuildQueryString 198
 - dmEnumLinkFields 199
 - dmGet 200
 - dmGetProperty 201 - 202
 - dmHasTable 203
 - dmLinkToFields 203 - 205
 - dmLinkToIndex 206 - 207
 - dmPutdmPut method/procedure 208
 - dmRemoveTable 208
 - dmResyncdmResync method/procedure 208 - 209
 - dmSetProperty 210
 - dmUnlink 211
 - enumDataModel 212
 - enumUIObjectNames 216
 - enumUIObjectProperties 217
 - formCaller 217
 - formReturn 218
 - getFileName 219
 - getPosition 220
 - getProtoProperty 220
 - getStyleSheet 222
 - getTitle 223
 - hide 223

- isAssigned 224
- isCompileWithDebug 225
- isDesign 226
- isMaximized 227
- isMinimized 227
- isVisible 228
- keyChar 228
- keyPhysical 229 - 230
- load 231
- maximize 232
- menuAction 233
- methodEdit 234
- methodGet 234
- methodSet 234
- minimize 235
- mouseDouble 235
- mouseDown 236
- mouseEnter 237
- mouseExit 238
- mouseMove 239
- mouseRightDouble 239
- mouseRightDown 240
- mouseRightUp 241
- mouseUp 242
- moveTo 243
- moveToPage 244
- save 248
- saveStyleSheet 249
- selectCurrentTool 249
- setCompileWithDebug 250 - 251
- setIcon 252
- setMenu 252
- setPosition 253
- setProtoProperties 254
- setSelectedObjects 254
- setStyleSheets 255
- setTitle 255
- show 256
- showToolBar 256
- wait 256
- windowClientHandle 257
- windowHandle 257
- writeText 258
- action method/procedure 189
- delayScreenUpdates procedure 193 - 194
- enumSource method/procedure 213
- enumSourceToFile method/procedure 214
- getSelectedObjects method/procedure 221
- isToolBarShowing procedure 228
- methodDelete 233
- open method 244 - 245
- openAsDialog method 246
- postAction method 247
- Form windows 191 - 192, 220, 223, 228, 243 - 246, 248, 253, 1001
- format 509 - 515
- Format.DateFormat 1004
- Format.LogicalFormat 1004
- Format.NumberFormat 1004
- Format.TimeFormat 1004
- Format.TimeStampFormat 1004
- formatAdd 565
- formatDelete 565
- formatExist 566
- formatGetSpec 566
- formatSetCurrencyDefault 567
- formatSetDateDefault 567
- formatSetDateTimeDefault 567
- formatSetLogicalDefault 568
- formatSetLongIntDefault 569
- formatSetNumberDefault 569
- formatSetSmallIntDefault 570
- formatSetTimeDefault 570
- formatStringToDate 571
- formatStringToDateTime 572
- formatStringToNumber 572
- formatStringToTime 573
- formatting 547, 565
- BLOBs 65 - 66
- currency 567
- dates and 811
- dates and time 123, 567, 570
- default settings 567 - 570
- logical values 568
- numbers 569 - 570
- returning information 565 - 566
- text strings 509 - 515
- dates and time 126, 1004
- logical values 1004
- numbers 1004
- formCaller 217
- FormOpenInfo 244 - 245
- formReturn 218
- forms 188
- activating 191, 243
- changing pages 244
- client areas 257
- closing 192, 530
- compiling 225, 250 - 252, 531
- creating 193
- custom menus 252
- debugging 225, 250 - 252, 533, 613 - 615
- delivering 195
- designing 193, 195, 221, 226, 1003
- disabling interrupts 196
- displaying 256
- event handling 228 - 230, 233, 235 - 242
- hiding windows 223, 228
- moving through 244
- multi 1002
- naming 190, 223, 255, 1001
- opening 190, 195, 244 - 246, 548
- returning information 212 - 217, 219
- running 196, 217 - 218, 247 - 248, 256, 608
- saving 195, 248 - 249
- selecting objects 221, 254
- sizing 227, 235, 256
- style sheets 222, 249, 255, 573, 602 - 603
- type 188
- updating data 193 - 194, 208, 539, 782 - 783, 797 - 798
- changing pages 1007
- inserting records 1005
- moving through 1007
- multi-record 1001 - 1003, 1007
- running 189
- sizing 1006
- fraction 353
- fractions 353
- Frame.Color 1004
- Frame.Style 1004
- Frame.Thickness 1004
- FrameObjects 1004
- frames 1001
- adding 1001, 1004
- returning 852, 1001
- size 1001
- adding 1004
- color settings 1004
- constants 965, 991
- returning 1004
- size 1005
- FrameStyles 965
- freeDiskSpace 161
- freeDiskSpaceEx 162
- fromHex 45
- fullName 162, 1004
- FullSize 1004
- future values 354
- fv method 354

Index

G

General	965
general constants	965
getAddress	290
getAddressCount	290
getAliasPath	472
getAliasProperty	472 - 473
getAnswerFieldOrder	413
getAnswerName	413
getAnswerSortOrder	414
getAppend	94
getAttachment	291
getAttachmentCount	291
getBoundingBox	852
getCheck	414
getCriteria	415
getDefaultPrinterStyleSheet	573
getDefaultScreenStyleSheet	573
getDesktopPreference	574
getDestCharSet	94
getDestDelimitedFields	95
getDestDelimiter	96
getDestFieldNamesFromFirst	97
getDestination	345
getDestName	97
getDestSeparator	98
getDestType	99
getDir	163
getDrive	163
getFileAccessRights	164
getFileName	219
getGenFilter	659 - 660, 745, 852
getHTMLTemplate	853
getIndexName	746
getKeys	141
getKeyViol	100
getLanguageDriver	574, 746
getLanguageDriverDesc	747
getMaxRows	78
getMenuChoiceAttribute	313
getMenuChoiceAttributeById	314 - 315
getMessage	292
getMessageType	292
getMousePosition	333
getMouseScreenPosition	574
getNetUserName	474
getObjectHit	334
getPosition	220, 816, 854
getProblems	100

getProperty	855
getPropertyAsString	856
getProtoProperty	220
getQueryRestartOptions	416
getRange	661, 747, 856
getRegistryValue	575
getRGB	857
getRowID	416
getRowNo	417
getRowOp	417
getSelectedObjects	221
getSender	293
getServerName	371
getSourceCharSet	101
getSourceDelimitedFields	102
getSourceDelimiter	103
getSourceFieldNamesFromFirst	104
getSourceName	104
getSourceRange	105
getSourceSeparator	106
getSourceType	106
getState	817
getStyleSheet	222
getSubject	293
getTableID	418
getTableNo	418
getTarget	147
getting data types	45
getting file information	157, 164, 166, 180, 183
getting form objects	213 - 214, 216, 221
getting OLE servers	369, 371
getting report objects	441 - 442
getTitle	223
getUserLevel	576
getValidFileExtensions	164
GIF files	260
graph constants	966 - 968
GraphBindTypes	966
Graphic	258
graphic and binary data	35
graphic constants	966
graphic objects	258
data type	258
importing	260
loading	259 - 260
saving	260 - 262
magnification	1006
Graphic type	258
writeToFile	261 - 262
readFromClipboard method	259

readFromFile method	260
writeToClipboard method	260
GraphicMagnification	966
graphics types	35, 258, 812
GraphLabelFormats	966
GraphLabelLocation	966
GraphLegendPosition	967
GraphMarkers	967
GraphMarkerSize	968
graphs	851
constants	966 - 968
GraphTypeOverRide	968
GraphTypes	968
Grid.Color	1004
Grid.GridStyle	1004
Grid.RecordDivider	1004
GridLines.Color	1004
GridLines.ColumnLines	1004
GridLines.HeadingLines	1004
grids	
tables	1004
grouping records	1002, 1008
grow	56

H

handle	748
handles	49, 73, 257, 438
hasCriteria	419
hasMenuChoiceAttribute	316
hasMouse	858
Header	1005
headers	
reports	1007
table frames	1001
reports	1005, 1007
table frames	1005
HeadingHeight	1005
Headings	1005
help	576 - 580
helpOnHelp	576
helpQuit	577
helpSetIndex	577
helpShowContext	578
helpShowIndex	579
helpShowTopic	580
helpShowTopicInKeywordTable	580
hexadecimal numbers	45, 48
hide	223, 817
hideToolBar	224

