Encyclopaedia of Classes

Volume 2

VERSION 2018.0.01

jade

Copyright©2018 Jade Software Corporation Limited. All rights reserved.

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2018 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the JADE **ReadMe.txt** file.

Contents

Contents ..... **iii**

Before You Begin ..... **xxxiii**

Who Should Read this Encyclopaedia xxxiii
What's Included in this Encyclopaedia xxxiii
Related Documentation xxxiii
Conventions xxxiv

Chapter 1 System Classes **35**

JadeSkinApplication Class 44
JadeSkinApplication Properties 44
myFormSkins 44
myControlSkins 44
JadeSkinApplication Method 45
updateSkinTimeStamp 45
JadeSkinArea Class 46
JadeSkinArea Class Constants 48
JadeSkinArea Properties 48
backColor 48
imgBorderBottomLeft 49
imgBorderBottomRight 49
imgBorderBottomStrip 49
imgBorderLeftStrip 50
imgBorderRightStrip 50
imgBorderTopLeft 50
imgBorderTopRight 50
imgBorderTopStrip 50
imgInner 50
innerIsBrush 51
JadeSkinCategory Class 52
JadeSkinControl Class and Subclasses 53
JadeSkinControl Class Constants 53
JadeSkinControl Properties 54
applyCondition 54
borderStyle 55
focusBackColor 55
focusForeColor 57
fontBold 58
fontItalic 59
fontName 59
fontSize 60
fontStrikethru 60
fontUnderline 61
foreColor 61
foreColorDisabled 61
JadeSkinBaseControl Class 62
JadeSkinBrowseButtons Class 62
JadeSkinBrowseButtons Properties 62
myFirstButton 62
myLastButton 62
myNextButton 63
myPriorButton 63
JadeSkinButton Class 63
JadeSkinButton Properties 63
createRegionFromMask 64
myButtonDisabled 64
myButtonDown 64
myButtonFocus 64

Contents iv

myButtonFocusDown 64
myButtonRollOver 65
myButtonRollUnder 65
myButtonUp 65
JadeSkinCheckBox Class 65
JadeSkinCheckBox Properties 66
myFalseImage 66
myTrueImage 66
JadeSkinComboBox Class 66
JadeSkinComboBox Properties 67
buttonRightOffset 67
imgComboButtonDownRollOver 67
myComboButton 68
myListBoxSkin 68
mySimpleComboTextBoxSkin 68
JadeSkinFolder Class 68
JadeSkinFolder Properties 68
myTabsButton 69
tabActiveColor 69
tabHeight 69
tabInactiveColor 70
JadeSkinFrame Class 70
JadeSkinGroupBox Class 70
JadeSkinGroupBox Class Constants 71
JadeSkinGroupBox Properties 71
captionPosition 71
captionPositionLeftOffset 72
captionPositionTopOffset 72
myLabelSkin 72
JadeSkinHScroll Class 73
JadeSkinHScroll Properties 73
myLeftButton 73
myRightButton 73
JadeSkinJadeDockBar Class 74
JadeSkinJadeDockBase Class 74
JadeSkinJadeDockBase Properties 74
myHorizontalGripBar 74
myHorizontalResizeBar 74
myVerticalGripBar 75
myVerticalResizeBar 75
JadeSkinJadeDockContainer Class 75
JadeSkinJadeEditMask Class 75
JadeSkinJadeMask Class 75
JadeSkinJadeMask Property 76
myButtonSkin 76
JadeSkinJadeRichText Class 76
JadeSkinLabel Class 76
JadeSkinListBox Class 76
JadeSkinListBox Properties 77
alternatingRowBackColor 77
alternatingRowBackColorCount 78
imgPictureClosed 78
imgPictureLeaf 79
imgPictureMinus 79
imgPictureOpen 79
imgPicturePlus 79
selectionColor 79
selectionColorText 80
JadeSkinOleControl Class 80
JadeSkinOptionButton Class 80
JadeSkinOptionButton Properties 80