- hiding objects 1005
 - hiding Toolbars 224, 228, 256, 817 - 819
 - hiding windows 223, 228
 - highlight current record 1002
 - home 749, 804, 858
 - horizontal Toolbars 820
 - HorizontalScrollBar 1005
 - hour 130
 - HTML 853, 1003, 1005
 - HTMLAction property 1005
 - HTMLFormParams property 1005
 - HTMLMethod property 1005
 - hyperbolic cosine 349, 352
 - hyperbolic sine 350, 362 - 363
 - hyperbolic tangent 350 - 351, 364
- I**
- id 41, 325 - 326
 - ID numbers 41
 - events 601
 - menus 314 - 315, 321 - 322, 325 - 326, 332
 - Windows messages 616
 - events 41 - 43
 - identifiers
 - cluster 997
 - IdRanges 969
 - if keyword 13
 - Ignore messages 583
 - ignoreCaseInLocate 474
 - ignoreCaseInStringCompares 516
 - iif keyword 14
 - Import options dialog 85
 - Import options dialog box 88
 - importASCIIFix 107
 - importASCIIVar 108
 - importing 84, 86, 107
 - importing data 83, 118
 - delimited formats 85, 87, 94 - 96, 98, 102, 106, 108, 113 - 114, 119
 - fixed formats 111
 - from 89
 - from spreadsheets 87, 97, 104 - 105, 109 - 110, 115, 120 - 121
 - getting specifications 94, 97, 99 - 100, 104, 106
 - setting 88
 - setting specifications 85, 112, 116 - 117
 - type 83
 - importing graphics 260
 - importSpreadsheet 109 - 110
 - InactiveColor 1005
 - IncludeAllData 1005
 - index 662 - 663
 - indexes 662 - 663
 - adding to tables 637 - 638, 662 - 663, 712 - 714
 - deleting 645, 722
 - rebuilding 672, 780 - 781
 - returning information 647, 650 - 651, 669, 726, 729 - 730, 746, 758, 773, 886
 - specifying 679, 688, 900
 - returning information 1005
 - IndexField 1005
 - indexOf 57
 - informational messages 584
 - initialization 89, 750
 - initRecord 750
 - insert 58
 - insertAfter 58
 - insertAfterRecord 750, 859
 - insertBefore 59
 - insertBeforeRecord 751, 860
 - InsertColumn 1005
 - InsertField 1005
 - insertFirst 59
 - Inserting 1005
 - insertion point 695
 - Answer tables 434
 - data models 197, 208 - 209, 721
 - field objects 1002
 - SQL tables 497
 - ty 372 - 374
 - insertObject 372 - 374
 - insertRecord 752, 861
 - insertRow 420
 - insertTable 420
 - instantiateView 753
 - int 490
 - integers 279, 347, 486
 - interrupts 196
 - inverse sine 350
 - Invisible 1005
 - isAbove 389
 - isAdvancedWildcardsInLocate 475
 - isAltKeyDown 264
 - isAppBarVisible 823 - 824
 - isAssigned 46, 79, 165, 224, 276, 317, 402, 421, 476, 493, 664, 692, 754, 804, 862
 - isBelow 390
 - isBlank 47
 - isBlankZero 476
 - isCompileWithDebug 225
 - isContainerValid 862
 - isControlKeyDown 265, 335
 - isCreateAuxTables 421
 - isDesign 226
 - isDir 165
 - isEdit 754, 863
 - isEmpty 422, 516, 665, 755, 863
 - isEncrypted 665, 756
 - isErrorTrapOnWarnings 581
 - isExecuteQBELocal 423
 - isFile 166
 - isFirstTime 148
 - isFixed 167
 - isFixedType 47
 - isFromUI 265, 327 - 329, 335
 - isIgnoreCaseInLocate 476
 - isIgnoreCaseInStringCompares 517
 - isInMemoryTCursor 756
 - isInside 336
 - isLastMouseClickedValid 864
 - isLastMouseRightClickedValid 864
 - isLeapYear 130
 - isLeft 390
 - isLeftDown 336
 - isLinked 375
 - isMaximized 227
 - isMiddleDown 337
 - isMinimized 227
 - isMousePersistent 582
 - isOnSQLServer 757
 - isOpenOnUniqueIndex 758
 - isPreFilter 149
 - isQueryValid 423
 - isRecordDeleted 758, 865
 - isRemote 167
 - isRemovable 168
 - isResizable 60
 - isRight 390
 - isRightDown 337
 - isShared 666, 759
 - isShiftKeyDown 266, 338
 - isShowDeletedOn 760
 - isSpace 518
 - isSQLServer 79
 - isTable 80, 667
 - isTableCorrupt 582
 - isTargetSelf 149
 - isToolbarShowing 228
 - isValid 760

Index

isValidDir 168
 isValidFile 169
 isVisible 761
 isVisible 228, 818

J
 justification 1005
 justifying text 1001, 1005

K
 key codes 597 - 600
 key fields 669, 773, 886, 1005
 searching 777
 key violations 100, 116, 701, 798
 keyboard
 virtual key codes 262
 keyboard buffers 601
 keyboard constants 969
 keyboard events 261
 event type 261
 handling 228 - 230, 597 - 601, 865
 key code 262
 simulating 266 - 268
 testing keypresses 263 - 266, 269 - 271
 KeyboardState 971
 KeyboardState constants 971
 keyChar 228, 865
 KeyEvent 261 - 262
 KeyEvent 269
 KeyEvent type
 isAltKeyDown 264
 isControlKeyDown 265
 isFromUI 265
 isShiftKeyDown 266
 setAltKeyDown 266
 setChar 267
 setControlKeyDown 267
 setShiftKeyDown 268
 char method 263
 KeyEvent type 261
 vChar method 269 - 270
 vCharCode method 271
 KeyField 1005
 keyNameToChr 518
 keyNameToVKCode 518
 keyPhysical 229 - 230, 865
 keypresses 261 - 262, 597 - 601
 assigning 266 - 269

event type 261
 extended 543
 mouse events and 335, 338, 342
 testing 263 - 266, 269 - 271
 virtual 269 - 271
 virtual characters 269, 528
 virtual codes 262
 virtual characters 508, 518
 keyword
 try 21
 keywords
 assignment/comparison operator 8
 code comments 7
 const 9
 declarations 9
 disableDefault 10
 doDefault 10
 enableDefault 11
 for 11
 forEach 12
 if 13
 iif 14
 loop 14
 method 15
 passEvent 16
 proc 16
 quitLoop 17
 return 18
 scan 18
 switch 19
 type 22
 uses 22
 var 35
 while 36
 killTimer 866

L
 labeling button objects 1001, 1005
 LabelText 1005
 language drivers 432, 574, 610 - 612, 633, 635, 746 - 747
 lastMouseClicked built-in object variable 37
 lastMouseRightClicked built-in object variable 38
 launching
 web browser 610
 leap years 123, 130
 LeftBorder 1005
 letters 284
 adding attachments 285, 291

addressing 284, 286, 288, 290
 clearing contents 287
 deleting attachments 288
 reading 296
 returning content 292 - 293
 sending 294 - 295, 297
 setting content 298 - 299
 type 284
 libraries 552
 closing 272
 creating 273
 executing methods 275
 opening 277
 returning information 551 - 553
 saving 274
 writing 273 - 274
 run-time 551, 553
 uses 23
 library
 data 388
 Library
 object types 272
 system objects 272
 type 272
 types, object 272
 library constants 972
 Library type 274
 close 272
 create 273
 enumSource 273
 isAssigned 276
 methodEdit 276
 open 277
 execMethod method 275
 LibraryScope 972
 line spacing 1006
 line squeeze 1006
 line styles 1006
 records 1002
 constants 973
 ellipses 1006
 frames 1004
 grids 1004
 Line.Color 1005
 Line.LineStyle 1006
 Line.Thickness 1006
 line-drawing constants 972
 LineEnd 972
 LineEnds 1006
 lines

- placing arrows on 1006
- LineStyle 972
 - constants 972
- LineThickness 973
- LineType 973
- linkFromClipboard 375
- linking DDE objects 134 - 138
- linking OLE objects 366, 372 - 375
- linking tables 199, 203 - 207, 211, 215
- List.Count 1006
- List.Selection 1006
- List.Value 1006
- lists
 - adding items 1002 - 1003
 - adding items 1006
 - selecting items 1006
- live query views 753, 756, 761
- ln 355
- load 231, 443, 450
- LoadDestSpec 111
- loading 231, 450
 - alias specifications 476
 - ASCII files 111
 - forms 190, 195, 231, 244 - 246
 - graphic objects 259 - 260
 - scripts 450
- loadProjectAliases 476
- LoadSourceSpec 111
- locate 761, 867
- locateNext 762, 867
- locateNextPattern 763, 868 - 869
- locatePattern 764 - 765, 870
- locatePrior 766, 871
- locatePriorPattern 767, 872
- locating data 761 - 767, 867 - 872
- lock 477, 668, 768
- Locked 1006
- locking 477, 668, 768 - 769, 873
 - directories 153
 - records 720, 769, 784, 796, 873, 902
 - tables 477, 481, 484 - 485, 538 - 539, 668, 687, 731, 768 - 769, 795, 844, 874
 - directories 172
 - tables 1006
- lockRecord 769, 873
- lockStatus 769, 874
- log 355
- logarithms 355
- logging into mail boxes 294 - 295
- logical 278 - 279
 - Logical type
 - logical procedure 278 - 279
 - logical values 278
 - conversions 278 - 279
 - data type 278
 - formatting 568
 - formatting 1004
 - logoff 294
 - logoffDlg 294
 - logon 295
 - logonDlg 295
 - long integers 279, 347
 - bitwise operations 280 - 283
 - conversions 284
 - data type 279
 - formatting 569
 - LongInt 279, 284
 - LongInt type 279
 - LongInt 284
 - bitAND method 280
 - bitlSet method 281
 - bitXOR method 283
 - lookup tables 1006
 - LookupTable 1006
 - LookupType 1006
 - loop keyword 14
 - lower 519
 - lowercase characters 519
 - lTrim 519
- M**
 - Magnification 1006
 - magnification constants 966
 - Mail 284, 973
 - mail boxes 284
 - Mail type 294
 - 295
 - addAddress 284
 - addAttachment 285
 - addressBook 286
 - addressBookTo 286
 - empty 287
 - emptyAddressess 288
 - emptyAttachments 288
 - enumInbox 289
 - getAddress 290
 - getAddressCount 290
 - getAttachment 291
 - getAttachmentCount 291
 - getMessage 292
 - getMessageType 292
 - getSender 293
 - getSubject 293
 - logoff 294
 - logon 295
 - readMessage 296
 - send 297
 - sendDlg 297
 - setMessage 298
 - setMessageType 299
 - setSubject 299
 - MailAddressTypes constants 284, 973
 - mailings 284
 - adding attachments 285, 291
 - addressing 284, 286, 288, 290
 - clearing contents 287
 - deleting attachments 288
 - logging onto systems 294 - 295
 - reading 296
 - returning content 292 - 293
 - sending 294 - 295, 297
 - setting content 298 - 299
 - type 284
 - MailReadOptions constants 973
 - makeDir 169
 - Manager 1006
 - Margins.Bottom 1006
 - Margins.Left 1006
 - Margins.Right 1006
 - Margins.Top 1006
 - MarkerPos 1006
 - master tables 199, 203 - 207, 211
 - match 520
 - matching
 - arrays 58, 139
 - enabling wildcards 456, 475
 - filenames 158 - 160, 162, 170
 - records 761 - 767, 867 - 870, 872
 - specifying case sensitivity 476
 - text strings 503 - 504, 520, 522 - 523, 799 - 800
 - arrays 53 - 54, 57, 59, 63
 - math coprocessors 610 - 611
 - MAX 356
 - maximize boxes 1006
 - MaximizeButton 1006
 - maximizeforms
 - sizing 232
 - maximizing forms 227, 232, 1006
 - Maximum 1006

Index

- maximum rows 78, 82
- maximum values 356, 624, 705, 1006
- memo 300
- Memo type. 300
 - memo 300
 - readFromClipboard 301
 - readFromFile 302
 - readFromRTFFile 304
 - writeToClipboard 303
 - writeToFile 303
 - writeToRTFFile 304
- Memo View 1006 - 1007
- memory 594
- memos 300
 - conversions 300
 - data type 300
 - displaying text 1001
 - editing text 1003
 - moving through 1003
 - reading 301 - 302, 304
 - saving 303 - 304
 - writing 303 - 304
 - aligning text 1001, 1005
 - displaying text 1004
 - editing text 1003
 - printing 1006
- MemorView 1006
- menu 305, 395
- menu bars 305, 324
- menu commands
 - adding. 305 - 306, 308 - 311, 394, 396, 400 - 401
 - choosing 325 - 332
 - counting 312
 - display attributes 313 - 316, 319 - 322
 - event 324
 - grouping 306, 396, 398
 - removing 313, 318
 - wrapping 306
 - display attributes 317
 - user-defined constants 325
- menu constants 325, 974, 982
- menu events 324
 - event type 324
 - handling 233
 - identifying 325 - 326, 330, 332
 - testing 325, 327 - 329, 331 - 332
 - Windows messages 325, 331
 - user 325
- Menu type. 306, 314 - 315, 321 - 322
 - addArray 305
 - addStaticText 307
 - addText 308 - 310
 - contains 311
 - Count 312
 - empty 313
 - getMenuChoiceAttribute 313
 - hasMenuChoiceAttribute 316
 - isAssigned 317
 - remove 318
 - setMenuChoiceAttribute 319 - 320
 - show 323 - 324
 - addBreak method 306
 - Menu type 305
 - menuSetLimit 317
 - removeMenu procedure 318
- menu types 305
- menuAction 233, 875
- menuChoice 330
- MenuChoiceAttributes 974
- MenuCommands 974
- MenuEvent 324, 330
- MenuEvent type
 - datadata method 325
 - id method 325 - 326
 - isFromUI method 327 - 329
 - MenuEvent type 324
 - reason method 331
 - setData method 331
 - setId method 332
 - setReason method 332
- MenuReasons 982
- menus 305, 318, 396
 - adding 306
 - adding items. 305, 308 - 311, 394, 396, 400 - 401
 - command separators 395, 398
 - counting items 312
 - customizing 252
 - deleting items 313, 318
 - displaying . . 306, 318, 396 - 397, 403 - 404, 535
 - displaying items 313 - 316, 319 - 322
 - event handling 875
 - naming 307, 399
 - restoring default 318
 - reusing 1003
 - selecting items 325 - 332
 - types 305
 - displaying 323 - 324
 - displaying items 317
 - pop-up 306, 394 - 401, 403 - 404
 - setting 317
 - two-column 306, 395
- Menus
 - customizing 447 - 448
- menuSetLimit 317
- message 583
- messages
 - status bar 498 - 502, 583
 - system 583 - 586
 - Windows 616 - 617
- method keyword 15
- methodDelete 233, 875 - 876
- methodEdit 234, 276, 451, 877
- methodGet 234, 877
- methods
 - customizing 234
 - deleting 875 - 877
 - executing 275
 - returning . . . 213 - 214, 234, 273 - 274, 553, 877
 - running 694, 849 - 850
 - saving 847
 - writing 234, 845 - 847, 878
 - custom methods, saving 272
 - customizing 272
 - deleting 233
 - derived 6
 - running 561
- methodSet 234, 878
- military time 130
- milliSec 131
- MIN 356
- minimize 235
- minimize boxes 1006
- MinimizeButton 1006
- minimizing forms 227, 235, 256
- minimum values 356, 625, 706, 1006
- minute 131
- mod 357
- Modal 1006
- modal dialog 583, 586
- modal dialog boxes 246, 1006
 - displaying field values 143, 437 - 438
 - opening forms as 246, 535, 1003
 - position 220, 253
 - system 585
 - system messages 584 - 585
 - displaying field values 49 - 50, 64
- modulus 357
- monetary values 70
 - conversions 71 - 72

- types 70
 - month 132
 - mouse 332
 - mouse buttons. . . 332, 610 - 611, 878 - 879, 881 - 882
 - setting 338 - 342
 - testing 334 - 338
 - mouse events. . . 332, 335, 879 - 880, 1006
 - clicking. . . 334 - 342, 878 - 879, 881 - 882
 - current position. . . 333, 336, 339, 341, 343 - 344
 - event type 332
 - handling. . . 235 - 242
 - keypresses and. . . 335, 338, 342
 - testing. . . 334 - 338, 864, 903 - 904
 - MouseActivate 1006
 - mouseClick 878
 - mouseDouble 235, 878
 - mouseDown 236, 879
 - mouseEnter 237, 879
 - MouseEvent 332
 - MouseEvent type 338, 342
 - getMousePosition 333
 - getObjectHit 334
 - isFromUI 335
 - isInside 336
 - isLeftDown 336
 - isMiddleDown 337
 - isRightDown 337
 - setInside 339
 - setLeftDown 339
 - setMiddleDown 340
 - setMousePosition 341
 - setRightDown 341
 - setX 343
 - setY 344
 - x 344
 - y 344
 - isControlKeyDown method. . . 335
 - isShiftKeyDown method 338
 - MouseEvent type 332
 - mouseExit 238, 880
 - mouseMove 239, 880
 - mouseRightDouble 239, 881
 - mouseRightDown 240, 881
 - mouseRightUp 241, 882
 - MouseShapes 982
 - mouseUp 242, 882
 - move constants 983
 - MoveEvent 344
 - MoveEvent type 344
 - getDestination 345
 - reason 346
 - setReason 346 - 348
 - MoveReasons 983
 - moveTo 883
 - form type. . . 243
 - moveToPage 244, 443
 - moveToRecNo 770, 884
 - moveToRecord 692, 771, 884
 - moving among objects 344 - 348, 883
 - moving objects. . . 1002
 - moving through forms 244, 1007
 - moving through tables. . . 39, 698, 702, 725, 736, 749, 770 - 772, 777, 789, 827 - 828, 843, 858, 884 - 885, 889, 899
 - moving through text 1003
 - moy method. . . 133
 - msgAbortRetryIgnore 583
 - msgInfo 584
 - msgQuestion 584
 - msgRetryCancel 585
 - msgStop 585
 - msgYesNoCancel 586
 - multi-objects
 - returning information. . . 1007
 - multiple objects 1002
 - selecting specific. . . 1002 - 1003
 - moving through 1003, 1007 - 1008
 - multi-record objects
 - active column. . . 1002
 - active row 1002
 - expanding 1002
 - layouts 1001
 - returning information. . . 1003
- N
- name 170, 1007
 - naming 255
 - fields 1003
 - files 171
 - forms 190, 223, 255, 1001
 - menus 307, 399
 - tables 541, 673
 - design objects 1007
 - report bands 1007
 - restrictions 37
 - table frames 1005
 - tables 793
 - Native Windows Controls 998
 - natural logarithms 355
 - navigating through forms. . . 244, 1007
 - navigating through objects. . . 344 - 348, 883
 - navigating through tables. . . 39, 698, 702, 725, 736, 749, 770 - 772, 777, 789, 827 - 828, 884 - 885, 889, 899
 - navigating through text 1003
 - Ncols 1006
 - net present values 626, 707
 - network drives 167
 - networks. . . 471, 474, 540, 666, 759
 - newValue 905
 - next 384, 1007
 - nextRecord 772, 885
 - NextTabStop 1007
 - nFields 669, 773, 885
 - nKeyFields 669, 773, 886
 - No messages 586
 - NoEcho 1007
 - notebook
 - currentPage property 1002
 - NumberPages property 1007
 - nRecords. . . 670, 774, 886, 1007
 - Nrows 1007
 - number 347, 357
 - Number type
 - abs 349
 - acos 349
 - asin 350
 - atan 350
 - atan2 351
 - ceil 351
 - cos 352
 - cosh 352
 - exp 353
 - floor 353
 - fraction 353
 - fv 354
 - ln 355
 - log 355
 - MAX 356
 - MIN 356
 - mod 357
 - number 357
 - numVal 358
 - pmt 358
 - pow 359
 - pow10 360
 - pv 360
 - rand 361
 - round 362