Contents v

myFalseImage ... 81
myTrueImage ... 81
JadeSkinPicture Class ... 81
JadeSkinProgressBar Class ... 81
JadeSkinProgressBar Property ... 81
myProgressImage ... 82
JadeSkinScrollBar Class ... 82
JadeSkinScrollBar Properties ... 82
imgHighLightBrush ... 82
myThumbTrack ... 83
myThumbTrackDisabled ... 83
myThumbTrackDown ... 83
myThumbTrackRollOver ... 83
JadeSkinSheet Class ... 83
JadeSkinSheet Property ... 84
myTabButton ... 84
JadeSkinStatusLine Class ... 84
JadeSkinTable Class ... 84
JadeSkinTable Properties ... 84
alternatingRowBackColor ... 85
alternatingRowBackColorCount ... 86
fixed3D ... 86
fixedColumnsBackColor ... 86
fixedRowColorHasPrecedence ... 87
fixedRowsBackColor ... 87
myCheckBoxSkin ... 87
selectionColor ... 88
selectionColorText ... 88
tabActiveColor ... 88
tabInactiveColor ... 88
JadeSkinTextBox Class ... 88
JadeSkinTextBox Properties ... 89
hintBackColor ... 89
hintForeColor ... 89
JadeSkinVScroll Class ... 90
JadeSkinVScroll Properties ... 90
myBottomButton ... 90
myTopButton ... 90
JadeSkinEntity Class ... 91
JadeSkinEntity Class Constant ... 91
JadeSkinEntity Properties ... 91
description ... 91
myOwners ... 92
mySkinRoot ... 92
name ... 92
JadeSkinForm Class ... 93
JadeSkinForm Properties ... 94
captionActiveForeColor ... 96
captionFontBold ... 96
captionFontItalic ... 97
captionFontName ... 97
captionFontSize ... 98
captionInactiveForeColor ... 98
captionLeft ... 98
captionTop ... 98
centerCaption ... 99
drawMenuSelectionFlat ... 99
imgInactiveBorderBottomLeft ... 99
imgInactiveBorderBottomRight ... 99
imgInactiveBorderBottomStrip ... 100
imgInactiveBorderLeftStrip ... 100

Contents vi

imgInactiveBorderRightStrip 100
imgInactiveBorderTopLeft 100
imgInactiveBorderTopRight 100
imgInactiveBorderTopStrip 101
imgMenuLeft 101
imgMenuRight 101
imgMenuStrip 101
menuBackColor 101
menuBackColorSelected 102
menuFontBold 102
menuFontItalic 102
menuFontName 102
menuFontSize 102
menuForeColor 102
menuForeColorDisabled 103
menuForeColorSelected 103
menuLeftPosition 103
menuTopPosition 103
myChildMinimizeBtn 103
myChildRestoreBtn 104
myChildTerminateBtn 104
myMaximizeBtn 104
myMaximizedBtn 104
myMenuSkin 105
myMinimizeBtn 105
myTerminateBtn 105
showMenuLineAlways 105
transparentColorForButtons 105
useMenuLineSkinForMenus 106
JadeSkinMenu Class 107
JadeSkinMenu Properties 107
backColorSelected 108
borderStyle 108
drawMenuSelectionFlat 109
fontBold 109
fontItalic 109
fontName 110
fontSize 110
foreColor 110
foreColorDisabled 110
foreColorSelected 111
imgCheckMark 111
imgRightArrow 111
imgSeparator 111
lineHeight 112
pixelsAfterCheckMark 112
pixelsAfterPicture 112
pixelsBeforeAccelerator 112
pixelsBeforeCheckMark 113
pixelsBeforeRightArrow 113
JadeSkinRoot Class 114
JadeSkinRoot Properties 114
allApplicationSkins 114
allControlSkins 114
allFormSkins 115
allMenuSkins 115
allSimpleButtonSkins 115
allSkinCategories 115
allSkinEntities 115
allWindowStateImages 116
JadeSkinSimpleButton Class 117

Contents vii

JadeSkinSimpleButton Properties 117
imgDisabled 117
imgDown 118
imgRollOver 118
imgUp 118
JadeSkinWindow Class 119
JadeSkinWindow Properties 119
myImageMask 119
mySkinCategory 119
JadeSkinWindowStateImage Class 121
JadeSkinWindowStateImage Properties 121
foreColor 121
isImageMask 121
JadeSOAPException Class 123
JadeSSLContext Class 124
JadeSSLContext Class Constants 125
JadeSSLContext Properties 125
caFile 126
caPath 126
cipherList 126
methodType 126
verifyDepth 127
verifyRemoteCertificate 127
x509 127
JadeSSLContext Methods 127
getActiveCipher 128
getPeerCertificate 128
JadeTableCell Class 129
JadeTableCell Properties 129
column 130
comboIndex 130
hyperLink 131
mergeCells 131
picture 133
row 133
sheet 134
text 134
JadeTableCell Methods 134
delete 134
getCellWidth 135
positionLeft 135
positionTop 135
setPictureDescription 136
JadeTableColumn Class 137
JadeTableColumn Properties 137
column 138
maxColumnWidth 138
sheet 139
sortAsc 139
sortCased 139
sortOrder 139
sortType 140
visible 140
width 140
widthPercent 141
JadeTableColumn Methods 141
delete 141
findObject 142
findString 142
restoreAutoSize 142
JadeTableElement Class 143