Index

- sin 362
- sinh 363
- sqr 363
- tan 364
- tanh 364
- truncate 365
- Number type 347
- number types 70, 279, 347, 486
- NumberPages 1007
- numbers 279, 347
 - 278 - 279
 - absolute values 349
 - binary 280 - 283, 488 - 490
 - comparisons 356
 - conversions 71 - 72, 284, 357 - 358, 490 - 491, 572
 - data types 70, 279, 347
 - dividing 357
 - exponential 353, 359 - 360
 - formatting 569 - 570, 1004
 - fractional part 353
 - random 361
 - rounding 351, 353, 362
 - square roots 363
 - truncating 365
 - conversions 45, 48
 - data types 486
 - floating-point 347
- numVal 358
- O**
- object constants 998
- object types
 - data models 73, 404, 491, 620, 695
 - design 305
 - display managers 438, 448, 689
 - returning 551
 - saving 847
 - system 134, 448, 452, 529, 798
 - add-in forms 42
 - design 825
 - display managers 49, 188
 - system 152
- ObjectPAL
 - basic elements 7
 - calling C routines 33, 35
 - declarations 2 - 3
 - executing code statements 11
 - passing to C procedure 34
- objects
 - borders 1001, 1004
 - bounding boxes 852
 - color settings 857, 891, 1002
 - containers 862, 1002
 - copying 836
 - creating 837
 - default properties 1002
 - deleting 838
 - event handling 865
 - moving 1002
 - moving among 344 - 348, 883
 - multiple 1002
 - position 854, 895
 - resizing 1002
 - returning 213 - 214, 216, 221, 441, 845, 848
 - selecting 221, 254, 1002 - 1003
 - viewing 902
 - borders 1004 - 1005
 - color 1005
 - color settings 1004
 - containers 1007
 - hiding 1005
 - multiple 1003, 1007 - 1008
 - overlapping 831
 - returning 1004
 - scrolling 1004 - 1005
 - UIObject type 825
- OEM characters 507, 521
 - returning 521
 - text files 94
 - returning 507
- oemCode 521
- OLE 367, 372 - 375
- OLE applications
 - defining actions 368 - 370
 - launching 368, 372 - 374
 - returning servers 369, 371
- OLE automation 377, 382, 386 - 387
 - event handling 380 - 381, 385
 - opening servers 384 - 385
 - registry 378
 - type 377
- OLE objects
 - copying 367, 372 - 376
 - editing 368
 - linking 366, 372 - 375
 - reading 367, 372 - 375
 - writing 376
- OLE type
- canLinkFromClipboard 366
- enumServerClassNames 369
- enumVerbs 369 - 370
- getServerName 371
- isLinked 375
- readFromClipboard 375
- updateLinkNow 375
- writeToClipboard 376
- edit method 368
- OleAuto 377
 - attaching objects 378, 381
- OLEAuto type 377
 - attach 377
 - close 378
 - enumAutomationServers 378
 - enumConstants 379
 - enumConstantValues 379
 - enumControls 380
 - enumEvents 380
 - enumMethods 381
 - enumObjects 381
 - enumProperties 382
 - enumServerInfo 382
 - first 383
 - invoke 384
 - next 384
 - open 384
 - openObjectTypeInfo 385
 - openTypeInfo 385
 - registerControl 386
 - unregisterControl 386
 - version 387
- online help 576 - 580
- open 136, 244 - 245, 277, 384, 478, 693, 775, 805
- openAsDialog 246
- opening 81, 136, 244 - 245, 277, 444, 478, 775, 805
 - databases 73, 81, 470
 - DDE conversations 136
 - forms 190, 195, 231, 244 - 246, 548
 - libraries 277
 - reports 439, 443 - 444, 550
 - sessions 476, 478
 - SQL tables 79, 81, 494
 - tables 622, 664, 676, 681, 686, 775
 - text files 802, 805
- openObjectTypeInfo 385
- openTypeInfo 385
- operators 417, 434
 - comparison 8
- Orphan/Widow 1007

- OtherBandName 1007
overlapping objects 831, 892
OverStrike 1007
Overwrite mode 1007
overwriting data 1007
Owner 1007
- ## P
- packing tables 627, 707
page breaks 1001
page headers 1007
page numbers 1007
PageSize 1007
PageTiling 1007
page-tiling constants 983
PageTilingOptions 983
passEvent keyword 16
passing by pointer 34
passing by value 33
passwords 454, 483
 adding 454, 483
 removing 480
 testing 665, 688, 756
pattern constants 984
Pattern.Color 1007
Pattern.Style 1007
patterns 508, 984
 color settings 1007
 defining 1007
 repeating characters 508
PatternStyles 984
PCX 260
pe xiii
 search xxiv
periodic payments 358
PersistView 1007
Pi 965
Picture 1007
PinHorizontal 1007
pinning objects 1002
PinVertical 1007
pixels
 conversions 586, 616, 887, 901
pixelsToTwips 586, 887
play 587
pmt 358
point 391
 data types, screen coordinates 388
Point type
 distance 389
 isAbove 389
 isBelow 390
 isLeft 390
 isRight 390
 point 391
 setX 392
 setXY 392
 setY 393
 x 393
 y 393
pointer 332
 event type 332
 getting position 333, 336, 344, 574, 858
 moving 880
 setting position 339, 341, 343 - 344, 604
 specifying shape 582, 604 - 605
 constants 982
pointers 695
 Answer tables 434
 attaching to tables 753 - 754, 756 - 758, 761, 775
 data models 197, 208 - 209, 721
 records 698, 701, 827 - 828
 SQL tables 497
 text files 803 - 804, 806, 809
 type 695
 attaching to a table 699 - 700
 text files 804
points
 conversions 391
 distance between 389, 833
 returning 389 - 390, 393
 setting 392 - 393
population variance 641 - 642, 716, 718
Pop-up Menu type
 396 - 398
 addArray method 396 - 401, 404
 addText method 400 - 401
 issAssigned 402
 show method 403
 switchMenu 404
Pop-up Menu Type
 addStaticText 399
pop-up menus
 adding items 394, 396, 400 - 401
 building 306, 396 - 397
 command separators 395
 displaying 403
 naming 399
 building 404
 command separators 398
 displaying 404
PopupMenuType
 addArrayaddArray method 394 - 395
 addBaraddBar method 395
 addBreakaddBreak method 396
 addArray method 396
portASCIIFix v
position 806, 1007
PositionalOrder 1007
postAction 247, 887
posting records 701, 776, 779, 782 - 783, 798, 888, 890
postRecord 776, 888
pow 359
pow10 360
PrecedePageHeader 1007
predefined record 244 - 245
predefined record structures 444, 562 - 564, 589
preferences 574, 603
present values 360, 626, 707
Prev 1008
Previous Error dialog box 196
primary indexes 637 - 638, 662 - 663, 669, 712 - 714, 773, 886
print 446
print constants 984 - 985, 987
PrintColor 984
PrintDuplex 985
PrinterDocument 1008
printerGetInfo 587
printerGetOptions 588
PrinterOptionInfo 589
PrinterOrientation 985
printers 548, 573, 587 - 591, 602
printerSetCurrent 590
printerSetOptions 591
PrinterSizes 985
PrinterType 987
printing 446, 573, 588, 591, 602
 ReportPrintInfo 444
 reports 444, 446, 1001
 design documents 1008
 reports 1005, 1008
 text 1006
PrintOnIstPage 1008
PrintQuality 987
PrintSources 987
priorRecord 777, 889
private directories 170, 175 - 176

Index

privDir 170
 Problems tables 100, 117
 proc keyword 16
 procedures 553
 processors 610 - 611
 profile strings 594, 619
 ProgID 1008
 Program Identifier 1008
 programming
 basic elements 7
 calling C routines 32 - 33, 35
 declarations 2 - 3, 9
 editing methods 10
 Project Viewer 592
 projectViewerClose 592
 projectViewerIsOpen 592
 projectViewerOpen 592
 properties 212, 1001
 alphabetical list 1001
 default 1002
 returning 201 - 202, 217, 220, 442, 472 - 473,
 735, 848, 855 - 856
 setting 210, 254, 484, 896
 protecting 454, 480, 483
 protecting data 665, 688, 756
 prototypes 836
 returning 220
 writing 254
 writing 2
 publishTo 445
 publishTo method 445
 PublishToFilter 988
 publishToRTF 445
 publishToWord97 445
 publishToWP9 445
 push buttons 890
 pushButton 890
 pv 360

Q

QBE 404
 type 404
 qbeCheckType 988
 qbeCheckType constants 406 - 407, 414
 qbeRowOperation 988
 qLocate 777
 queries 404, 423, 492
 building statements 198, 420 - 421, 424 - 427
 changing expressions 428 - 429

checking fields and rows 406 - 408, 414
 enabling auxiliary tables 408
 returning field structures 409
 running 410 - 412, 416, 423, 432 - 434, 753, 756,
 761
 saving statements 435 - 436
 setting criteria 415, 419, 428, 431
 SQL tables 493 - 498
 type 404
 query 404, 424 - 425
 query constants 988
 Query type 404
 appendRow 405
 appendTable 405
 checkField 406
 checkRow 407
 clearCheck 408
 createAuxTables 408
 createQBEString 409
 enumFieldStruct 409
 executeQBE 410 - 412
 getAnswerFieldOrder 413
 getAnswerName 413
 getAnswerSortOrder 414
 getCheck 414
 getCriteria 415
 getQueryRestartOption 416
 getRowID 416
 getRowNo 417
 getRowOp 417
 getTableID 418
 getTableNo 418
 hasCriteria 419
 insertRow 420
 insertTable 420
 isAssigned 421
 isCreateAuxTables 421
 isEmpty 422
 isExecuteQBELocal 423
 isQueryValid 423
 query 424 - 425
 readFromFile 426
 readFromString 426 - 427
 removeCriteria 428
 removeRow 428
 removeTable 429
 setAnswerFieldOrder 429
 setAnswerName 430
 setAnswerSortOrder 430
 setCriteria 431

setLanguageDriver 432
 setQueryRestartOptions 432 - 433
 setRowOp 434
 writeQBE 435 - 436
 WantInMemoryTCursor method 434
 QueryRestartOptions 988
 quitLoop keyword 17

R

radio buttons 1001
 rand 361
 random numbers 361
 Range 1008
 range of values 661, 680, 689, 747, 787, 856, 897,
 1008
 raster constants 989
 RasterOperation 1008
 RasterOperations 989
 readChars 807
 readEnvironmentString 593
 readFromClipboard 66, 259, 301, 375, 521
 readFromFile 67, 260, 302, 426, 494
 readFromRTFFile 304
 readFromString 426 - 427, 495 - 496
 reading mail 296
 readLine 807 - 808
 readMessage 296
 ReadOnly 1008
 readProfileString 594
 reason 145, 150, 331, 346, 499
 recNo 778
 RecNo 1008
 Record 436
 record buffers 717, 744, 750, 838
 Record type 436
 viewview method 437 - 438
 records 111, 436, 763, 767, 872
 111
 activating 692
 adding 696, 750 - 752, 755, 859 - 861, 863, 1001
 appending 85, 87, 108
 appending data 84 - 88, 94, 107, 109 - 110
 binding 828 - 830
 blank 1001
 color settings 1002
 data type 436
 displaying 437 - 438, 698, 725, 749, 770 - 772,
 777, 827 - 828, 843, 858, 884 - 885, 889, 899

- editing . . . 703, 717, 723, 725, 754, 833, 838, 840, 843, 863, 1003
- grouping 1002
- locking 720, 769, 784, 796, 873, 902
- matching . . . 761 - 762, 764 - 766, 867, 870 - 871
- posting 701, 776, 779, 782 - 783, 888, 890
- removing from tables . . . 627, 682, 707, 719, 724, 758, 760, 839, 841 - 842, 865, 1002
- reporting status 779, 890
- returning 670, 774, 778, 781, 886
- setting current 771, 1002
- subtracting 542 - 543, 684
- testing existence 755, 863
- undeleting 795, 901
- adding 1005
- displaying 789
- grouping 1008
- removing from tables 788
- returning 1007
- subtracting 791
- recordStatus 779, 890
- redrawing screens 193 - 194, 539
- referential integrity 652, 731 - 732
- Refresh 1008
- RefreshOption 1008
- registerControl 386
- registry 378
 - deleting keys 534
 - returning information 549 - 550, 575, 596
 - setting values 606
 - key type constants 989
- RegistryKeyType 989
- RegistryKeyType constants . 534, 549 - 550, 575, 596, 606
- reIndex 672, 780
- reIndexAll 672, 781
- remainder 357
- remote drives 167
- removable drives 168
- remove 60, 318, 819
- removeAlias 479
- removeAllItems 61
- removeAllPasswords 480
- removeButton 819
- removeCriteria 428
- RemoveGroupRepeats 1008
- removeItem 62, 142
- removeMenu 318
- removePassword 480
- removeProjectAlias 481
- removeRow 428
- removeTable 429
- removing 60, 287, 318, 627, 707, 819
 - aliases 479, 481
 - array items 140, 142
 - BLOBs 65
 - columns in tables 1002
 - design objects 838
 - directories 155
 - empty fields 1002
 - files 154
 - indexes from tables 645, 722
 - menu commands 313, 318
 - menus 318
 - methods 875 - 877
 - passwords 480
 - records from tables 542 - 543, 719, 724, 758, 760, 839, 841 - 842, 865, 1002
 - tables 77, 208, 537, 643
 - Toolbars 224, 228, 817 - 819
 - array items 54, 56, 60 - 62
 - methods 233
 - records from tables 788
- rename 673
- renaming 673
 - files 171
 - tables 541, 673
- RepeatHeader 1008
- replacetItem 63
- replacing field values 63
- Report 438
- report bands
 - drilldowns 1003
 - grouping records 1002
 - grouping 1007
 - grouping records 1008
 - naming 1007
 - position 1007
- report constants 989 - 990
- Report Design windows 441, 443, 447
- report headings 1005, 1007
- Report type 438, 441 - 442, 445
 - attach 439
 - close 440
 - currentPage 440
 - load 443
 - moveToPage 443
 - print 446
 - setMenu 447 - 448
 - design method 441
 - open method 444
 - publishTo 445
 - publishToRTF 445
 - publishToWord97 445
 - run method 447
- Report windows 440 - 441, 447
- ReportOpenInfo 444
- ReportOrientation 989
- reportPrintInfo 446
- ReportPrintPanel 990
- ReportPrintRestart 990
- reports 438, 444
 - changing pages 443
 - closing 440
 - current page 440
 - custom menus 447 - 448
 - deleting empty fields 1002
 - designing 1003
 - getting information 441 - 442
 - margins 1006
 - multi-record 1002
 - opening 439, 443 - 444, 550
 - printing 444, 446, 1001
 - specifying groups 1002
 - type 438
 - viewing 447
 - designing 441
 - displaying 1007
 - margins 1006
 - multi-record 1001 - 1003, 1007
 - printing 1005, 1008
 - publishing 445
 - specifying groups 1008
- Required 1008
- required elements 2
- reserved keywords 37
- reserved words 11
- resizeable arrays 50, 138
- resourceInfo 594
- resources 594
- restructure 674 - 675
- RestructureOperations
 - restructure constants 990
- restructuring 674 - 675
- restructuring 719
- restructuring tables 541 - 543, 724, 841 - 842
- resync 891
- resynchronizing 208 - 209
- resynchronizing tables 891

Index

- retry 539
 - Retry messages 583, 585
 - retryPeriod 481
 - return keyword 18
 - return path 561
 - return values 35
 - returning data types 45
 - returning file information 157, 164, 166, 180, 183
 - returning form objects 213 - 214, 216, 221
 - returning OLE servers 369, 371
 - returning report objects 441 - 442
 - rgb 891
 - rich text formats 301, 303
 - RightBorder 1008
 - rights and privileges 153, 684, 793
 - fields 1003
 - files 153, 164, 174
 - tables 653 - 654, 676, 681, 684, 733 - 734
 - fields 738
 - tables 793
 - riteToClipboard 260
 - riteToFile 261 - 262
 - rollBackTransaction 82
 - rootkey 989
 - round 362
 - rounding 351, 353, 362
 - RowHeight 1008
 - RowNo 1008
 - rows 405
 - returning 416 - 417
 - counting 1007
 - setting CurrentRow property 1002
 - rTrim 522
 - run 248, 447, 451
 - runExpert 595
 - running 248, 434, 447
 - DDE 136
 - DOS 559
 - forms 196, 217 - 218, 247 - 248, 256, 608
 - library methods 275
 - methods 694, 849 - 850
 - queries 410 - 412, 416, 423, 432 - 433
 - reports 447
 - scripts 449, 451, 560, 587
 - SQL statements 492, 497
 - code statements 11
 - forms 189
 - methods 561
 - run-time errors 552, 555, 558
 - run-time libraries 551 - 553
 - run-time properties 1007
- S
- sample variance 639 - 640, 715
 - save 248
 - saveCFG 482
 - saveProjectAliases 482
 - saveStyleSheet 249
 - saving 248
 - aliases 482
 - BLOBs 69 - 70
 - data 701, 776, 779, 798, 888, 890
 - forms 195, 248 - 249
 - graphic objects 260 - 262
 - libraries 274
 - memos 303 - 304
 - methods 847
 - object types 847
 - query statements 435 - 436
 - text files 801
 - custom method 272
 - scan keyword 18
 - screen 193 - 194
 - screen coordinates 586
 - conversions 391, 586, 616, 887, 901
 - distance between 389
 - returning points 389 - 390, 393
 - setting 392 - 393
 - screen refreshes 539
 - Script 448
 - Script type 448
 - attach 449
 - create 449
 - load 450
 - methodEdit 451
 - run 451
 - scripts 449, 560
 - creating 449, 560
 - loading 450
 - running 449, 451, 560, 587
 - type 448
 - Scroll 1008
 - scroll bars 1005
 - scrolling objects 1004
 - search 522 - 523
 - searches 522 - 523
 - arrays 53 - 54, 139
 - enabling wildcards 456, 475
 - files 158 - 160, 162, 170
 - key fields 777
 - specifying case sensitivity 476
 - text strings 522 - 523
 - arrays 57 - 59, 63
 - searchRegistry 596
 - second 133
 - secondary indexes 650 - 651, 662 - 663, 669, 729 - 730, 773, 886
 - security 153, 684, 793
 - adding passwords 483, 665, 688, 756
 - field rights 1003
 - files 153, 164, 174
 - tables 653 - 654, 676, 681, 684, 733 - 734
 - adding passwords 454
 - field rights 738
 - tables 793
 - SeeMouseMove 1008
 - Select 1008
 - selectCurrentTool 249
 - SelectedText 1008
 - selecting items in lists 1006
 - selecting menu 325 - 329, 331 - 332
 - selecting menu commands 325, 330 - 331
 - selecting objects 1002 - 1003
 - self built-in object variable 38
 - send 297
 - sendDlg 297
 - sending mail 294 - 295, 297
 - sendKeys 597 - 600
 - sendKeysActionID 601
 - sendToBack 892
 - seqNo 781
 - SeqNo 1008
 - Session type 452, 472 - 473, 483
 - addAlias 452 - 453
 - addProjectAlias 454 - 455
 - blankAsZero 457
 - close 457
 - enumAliasLoginInfo 458
 - enumAliasNames 459 - 460
 - enumDatabaseTables 461
 - enumDriverCapabilities 461 - 465
 - enumDriverInfo 466
 - enumDriverNames 467
 - enumDriverTopics 467
 - enumEngineInfo 468
 - enumFolder 469
 - enumOpenDatabases 470
 - enumUsers 471
 - getAliasPath 472