Contents viii

JadeTableElement Properties 144
alignment 145
backColor 145
cellControl 145
comboList 146
decimals 146
editMask 146
enabled 146
fontBold 147
fontItalic 147
fontName 147
fontSize 147
fontStrikethru 148
fontUnderline 148
foreColor 148
gridBottom 148
gridRight 149
inputType 150
itemObject 150
marginBottom 150
marginLeft 151
marginRight 151
marginTop 152
maxLength 152
partialTextIndication 152
selected 153
wordWrap 153
JadeTableRow Class 154
JadeTableRow Properties 154
height 154
row 155
sheet 155
visible 155
JadeTableRow Methods 156
delete 156
findObject 156
findString 156
restoreAutoSize 157
JadeTableSheet Class 158
JadeTableSheet Properties 158
alternatingRowBackColor 159
alternatingRowBackColorCount 160
caption 161
column 161
columns 161
currentRowImage 162
displaySorting 162
extendedColumn 163
fixed3D 164
fixedColumns 164
fixedRows 164
gridColor 164
gridLines 165
leftColumn 165
myTable 165
pixelHorzScrollIncrement 165
pixelVertScrollIncrement 166
row 166
rows 167
scrollBars 167
scrollHorzPos 167

scrollMode ... 168
scrollVertPos ... 168
sheet ... 169
showCurrentRowImage ... 169
showPartialTextBubbleHelp ... 169
tabInitialPosition ... 170
tabOffEnds ... 171
topRow ... 171
visible ... 171
widthPercentStyle ... 171
JadeTableSheet Methods ... 172
accessCell ... 173
accessColumn ... 173
accessRow ... 174
addItem ... 174
addItemAt ... 174
clear ... 175
delete ... 175
findColumnObject ... 175
findObject ... 175
findRowObject ... 176
findString ... 176
getCollection ... 177
getCellFromPosition ... 177
insertColumn ... 177
moveColumn ... 177
moveRow ... 178
positionCollection ... 178
refreshEntries ... 178
removeItem ... 179
resort ... 179
restoreAutoSize ... 179
selectedCount ... 179
selectedNext ... 179
setCollectionObject ... 180
JadeTcpIpProxy Class ... 181
Proxy Communication Code Examples ... 181
Considerations when Implementing JadeTcpIpProxy Class Features ... 183
JadeTcpIpProxy Class Constants ... 184
JadeTcpIpProxy Properties ... 184
browserType ... 184
domain ... 185
host ... 185
password ... 185
port ... 186
proxyType ... 186
userName ... 187
JadeTcpIpProxy Method ... 188
connect ... 188
JadeTestCase Class ... 189
JadeTestCase Methods ... 189
assert ... 190
assertEquals ... 190
assertEqualsMsg ... 190
assertFalse ... 191
assertFalseMsg ... 191
assertNotNull ... 192
assertNotNullMsg ... 192
assertNull ... 192
assertNullMsg ... 193
assertTrue ... 193

Contents x

assertTrueMsg ... 194
expectedException ... 194
info ... 195
JadeTestListenerIF Interface ... 196
JadeTestListenerIF Interface Callback Method Signatures ... 197
finish ... 197
message ... 198
methodSuccess ... 198
start ... 198
testFailure ... 199
testSkipped ... 199
testSuccess ... 199
JadeTestRunner Class ... 201
JadeTestRunner Class Methods ... 201
runTests ... 201
setDebugOnAssert ... 202
setDebugOnException ... 202
setDebugOnUnexpectedException ... 202
setLogCallStack ... 202
setTestListener ... 202
JadeTransactionTrace Class ... 204
JadeTransactionTrace Class Constants ... 204
JadeTransactionTrace Properties ... 205
myProcess ... 205
startTime ... 205
status ... 205
stopTime ... 205
tranId ... 206
JadeTransactionTrace Methods ... 206
clear ... 206
getEntry ... 206
getEntryCount ... 207
JadeUserCollClass Class ... 208
JadeUserCollClass Methods ... 208
addExternalKey ... 208
addMemberKey ... 209
clearKeys ... 210
endKeys ... 210
setLength ... 210
setMembership ... 210
Using JadeUserCollClass Collections ... 211
JadeWebService Class ... 212
JadeWebService Class Constants ... 212
JadeWebService Methods ... 212
isNilItem ... 213
setAnyPropType ... 213
setError ... 214
JadeWebServiceConsumer Class ... 215
JadeWebServiceConsumer Class Constants ... 215
JadeWebServiceConsumer Properties ... 215
characterConversionException ... 216
handleCharConversionException ... 216
logStatistics ... 217
password ... 217
proxyHostName ... 217
proxyPassword ... 217
proxyUsername ... 217
soapHeaders ... 218
soapRequest ... 218
soapResponse ... 218
timeout ... 218

Contents xi

unknownHeaders 218
userName 219
workerApp 219
JadeWebServiceConsumer Methods 219
addHttpHeader 220
getEndpointURL 220
getHttpHeader 221
getHttpHeaderClient 221
getHttpHeaderServer 221
getLastStatistics 222
getTimeouts 224
invoke 224
invokeAsync 225
invokeAsyncWithVerb 226
invokeWithVerb 227
processReply 228
reset 228
sendRequest 228
setEndpointURL 228
setTimeouts 229
JadeWebServiceProvider Class 230
JadeWebServiceProvider Properties 230
deleteTransientReturnType 230
incomingMessage 230
rawXML 230
unknownHeaders 231
JadeWebServiceProvider Methods 231
createVirtualDirectoryFile 231
deleteVirtualDirectoryFile 232
getLastStatistics 232
getServerVariable 234
initialize 236
isVDFilePresent 236
processMessage 236
processRequest 236
processRequestPostHeaders 237
reply 237
JadeWebServiceSoapHeader Class 238
JadeWebServiceSoapHeader Properties 238
actor 238
didUnderstand 238
mustUnderstand 239
JadeWebServiceUnknownHeader Class 240
JadeWebServiceUnknownHeader Properties 240
headerXML 240
webService 240
JadeWebSocket Class 241
JadeWebSocket Property 241
id 241
JadeWebSocket Methods 241
onClose 241
onMsg 242
onOpen 242
send 242
sendText 242
JadeWebSocketServer Class 243
JadeWebSocketServer Methods 243
getWebSocket 243
run 243
stop 244
JadeX509Certificate Class 245

JadeX509Certificate Properties 245
endDate 245
issuer 245
purpose 245
startDate 246
subject 246
JadeX509Certificate Methods 246
readCertificateDataFromFile 246
readPrivateKeyDataFromFile 246
JadeXMLAttribute Class 248
JadeXMLAttribute Properties 248
element 248
localName 248
name 248
namespaceURI 248
value 249
JadeXMLAttribute Method 249
namespacePrefix 249
JadeXMLCDATA Class 250
JadeXMLCharacterData Class 251
JadeXMLCharacterData Property 251
data 251
JadeXMLComment Class 252
JadeXMLDocument Class 253
JadeXMLDocument Properties 253
docType 253
endOfLine 254
indentString 254
keepWhitespace 254
outputDeclaration 254
rootElement 254
JadeXMLDocument Methods 254
addComment 255
addCommentObject 255
addDocumentType 256
addDocumentTypeObject 256
addElement 256
addElementNS 257
addElementObject 257
addElementObjectNS 257
addProcessingInstruction 257
addProcessingInstructionObject 257
findElementByNameNS 257
findElementByTagName 258
findElementsByNameNS 258
findElementsByTagName 258
getElementByTagName 259
getElementByTagNameNS 259
getElementsByTagName 259
getElementsByTagNameNS 259
parseFile 260
parseString 260
writeToFile 260
JadeXMLDocumentParser Class 261
JadeXMLDocumentParser Methods 261
comment 261
parseDocumentFile 261
parseDocumentString 262
processingInstruction 262
setClassMapping 262
startCDATA 262

Contents xiii

JadeXMLDocumentType Class 263
JadeXMLDocumentType Properties 263
internalSubset 263
name 263
publicId 263
systemId 263
JadeXMLElement Class 264
JadeXMLElement Properties 264
attributes 264
localName 264
namespaceURI 265
tagName 265
textData 265
JadeXMLElement Methods 265
addAttribute 266
addAttributeNS 266
addAttributeObject 266
addAttributeObjectNS 267
addCDATA 267
addCDATAObject 267
addComment 267
addCommentObject 267
addElement 267
addElementNS 268
addElementObject 268
addElementObjectNS 268
addProcessingInstruction 268
addProcessingInstructionObject 268
addText 268
addTextObject 269
findAllElementsByNameNS 269
findAllElementsByTagName 269
getAllElementsByTagName 269
getAllElementsByTagNameNS 270
getAttributeByName 270
getAttributeByNameNS 270
getElementByTagName 270
getElementByTagNameNS 270
getElementsByTagName 271
getElementsByTagNameNS 271
namespacePrefix 271
parentElement 271
setText 271
text 271
JadeXMLException Class 272
JadeXMLException Class Constants 272
JadeXMLException Properties 273
columnNumber 273
fileName 273
lineNumber 273
JadeXMLNode Class 274
JadeXMLNode Properties 274
childNodes 274
document 274
parentNode 274
JadeXMLNode Methods 275
copyAfter 275
copyAsChildOf 275
copyBefore 275
descendsFrom 275
moveAfter 276