getNetUserName	474	setFlyAwayControl	784	setting properties	210, 254, 484
isAssigned	476	setGenFilter	677 - 678, 785 - 786, 893 - 894	setTitle	255
isBlankZero	476	setIcon	252	setUserLevel	607
loadProjectAliases	476	setId	41 - 43, 332	setVChar	269
lock	477	setIndex	679	setVCharCode	269
open	478	setInside	339	setWorkingDir	177 - 178
removeAlias	479	setItem	137 - 138	setX	343, 392
removeProjectAlias	481	setKeyviol	116	setXY	392
retryPeriod	481	setLanguageDriver	432	setY	344, 393
saveCFG	482	setLeftDown	339	shelling	559
saveProjectAliases	482	setMaxRows	82	shortName	179
setAliasPath	483	setMenu	252, 447 - 448	show	256, 323 - 324, 403, 821 - 824
setAliasProperty	484	setMenuChoiceAttribute	319 - 320	ShowAllColumns	1008
setRetryPeriod	484	setMenuChoiceAttributeById	321 - 322	showapplicationbar	822
unlock	485	setMessage	298	showDeleted	682, 788
addPassword method/procedure	454	setMessageType	299	ShowGrid	1008
sessions	452	setMiddleDown	340	showToolBar	256
closing	457	setMousePosition	341	Shrinkable	1008
opening	476, 478	setMouseScreenPosition	604	sin	362
setAliasPassword	483	setMouseShape	604	sine	350, 362 - 363
setAliasPath	483	setMouseShapeFromFile	605	sInfo	xxv
setAliasProperty	484	setNewValue	905 - 906	sinh	363
setAltKeyDown	266	setPosition	253, 809, 820, 895	size	64, 68, 142, 180, 523, 809
setAnswerFieldOrder	429	setPrivDir	175 - 176	Size	1008
setAnswerName	430	setProblems	117	sizeEx	524
setAnswerSortOrder	430	setProperty	896	SizeToFit	1008
setAppend	112	setProtoProperty	254	skip	789, 899
setBatchOff	782	setQueryRestartOptions	432 - 433	sleep	608
setBatchOn	782 - 783	setRange	680, 787, 897	small integers	347
setChar	267	setReadOnly	681	bitwise operations	488 - 490
setCompileWithDebug	250 - 251	setReason	145, 151, 332, 346 - 348, 499	conversions	490 - 491
setControlKeyDown	267, 338	setRegistryValue	606	formatting	570
setCriteria	431	setRetryPeriod	484	data type	486
setData	331	setRightDown	341	smallInt	486, 491
setDefaultPrinterStyleSheet	602	setRowOp	434	SmallInt type	
setDefaultScreenStyleSheet	603	setSelectedObjects	254	bitXOR method	490
setDesktopPreference	603	setShiftKeyDown	268, 342	int	490
setDest	112	setSize	63	smallInt	491
setDestCharSet	113	setSource	117	bitIsSet method	488
setDestDelimitedFields	114	setSourceCharSet	118	SnapToGrid	1008
setDestDelimiter	114	setSourceDelimitedFields	119	sort	683
setDestFieldNamesFromFirst	115	setSourceDelimiter	119	sort order	414, 430
setDestSeparator	116	setSourceFieldNamesFromFirst	120	sorting data	542, 683, 790, 1003
setDir	171	setSourceRange	121	SortOrder	1008
setDirLock	172	setSourceSeparator	122	sortTo	790
setDrive	173	setState	820	sound	65, 530, 609
setErrorCode	150	setStatusValue	500	special fields	
setExclusive	676	setStyleSheet	255	constants	990
setFieldValue	784	setSubject	299	SpecialField	1009
setFileAccessRights	174	setTimer	898	SpecialFieldTypes	990

Index

- splitFullFileName 180 - 181
- Spreadsheet Export dialog box 93
- Spreadsheet Import dialog box 87, 109 - 110
- spreadsheets 93, 109 - 110
 - 109 - 110
 - reading 89 - 90, 97, 104 - 105, 115, 120 - 121
 - writing 105, 121
 - writing to 89 - 90, 93, 97, 104, 115, 120
- SQL 491
- SQL 497
- SQL statements 491
 - creating 493 - 496
 - running 492
 - type 491
 - writing 497 - 498
- SQL tables 491
 - detecting 757
 - error handling 556, 558
 - opening 79, 81, 494
 - queries 492 - 498
 - transactions 74 - 76, 82 - 83
- SQL type 491
 - executeSQLexecuteSQL method/procedure 493, 497 - 498
 - isAssigned 493
 - readFromFile 494
 - readFromString 495 - 496
 - writeSQL 497 - 498
 - executeSQL method/procedure 492, 494 - 497
- sqrt 363
- square roots 363
- SquareTabs 1009
- standard deviation 639, 641, 715 - 716
- StandardMenu 1009
- StandardToolBar 1009
- Start 1009
- StartPageNumbers 1009
- startup directories 182
- startUpDir 182
- startwebbrowser 610
- state constants 997
- status bars 498
 - messages 498 - 502, 583
 - type 498
- status constants 991
- StatusEvent 498
- StatusEvent type
 - reason 499
 - setReason 499
 - setStatusValue 500
 - statusValue 500 - 502
- StatusEvent typeStatusEvent type 498
- StatusReasons 991
- statusValue 500 - 502
- stop messages 585
- streams 798
- string 502, 525
- String 508
- String 503 - 504, 518
- String ty 505, 524
- String type 505, 524
 - 521
 - chr 507
 - format 509 - 515
 - isEmpty 516
 - readFromClipboard 521
 - rTrimrTrim method 522
 - size 523
 - string 525
 - stringVal 525
 - subStr 525
 - toAnsi 526
 - toOEM 526 - 528
 - upperupper method 527
 - vkCodeToKeyNamevkCodeToKeyName procedure 528
 - writeToClipboard 528 - 529
 - breakApart 506
 - chrOEM procedure 507
 - fill procedure 508
 - isSpace method 518
 - keyNameToVKCode procedure 518
 - lower method 519
 - lTrim method 519
 - sizeEx 524
- String typeString type 502
- strings 394, 400 - 401, 456, 475
 - ANSI characters 526
 - case 476
 - case sensitivity 474, 519, 527
 - conversions 519, 525 - 527, 571 - 573
 - data type 502
 - dates 123
 - empty 502, 516
 - formatting 509 - 515
 - getting size 523
 - matching patterns 503 - 504, 520, 522 - 523, 799 - 800
 - menus 305, 307 - 310, 399
- OEM ch 521
- OEM characters 521
- queries 198, 426 - 427
- reading 521
- returning substrings 525
- searching 522 - 523
- status bar messages 500 - 502, 583
- trimming 519, 522
- virtual key codes 528
- white space characters 519, 522
- writing 528 - 529
- ANSI characters 505, 507
- case 516 - 517
- comparisons 516 - 517
- dates 126
- empty 518
- getting size 524
- OEM characters 507
- returning substrings 506
- toOEM method 526
- virtual key codes 508, 518
- white space characters 518, 524
- writing 508
- structural elements 7
 - control 11
- strVal 525
- Style 1009
- style sheets 222, 249, 255, 573, 602 - 603
- subject built-in object variable 38
- substr 525
- substrings 525
 - 520, 799 - 800
 - matching 503 - 504, 522 - 523
 - returning 525
 - returning 506
- subtract 684, 791
- subtracting records 542 - 543, 684, 791
- summary constants 920
- summary values 642, 717
- SummaryModifier 1009
- switch keyword 19
- switchIndex 792, 900
- switchMenu 404
- synchronizing tables 208 - 209, 891
- syntax 2 - 3
- sysInfo 610 - 612
- System 529
- System 539, 562 - 564
- System menus 1002
- system objects 1002

clock	533	enumWindowNames	554	msgQuestions	584
information	610 - 611, 619	errorClear	555	msgRetryCancelmsgRetryCancel procedure	585
resources	594	errorCode	555	msgStop	585
types	134, 448, 452, 529, 798	errorHasErrorCode	556	msgYesNoCancelmsgYesNoCancel procedure	586
types	152	errorHasNativeErrorCode	556	pixelsToTwips	586
System type	529	errorLog	557	play	587
	553, 558, 567, 571 - 572	errorNativeCode	558	printerGetInfo	587
beep	530	errorPop	558	printerGetOptions	588
close	530	errorShow	559	PrinterOptionInfo	589
compileInformation	531	errorTrapOnWarnings	559	printerSetCurrent	590
constantNameToValue	532	ex		printerSetOptions	591
constantValueToName	532	executeString	560	projectViewerClose	592
cpuClockTime	533	exit	561	projectViewerIsOpen	592
debug	533	fileBrowser	562	projectViewerOpen	592
deleteRegistryKey	534	formatAdd	565	readEnvironmentString	593
desktopMenu	535	formatDelete	565	readProfileString	594
dlgAdd	535	formatExist	566	resourceInfo	594
dlgCopy	536	formatGetSpec	566	runExpert	595
dlgCreate	536	formatSetCurrencyDefault	567	searchRegistry	596
dlgDelete	537	formatSetDateDefault	567	sendKeys	597 - 600
dlgEmpty	537	formatSetLongIntDefault	569	sendKeysActionID	601
dlgNetDrivers	537	formatSetNumberDefault	569	setDefaultPrinterStyleSheet	602
dlgNetLocks	538	formatSetSmallIntDefault	570	setDefaultScreenStyleSheet	603
dlgNetRetrydlgNetRetry procedure	539	formatSetTimeDefault	570	setDesktopPreference	603
dlgNetSetLocks	539	formatStringToDateTimeformatStringToDateTime method	572	setMousePosition	604
dlgNetSystem	540	formatStringToTimeformatStringToTime method	573	setMouseShape	604
dlgNetUserName	540	getDefaultPrinterStyleSheet	573	setMouseShapeFromFile	605
dlgNetWho	540	getDefaultScreenStyleSheet	573	setRegistryValue	606
dlgRename	541	getDesktopPreference	574	setUserLevel	607
dlgRestructure	541	getLanguageDrivers	574	sleep	608
dlgSort	542	getMouseScreenPosition	574	sound	609
dlgSubtractdlgSubtract procedure	542	getRegistryValue	575	sy	
dlgTableInfodlgTableInfo procedure	543	helpOnHelp	576	tracerClear	613
enableExtendedCharactersenableExtendedCharacters procedure	543	helpQuit	577	tracerHide	613
enumDesktopWindowHandles	544	helpSetIndex	577	tracerOff	614
enumDesktopWindowNames	544	helpShowContext	578	tracerOn	614
enumEnvironmentStrings	545	helpShowIndex	579	tracerSave	615
enumExperts	546	helpShowTopic	580	tracerShow	615
enumFonts	547	helpShowTopicInKeywordTable	580	tracerToTop	615
enumFormats	547	isErrorTrapOnWarnings	581	tracerWrite	615
enumFormNames	548	isMousePersistent	582	twipsToPixels	616
enumPrinters	548	isTableCorrupt	582	version	616
enumRegistryKeys	549	LanguageDrivers	612	winGetMessageID	616
		lock type values	538	winPostMessage	617
enumRegistryValueNamesenumRegistryValueNames method	550	message	583	winSendMessage	617
enumReportNames	550	msgAbortRetryIgnoremsgAbortRetryIgnore procedure	583	writeEnvironmentString	618
enumRTLConstants	551	msgInfo	584	writeProfileString	619
enumWindowHandles	554			enumRTLErrors method	552
				fail	561