Contents xiv

moveAsChildOf 276
moveBefore 276
remove 276
writeToString 276
JadeXMLParser Class 278
JadeXMLParser Methods 278
characters 279
columnNumber 279
comment 279
endCDATA 279
endDTD 280
endElement 280
fileName 280
getAttribute 280
getAttributeValueByName 281
getAttributeValueByNameNS 281
lineNumber 282
parseFile 282
parseString 282
processingInstruction 283
startCDATA 283
startDTD 283
startElement 284
JadeXMLProcessingInstruction Class 285
JadeXMLProcessingInstruction Properties 285
data 285
target 285
JadeXMLText Class 286
List Class 287
List Methods 287
clear 287
copy 287
purge 288
Locale Class 289
Locale Class Constants 289
Locale Properties 289
cloneOf 289
clones 290
forms 290
languageId 290
schema 290
translatableStrings 290
Locale Methods 290
getAllTranslatableStrings 291
getForms 291
getStringValue 291
getTranslatableStringLocal 291
getTranslatableStrings 291
getTranslatableStringsByNum 291
hasClones 291
isClone 292
makeLocaleName 292
LocaleFormat Class 293
LocaleFormat Property 293
schema 293
LocaleFullInfo Class 294
LocaleFullInfo Class Constants 294
LocaleFullInfo Properties 294
currencyInfo 295
dateInfo 295
defaultCodePage 295

Contents XV

defaultCountryCode 295
defaultLanguageId 295
listSeparator 296
measurementSystem 296
nativeDigits 296
numericInfo 296
timeInfo 297
LocaleNameInfo Class 298
LocaleNameInfo Properties 298
abbreviatedCountryName 298
abbreviatedLangName 299
countryCode 299
englishCountryName 299
englishLangName 299
languageId 299
localeId 299
localizedCountryName 300
localizedLangName 300
nativeCountryName 300
nativeLangName 300
Lock Class 301
Lock Class Constants 301
Lock Properties 301
duration 302
elapsedTime 302
kind 302
lockedBy 303
requestedBy 303
requestTime 304
type 304
waitTime 304
Lock Method 305
target 305
LockArray Class 306
LockContentionInfo Class 307
LockContentionInfo Properties 307
maxWaitTime 307
totalContentions 307
totalWaitTime 307
LockContentionInfo Method 308
target 308
Example of Displaying Lock Contention Information 308
LockException Class 310
LockException Properties 310
lockDuration 310
lockTimeout 311
lockType 312
retryCount 312
targetLockedBy 312
LockException Methods 312
lockTarget 313
retryLock 313
showDialog 314
MemberKeyDictionary Class 315
MemberKeyDictionary Methods 315
add 315
includes 316
indexNear 316
indexNear64 317
purge 317
remove 317

MenuItem Class 318
MenuItem Class Constants 319
MenuItem Properties 319
allChildren 320
caption 320
checked 321
children 321
description 322
disableReason 322
enabled 322
form 322
helpContextId 323
helpKeyword 323
index 324
name 324
picture 324
securityLevelEnabled 325
securityLevelVisible 325
userObject 325
visible 326
webFileName 326
MenuItem Methods 326
getLevel 327
getMenuItem 327
loadMenu 327
loadSubMenu 328
setEventMapping 328
setEventMappingEx 329
setShortCutKey 330
MenuItem Events 330
click 331
select 331
MergeIterator Class 332
MergeIterator Property 333
ignoreDuplicates 333
MergeIterator Methods 333
addCollection 334
back 334
current 334
getCollectionAt 335
getCollectionCount 335
getCurrentCollection 335
getCurrentKey 335
getCurrentKeys 335
isValid 336
next 336
removeCollection 337
reset 337
startAtObject 337
startKeyGeq 337
startKeyGtr 338
startKeyLeq 338
startKeyLss 338
MethodCallDesc Class 339
MethodCallDesc Properties 339
invocationMode 339
method 340
position 340
MethodCallDesc Methods 340
getName 340
getReceiver 340

logSelf 341
MultiMediaType Class 342
MultiMediaType Property 342
usePresentationFileSystem 342
NamedPipe Class 343
NamedPipe Property 343
serverName 344
NamedPipe Methods 344
close 344
closeAsynch 345
getMaxMessageSize 345
listen 345
listenAsynch 346
open 346
openAsynch 347
readBinary 347
readBinaryAsynch 347
writeBinary 348
writeBinaryAsynch 348
Node Class 350
Node Class Constants 350
Node Properties 351
accessPatterns 351
name 352
osID 352
processes 352
system 352
userExitCode 352
Node Methods 353
beginIndividualRequestsLogging 356
beginSample 357
clearMethodCache 358
createExternalProcess 358
downloadCount 360
endIndividualRequestsLogging 360
endSample 360
getAppServerGroupName 361
getCacheSizes 361
getCacheSizes64 361
getCharacterSize 361
getCommandLine 362
getComputerName 362
getDefaultLCID 362
getEnvironmentVariable 363
getExecuteFlagValue 363
getIniFileName 363
getJadeInstallDirectory 364
getJadeHomeDirectory 364
getJadeWorkDirectory 364
getLCIDFromCharacterSet 364
getLineDelimiter 365
getLocks 365
getMutexCounts 366
getNotes 367
getObjectCaches 367
getOSDetails 369
getOSPlatform 371
getProfileString 372
getProgramDataDirectory 373
getQueuedLocks 374
getRequestStats 375