Index

T

tab order 831, 883, 892, 1007
 TabHeight 1009
 Table 620
 Table 665, 688
 table data 1005
 Table Export dialog box 85, 92
 table frames
 deleting columns 1002
 manipulating columns 1002
 constants 991
 displaying data 1004
 grids 1004
 headers 1001, 1005
 manipulating columns 1005 - 1006
 returning information 1003, 1007
 scrolling 1005
 Table type 620
 653 - 654, 676, 681
 add 620 - 621
 attach 622
 cAverage 623
 cCount 623
 cMax 624
 cMin 625
 cNpv 626
 copy 628
 create 628 - 636
 createIndex 637 - 638
 cSamStd 639
 cSamVar 640
 cStd 641
 cSum 642
 cVar 642
 delete 643
 dropGenFilter 644
 dropIndex 645
 empty 646
 enumFieldNames 647
 enumIndexStruct 650 - 651
 enumRefIntStruct 652
 familyRights 655
 field translations for tables 636
 fieldName 656
 fieldType 657 - 658
 getRange 661
 index 662 - 663
 isAssigned 664
 isEmpty 665

isShared 666
 isTable 667
 language drivers for dBASE tables 635
 language drivers for tables 633
 lock 668
 nFields 669
 nKeyFields 669
 nRecords 670
 reIndex 672
 reIndexAll 672
 rename 673
 restructure 674 - 675
 setGenFilter 659 - 660, 677 - 678
 setIndex 679
 setRange 680
 showDeleted 682
 sort 683
 subtract 684
 tableRights 684
 type 685
 unAttach 686
 unlock 687
 usesIndexes 688
 using ranges and filters 689
 compact method 627
 enumFieldStruct 648 - 649
 fieldNo 656
 Table windows 689, 1004
 closing 691
 opening 693
 type 689
 TableFrameStyles 991
 tableName 793
 TableName 1009
 tableRights 684, 793
 tables 620
 542 - 543
 access rights. 653 - 654, 676, 681, 684, 733 - 734
 adding 197, 203, 212, 535 - 536, 628 - 636
 attaching pointers 699 - 700, 753 - 754, 756 - 758, 761, 775
 closing 691, 705
 copying 536, 620 - 621, 628, 708
 creating 674 - 675
 deleting 77, 208, 537, 643
 displaying 693
 emptying 537, 646, 665, 755, 863
 family members 77, 655, 737
 filtering 644, 659 - 660, 677 - 678, 689, 721, 745, 785 - 786

inserting records . 696, 750 - 752, 755, 859 - 861, 863, 1001
 linking 199, 203 - 207, 211, 215
 locking 477, 481, 484 - 485, 538 - 539, 668, 687, 731, 768 - 769, 795, 844, 874
 moving . 698, 771 - 772, 777, 789, 827 - 828, 884 - 885, 889, 899
 moving through 39, 702, 725, 736, 749, 843, 858
 naming 541, 673
 opening 622, 664, 676, 681, 686, 775
 packing 627, 707
 password protecting 454, 480, 483, 665, 688, 756
 restructuring 541, 674 - 675, 719, 724, 839, 841 - 842
 returning information 77, 212, 215, 461, 647, 653 - 654, 666, 685, 726 - 728, 733 - 734, 759, 794, 843
 sorting data 542, 683, 790
 synchronizing 208 - 209, 891
 testing for existence 80, 667
 type 620
 access rights 793
 entering data 1007
 grids 1004
 locking 1006
 naming 793
 returning information 648 - 649, 793
 sorting data 1003
 TableView 689
 TableView type 689
 690
 close 691
 isAssigned 692
 moveToRecord 692
 open 693
 wait 694
 TabsAcross 1009
 TabsOnTop 1009
 TabStop 1009
 tan 364
 tangent 350 - 351, 364
 tanh 364
 targets 147, 149
 TCursor 695
 TCursor 756, 758, 760, 788
 TCursor type
 724, 733 - 734, 744, 758, 793
 add 696
 aliasName 697
 atLastatLast method 698
 attach 699 - 700

attachToKeyViol	701	isAssigned	754	skip method	789
bot	702	isEdit	754	subtract	791
cancelEdit	703	isEmpty	755	switchIndex	792
cAverage	703	isInMemoryTCursor	756	tableName	793
cCount	704	isOnSQLServer	757	unDeleteRecord method	795
close	705	isShared	759	updateRecord method	798
cMax	705	isValid	760	template	853
cMin	706	isView	761	testing	263
cNpv	707	lo		testing	338
copy	708	locateLocate method	761	testing error conditions	144 - 145
copyFromArray	709	locateNextLocateNext method	762	testing keypresses	264 - 266, 269 - 271
copyRecord	710	locatePatternLocatePattern method	764 - 765	testing menu events	325, 327 - 329, 331 - 332
createIndex	712 - 714	locatePriorLocatePrior method	766	testing mouse events	334 - 337, 864, 903 - 904
cSamStd	715	lock	768	text	300
cSamVar	715	lockRecord	769	aligning	1001
cStd	716	lockStatus	769	copying	301 - 304, 521, 528 - 529
cSum	717	moveToRecNo	770	data type	300
currRecord	717	nextRecordnextRecord method	772	displaying	1001
cVar	718	nFields	773	finding456, 474 - 475, 503 - 504, 520, 522 - 523, 799 - 800	
didFlyAway	720	nKeyFields	773	moving through	1003
dmAttach	721	nRecords	774	saving	303 - 304
dropGenFilter	721	open	775	aligning	1005
dropIndex	722	postRecord	776	color settings	1004
edit	723	qLocate	777	displaying	1004
endEdit	725	recNo	778	editing	1003
enumFieldNames	726	recordStatus	779	printing	1006
enumFieldNamesInIndex	726	reIndex	780	Text	1009
enumFieldStruct	727 - 728	reIndexAll	781	text	522 - 523
enumIndexStruct	729 - 730	seqNo	781	text boxes	1003, 1005
enumLocks	731	setBatchOffsetBatchOff method	782	text buffers	801
enumRefIntStruct	731 - 732	setBatchOnsetBatchOn method	782 - 783	text constants	991 - 992
enumTableProperties	735	setFlyAwayControl	784	text files	798
eot	736	setGenFilter	785 - 786	closing	801
familyRights	737	setRange	787	creating	802
fieldName	737	setValue	784	exporting and importing	83, 89, 123
fieldNo	738	sortTo	790	opening	802, 805
fieldSize	739	type	794	position	803 - 804, 806, 809
fieldType	740	unlock	795	reading	84 - 88, 107 - 108, 111, 807 - 808
fieldUnits2	741	unlockRecord	796	returning size	809
getGenFilter	745	updateupdate method	797	saving	801
getIndexName	746	atFirst method	698	type	798
getLanguageDriver	746	compact method	707	writing to	85, 90 - 91, 810
getLanguageDriverDesc	747	copyToArray	711	character sets	101
getRange	747	deleteRecord method	719	position	804
homehome method	749	end method	725	text streams	798
initRecord	750	fieldValue	742 - 743	reading	807 - 808
insertAfterRecord	750	handle	748	type	798
insertBeforeRecord	751	moveToRecord method	771	writing	810
insertRecord	752	priorRecord method	777	text strings	502
instantiateView	753	security	738		

Index

.....	474, 502	size	809	show	821, 823 - 824
ANSI characters	526	writeLine	810	unAttach	822
case sensitivity	474, 476, 519, 527	writeString	810	show	822
conversions	519, 525 - 527, 571 - 573	home	804	ToolBarBitmap	992
data type	502	TextStream typeTextStream type	798	ToolBarButtonType	996
formatting	509 - 515	ThickFrame	1009	ToolBarClusterID	997
getting size	523	Thickness	1009	toolbars	
matching patterns 503 - 504, 520, 522 - 523, 799 - 800		TIF files	260	listing names	816
menus	305, 307 - 310, 399 - 401	time	183, 811 - 812	Toolbars	224, 228, 249, 256, 812
OEM characters	521	time	572	adding	815
queries	198, 426 - 427	time stamps	1004	displaying	228, 256, 818, 821
reading	521	Time type		emptying	815
returning substrings	525	time	811 - 812	inserting tools	813, 819, 836
searching	456, 475	Time typeTime type	811	position	816, 820
status bar messages	500 - 502, 583	time values	811	removing	224, 228, 817 - 819
trimming	519, 522	573	selecting tools	249
virtual	528	conversions	127, 811 - 812	shape	817, 820
white space characters	519, 522	data types	811	type	812, 814, 822
writing	528 - 529	formatting	567, 570, 811, 1004	Tooltip	1009
ANSI characters	505, 507	returning	130 - 131, 133	TooltipText	1009
case sensitivity	517	data types	126	TopBorder	1009
comparisons	516 - 517	formatting	1004	TopLine	1009
empty	518	timer objects	823	totalDiskSpace	183
getting size	524	starting	898	totalDiskSpaceEX	184
OEM characters	507	stopping	866	Touched	1009
returning substrings	506	type	823	Tracer window	613 - 615
virtual	518	TimerEvent	823	tracerClear	613
virtual key codes	508, 518	Title	1009	tracerHide	613
white space characters	518, 524	toAnsi	526	tracerOff	614
writing	508	today	125 - 126	tracerOn	614
text types	300, 502	toHex	48	tracerSave	615
TextAlignment	991	toOEM	526	tracerShow	615
TextDesignSizing		ToolBar	812	tracerToTop	615
text constants	992	ToolBar	819	tracerWrite	615
TextSpacing	992	ToolBar button type	996	TrackBar	998
TextStream	798	ToolBar clusters	997	transactionActive	83
TextStream type		ToolBar states	997	transactions	74 - 76, 82 - 83
advMatchadvMatch method	799 - 800	ToolBar type	812	TransferData	123
close	801	addButton	813	transferring data	83, 89, 123
commit	801	attach	814	translations	
create	802	create	815	field name	636
end	803	empty	815	Translucent	1009
eof	804	enumToolBarNames	816	trapping errors	559, 581
isAssigned	804	getPosition	816	trimming	519, 522
open	805	getState	817	truncate	365
position	806	hide	817	truncated numbers	365
readChars	807	isVisibleisVisible method	818	try keyword	21
readLine	807 - 808	removeButton	819	tSourceFieldNamesFromFirst	v
setPosition	809	setPosition	820	twips	
		setState	820	conversions	586, 616, 887, 901

- twipsToPixels 616, 901
 - two-column menus 306, 395 - 396
 - type 685, 794
 - type keyword 22
 - typefaces 1004
 - types 657 - 658, 685, 740, 794, 1003
 - 261
 - Array type 50
 - data 50, 65, 70, 123, 138, 258, 278 - 279, 300, 347, 436, 502, 811
 - event 144, 146, 324, 332, 344, 498, 823, 904
 - mail 284
 - object 73, 134, 188, 305, 404, 438, 448, 452, 491, 529, 551, 620, 689, 695, 798
 - saving 847
 - Toolbar 812
 - AddInForm type 42
 - AnyType 43
 - data 43, 126, 486
 - event 39
 - object 42, 49, 152, 825
 - typeActionEvent 39
 - types of constants 907
- U**
- UIObject 825
 - UIObject 865, 877
 - UIObject type 825
 - 831 - 832, 841 - 842, 852
 - attach 828 - 830
 - cancelEdit 833
 - convertPointWithRespectTo 833
 - copyFromArray 834
 - copyToArray 835
 - copyToToolbar 836
 - create 837
 - currRecord 838
 - delete 838
 - dropGenFilter 840
 - edit 840
 - endEdit 843
 - enumFieldNames 843
 - enumLocks 844
 - enumObjectNames 845
 - enumSource 845 - 846
 - enumSourceToFile 847
 - enumUIClasses 847
 - enumUIObjectNames 848
 - enumUIObjectProperties 848
 - execMethod 849 - 850
 - forceRefresh 851
 - getGenFilter 852
 - getHTMLTemplate 853
 - getPosition 854
 - getProperty 855
 - getPropertyAsString 856
 - getRange 856
 - getRGB 857
 - hasMouse 858
 - insertAfterRecord 859
 - insertBeforeRecord 860
 - insertRecord 861
 - isAssigned 862
 - isContainerValid 862
 - isEdit 863
 - isEmpty 863
 - isLastMouseClickedValid 864
 - isLastMouseRightClickedValid 864
 - keyChar 865
 - keyPhysical 865
 - killTimer 866
 - lo
 - locateloate method 867
 - locatePatternlocatePattern method 870
 - lockRecord 873
 - lockStatus 874
 - methodDelete 875 - 876
 - methodEdit 877
 - methodSet 878
 - mouseClick 878
 - mouseDouble 878
 - mouseDown 879
 - mouseEnter 879
 - mouseExit 880
 - mouseMove 880
 - mouseRightDouble 881
 - mouseRightDown 881
 - mouseRightUp 882
 - mouseUp 882
 - moveTo 883
 - moveToRecNo 884
 - nFields 885
 - nKeyFields 886
 - nRecords 886
 - pixelsToTwips 887
 - postAction 887
 - postRecord 888
 - pushButton 890
 - records 828 - 830
 - recordStatus 890
 - resync 891
 - rgb 891
 - sendToBack 892
 - setGenFilter 893 - 894
 - setPosition 895
 - setProperty 896
 - setRange 897
 - setTimer 898
 - skip method 899
 - switchIndex 900
 - twipsToPixels 901
 - unlockRecord 902
 - view 902
 - wasLastClicked 903
 - wasLastRightClicked 903 - 904
 - atFirst method 827
 - atLast method 828
 - priorRecord method 889
 - onDeleteRecord method 901
 - UIObject type
 - types, object 825
 - UIObjectTypes 998
 - unAssign 48
 - unAttach 686, 822
 - UncheckedValue 1009
 - onDeleteRecord 795, 901
 - undoing changes 703, 717, 795, 833, 838, 901
 - unique indexes 637 - 638, 662 - 663, 712 - 714, 758
 - unlock 485, 687, 795
 - unlocking 485, 687, 795 - 796, 902
 - directories 153
 - records 796, 902
 - tables 485, 687, 795
 - unlockRecord 796, 902
 - unprintable keyboard 264
 - unprintable keyboard characters 263
 - unProtect 688
 - unregisterControl 386
 - update 797
 - UpdateLink 999
 - updateLinkNow 375
 - updateRecord 798
 - updating 193 - 194, 375, 797 - 798
 - data 193 - 194, 208, 539, 782 - 783, 797 - 798
 - OLE links 375
 - upper 527
 - uppercase characters 527
 - user-defined constants 40, 144, 189, 247, 325, 690, 826

Index

UserError 144
 UserErrorMax 144
 UserMenu 325
 UserMenuMax 325
 users 471, 474, 540, 607
 Advanced level 576
 Beginner level 576
 uses block 23
 uses keyword 22
 usesIndexes 688

V

valid dates 123 - 124
 valid time formats 811
 validity checks 760, 1003
 Value 1009
 value constants 965, 999
 ValueEvent 904
 ValueEvent type
 newValue 905
 setNewValue 905 - 906
 ValueReasons 999
 values 43, 742 - 743, 843
 208, 749, 782 - 783, 858
 absolute 349
 assigning . 141, 165, 224, 276, 317, 402, 692, 784,
 797, 804, 862
 averaging 623, 703
 blank 44, 457, 476
 changing 904 - 906
 Currency 70
 default 1002
 displaying . 143, 200, 437 - 438, 698, 771 - 772,
 777, 827 - 828, 851, 884 - 885, 889, 899, 1002
 event type 904
 logical 278 - 279, 568
 ranges 661, 680, 689, 747, 787, 856, 897
 returning . . 623 - 625, 642, 704 - 706, 717, 760,
 777
 searching 761 - 767, 867 - 872
 truncating 365
 updating 193 - 194, 539, 797 - 798
 assigning 46, 48, 55
 blank 47
 changing 49 - 50
 color settings 1004
 displaying . . 49 - 50, 64, 744, 789
 logical 1004
 replacing 63

 returning 742 - 743
 templates 1007
 var keyword 35
 variance 639 - 642, 715 - 716, 718
 vChar 269 - 270
 vCharCode 271
 version 387, 616
 version numbers 616
 vertical Toolbars 817
 VerticalScrollBar 1009
 view 64, 437 - 438, 902
 virtual key codes . . . 262, 269 - 271, 508, 518, 528
 Visible 1009
 vkCodeToKeyName 528

W

wait 256, 694
 wantInMemoryTCursor 434, 497
 warning beeps 530, 609
 warnings 559, 581, 583 - 586
 wasLastClicked 903
 wasLastRightClicked 903 - 904
 while keyword 36
 whitespace 518
 whitespace characters 524
 text 519, 522
 text strings 524
 WideScrollBar 1009
 Width 1010
 wildcards 456, 475
 WIN.INI 594, 619
 window constants 999
 windowClientHandle 257
 windowHandle 257
 WindowHandles 554
 windows 257, 544
 447
 design 195, 226, 231, 441, 443
 Form . 191 - 192, 220, 223, 228, 243 - 246, 248,
 253, 1001
 Report 440
 returning information 257, 544
 Table 689, 691, 693, 1004
 design 1004
 Table 1004
 Windows messages 616 - 617
 windowsDir 184
 windowsSystemDir 185
 WindowStyles 999

winGetMessageID 616
 winPostMessage 617
 winSendMessage 617
 WordWrap 1010
 working directories 177 - 178, 185
 workingDir 185
 wrapping menu commands . . . 306
 writeEnvironmentString 618
 writeLine 810
 writeProfileString 619
 writeQBE 435 - 436
 writeSQL 497 - 498
 writeString 810
 writeText 258
 writeToClipboard 69, 303, 376, 528 - 529
 writeToFile 69 - 70, 303
 writeToRTFFile 304

X

x 344, 393
 Xseparation 1010

Y

y 393
 year 134
 Yes messages 586
 Yseparation 1010