getRpcServerStatistics 376
getTempPath 377
getUserDataDirectory 378
isApplicationServer 378
isCacheCoherencyEnabled 378
isReadOnlySchema 379
isReadOnlySystemSchema 379
isServerNode 379
isService 379
logObjectCaches 379
logRequestStatistics 380
logUserCommand 381
networkAddress 382
nodeRole 382
nodeType 382
osProcessId 383
processDump 383
setCacheSizes 383
setCacheSizes64 384
setExecuteFlagValue 384
setProfileString 385
wbemListClasses 386
wbemListInstanceNames 387
wbemQueryQualifiers 388
wbemRetrieveData 390
NormalException Class 394
Notification Class 396
Notification Properties 396
elapsedTime 396
eventType 396
featureNumber 397
isInterface 397
requestedBy 397
requestTime 397
responseType 397
serialNumber 398
typeNumber 398
userTag 398
Notification Methods 398
subscriber 398
target 399
NotificationArray Class 400
NotificationException Class 401
NotificationException Method 401
notificationTarget 401
NumberFormat Class 402
NumberFormat Class Constants 402
NumberFormat Properties 402
decimalPlaces 403
decimalSeparator 403
groupings 403
negativeFormat 403
negativeSign 404
positiveSign 404
showLeadingZeros 404
thousandSeparator 404
NumberFormat Method 404
defineNumberFormat 404
Object Class 406
Object Methods 406
autoPartitionIndex 410

beginClassNotification 410
theClass 411
transients 412
eventType 412
responseType 412
eventTag 412
beginClassNotificationForIF 413
theInterface 413
beginClassesNotification 414
theClass 415
includeSubclasses 415
transients 416
eventType 416
responseType 416
eventTag 416
beginClassesNotificationForIF 417
theInterface 418
beginNotification 418
theObj 419
eventType 419
responseType 419
eventTag 420
Example of Beginning Notifications 420
beginNotificationForIF 421
theInterface 422
beginTimer 422
beginTimerForIF 423
causeEvent 424
changeObjectVolatility 425
class 426
cloneSelf 426
cloneSelfAs 426
copySelf 426
copySelfAs 427
creationTime 427
deletePropertyValue 427
display 427
edition 428
endClassNotification 428
theClass 428
transients 428
eventType 428
endClassNotificationForIF 429
theInterface 429
endClassesNotification 430
theClass 430
includeSubclasses 430
transients 430
eventType 430
endClassesNotificationForIF 431
theInterface 431
endNotification 432
theObj 432
eventType 432
endNotificationForIF 432
theInterface 433
endNotificationForSubscriber 433
endTimer 434
endTimerForIF 434
exclusiveLock 434
getClassForObject 434

Contents

getClassNumberForObject 435
getInstanceIdForObject 435
getInstanceIdForObject64 435
getLockCallStack 435
getLockStatus 436
getModifiedBy 436
getName 436
getObjectStringForObject 436
getObjectVolatility 437
getOidString 437
getOidStringForObject 438
getOwnerForObject 438
getPropertyValue 438
getTimerStatus 439
getTimerStatusForIF 439
getUpdateTranID 440
hasMembers 440
inspect 440
inspectModal 440
invokeIOMethod 441
invokeMethod 442
isImportedObject 443
isKindOf 444
isLockedByMe 444
isObjectFrozen 444
isObjectNonSharedTransient 444
isObjectPersistent 444
isObjectSharedTransient 444
isObjectStable 444
isObjectTransient 445
isObjectVolatile 445
isSharedTransient 445
isSystemObject 445
isTransient 445
jadeReportWriterCheck 445
jadeReportWriterDisplay 446
latestEdition 446
lock 446
makeObjectFrozen 447
makeObjectStable 448
makeObjectVolatile 448
moveToPartition 448
reserveLock 449
respondsTo 449
resynch 449
resynchObject 449
sdeCauseEvent 450
sdsCauseEvent 451
sendMsg 453
sendMsgWithIOParams 453
sendMsgWithParams 454
sendTypeMsg 454
sendTypeMsgWithIOParams 455
sendTypeMsgWithParams 455
setPartitionID 456
setPartitionIndex 456
setPropertyValue 456
sharedLock 457
sysNotification 457
    eventType 457
    theObject 458

Contents xxi

eventTag 458
timerEvent 458
tryGetPropertyValue 458
tryLock 459
unlock 460
updateLock 460
updateObjectEdition 460
userNotification 461
eventType 461
theObject 461
eventTag 461
userInfo 461
version 462
ObjectArray Class 463
ObjectArray Method 464
addAll 464
ObjectByObjectDict Class 465
ObjectLongNameDict Class 466
ObjMethodCallDesc Class 467
ObjMethodCallDesc Property 467
interfaceMethod 467
ObjMethodCallDesc Method 467
getReceiver 467
ObjectSet Class 468
ODBCException Class 469
ODBCException Properties 469
nativeError 469
state 469
ODBCException Method 469
showDialog 470
OleObject Class 471
OleObject Properties 471
compressed 471
fullName 471
oleData 472
shortName 472
OleObject Methods 472
copy 472
getData 473
isServerRegistered 473
setData 473
PointArray Class 474
PrimMethodCallDesc Class 475
PrimMethodCallDesc Property 475
primNo 475
PrimMethodCallDesc Method 475
getReceiver 475
Printer Class 476
Defining Your JADE Report Layouts 477
Layering Print Output 478
Printer Class Constants 479
Printer Properties 480
autoPaging 481
bottomOfPage 481
collate 481
copies 481
documentType 482
drawFillColor 484
drawFillStyle 485
drawFontBold 485
drawFontItalic 485

drawFontName 486
drawFontSize 486
drawFontStrikethru 486
drawFontUnderline 486
drawStyle 487
drawTextAlign 487
drawTextCharRotation 488
drawTextRotation 488
drawWidth 488
duplex 489
footerFrame 489
headerFrame 489
leftMargin 490
orientation 490
pageBorderWidth 491
pageNumber 491
paperSource 491
printPreview 492
printPreviewAllowPrint 493
printPreviewAllowSelect 493
printPreviewReduce 493
retainCMDValues 493
rightMargin 494
suppressDialog 494
title 494
topOfPage 494
Printer Methods 495
abort 497
centreFrame 497
close 497
drawArc 498
drawChord 499
drawEllipse 499
drawFilledRectangle 500
drawGrid 501
drawLine 502
drawPie 502
drawRectangle 503
drawRoundRectangle 504
drawSolidRectangle 505
drawTextAt 505
drawTextIn 506
drawTextSize 507
drawTextSizeIn 507
frameFits 508
getAllPaperSources 508
getAllPrinterPaperSources 509
getAllPrinters 509
getDefaultDocumentType 510
getDefaultPaperSource 510
getFooter 511
getHeader 511
getPrintedStatus 511
getPrinterName 511
getPrintPosition 512
getReport 512
isPrinterOpen 512
newPage 512
pageHeight 513
pageWidth 513
print 513

printActive 516
printPage 516
printReport 516
printUnformatted 516
setCustomPaperSize 518
setFooter 518
setHeader 519
setMargins 519
setPrinter 520
setPrintFileName 521
setPrintPosition 522
setReport 523
useCustomPrinterSettings 523
Using the Common Print Setup Dialog 523
Using the Print Progress Dialog 524
Examples of Printer Methods 525
Free-Format Printing 525
Previewing Print Output 528
Using the Select Pages To Print Dialog 529
Searching Previewed Output 530
Portable Printing 531
Process Class 533
Process Class Constants 533
Process Properties 533
adminInfo 534
node 534
number 534
persistentApp 534
schema 534
signOnTime 535
signOnUserCode 535
status 535
type 535
userCode 536
userExitCode 536
userInfo 537
Process Methods 538
addLockCallStackFilter 543
adjustObjectCachePriority 543
allowTransientToPersistentInvs 544
allowTransientToSharedTranInvs 544
allTransientInstances 544
analyzeTransientFileUsage 545
appServerPort 545
beginMethodProfiling 545
changeUserCode 546
classAccessFrequenciesStatus 546
clearLockCallStackFilter 546
compactTransientFile 547
countQueuedNotifications 547
createTransientMethod 547
currentStack 548
debug 549
deleteTransientMethod 549
disableAllTransTraceCallbacks 549
enableClassAccessFrequencies 549
enableTransTraceCallback 550
endMethodProfiling 550
executeIOScript 551
executeScript 552
executeTransientIOMethod 553

executeTransientMethod 554
extractRequestStatistics 555
extractWebStatistics 557
finalizePackages 558
getAllApps 559
getBufferStatistics 559
getCallStackInfo 561
getCommandLine 561
getComputerName 562
getDateTimeDelta 562
getExceptionHandlerStack 562
getErrorText 563
getIniFileName 563
getJadeInstallDirectory 563
getJadeHomeDirectory 563
getJadeWorkDirectory 564
getLastExtFunctionCallError 564
getLockCallStackFilter 564
getMethodCacheLimit 564
getMethodCacheStatistics 564
getMethodProfileInfo 566
getOSDetails 567
getOSPlatform 570
getPersistentDeadlockPriority 571
getProcessApp 571
getProfileString 571
getProgramDataDirectory 572
getRequestStatistics 573
getRpcServerStatistics 573
getSaveLockCallStack 574
getSignOnUsage 575
getStringPoolLimit 575
getTempPath 575
getTimers 575
getTrackedMethod 576
getTrackedMethodReceiver 577
getTrackedMethodReturnValue 577
getTrackedMethodStatus 577
getTransactionId 577
getTransactionId64 577
getTransactionTraceCallbacks 578
getTransactionTraceObject 578
getTransientDeadlockPriority 578
getTransientFileLength 579
getTransientFileName 579
getUserDataDirectory 579
initializePackages 580
isCommitting 580
isInExceptionState 580
isInImportedContext 580
isInLoadState 581
isInLockState 581
isInTransactionState 581
isInTransientTransactionState 582
isRunningScript 582
isUserDataPump 582
isUsingThinClient 582
iteratorsExcludeOfflineObjects 582
networkAddress 582
profileMethod 583
profiler 583

Contents XXV

prohibitBeginTransaction 584
prohibitPersistentUpdates 584
removeMethodProfileInfo 585
resumeTimers 585
rpsSuppressTransactionDeletes 585
sendCallStackInfo 585
sendMethodCacheStatistics 586
sendRequestStatistics 587
sendTransientFileAnalysis 588
sendTransientFileInfo 589
sendWebStatistics 590
setDateTimeDelta 591
setDefaultLockTimeout 591
setMethodCacheLimit 592
setObjectCachePriority 592
setPersistentDeadlockPriority 593
setProfileString 593
setSaveLockCallStack 594
setStringPoolLimit 595
setTransientDeadlockPriority 595
sleep 595
startMethodTracking 596
startTransactionTrace 597
stopMethodTracking 597
stopTransactionTrace 597
suspendTimers 597
transactionTraceStarted 598
transientPersistentInvsEnabled 598
transientSharedTranInvsEnabled 598
truncateOnDecimalOverflow 598
useUpdateLocks 599
waitForMethods 599
ProcessDict Class 600
ProcessStackArray Class 601
RealArray Class 602
Rectangle Class 603
Rectangle Properties 603
bottom 603
left 603
right 603
top 603
Rectangle Methods 604
copy 604
display 604
isEmpty 604
set 604
RelationalView Class 605
RelationalView Class Constants 605
RelationalView Properties 606
creator 606
name 606
rpsDatabaseName 607
rpsDatabaseType 607
rpsDefaultConnectionString 607
rpsDefaultPassword 607
rpsDefaultUserName 607
rpsExceptionCreate 608
rpsExceptionDelete 608
rpsExceptionUpdate 608
rpsLoggingOptions 608
rpsShowMethods 609

rpsShowVirtualProperties 609
rpsTopSchemaName 609
rpsUseOidClassInstMap 609
schema 609
timeCreated 610
RelationalView Methods 610
addUserAttribute 611
addUserTable 611
changeColumnName 612
columnExists 612
createExcludedJcfFile 612
excludeTableColumnName 613
excludeTableName 613
extractData 613
extractDataAll 614
extractDataUsingIniFileOptions 616
generateRpsTableCreationScript 616
getColumnFeature 617
getExcludedTableColumnNames 617
getExcludedTableNames 617
getRpsMappedClasses 618
getTableColumnNames 618
getTableNames 618
isODBCRelationalView 618
isRpsMapping 619
removeColumn 619
removeTable 619
tableExists 619
versionRpsMapping 619
RootSchemaSession Class 621
RootSchemaSession Properties 621
allowHiddenControlEvents 621
userSecurityLevel 621
Schema Class 623
Schema Class Constants 623
Schema Properties 623
externalDatabases 623
formsManagement 624
jomVersion 624
name 624
needsReorg 624
patchVersion 624
superschema 624
relationalViews 625
rpsDatabases 625
text 625
Schema Methods 625
addCompileTranslatableString 628
addUserCollectionSubclass 629
addUserSubclass 629
allClasses 629
allDatabases 630
allJadeInterfaces 630
allLibraries 630
allPrimitives 630
allSubschemas 630
buildFormData 630
constantNames 631
createWebServiceApplication 631
deleteUserSubclass 633
extractControlIdsCSV 633

extractControlIdsCSVforSchema 634
findClassInBranch 634
findClassInSubschema 635
findFormForLocale 635
findFormForLocaleInAllSchemas 635
findFormForLocaleInSupers 635
findGlobalConstantInBranch 635
findMeForm 635
findName 636
findProperty 636
findType 636
generateWSDL 636
getAllBaseLocales 637
getAllClasses 637
getAllFormTranslations 637
getAllInheritedLocales 637
getAllLocales 637
getAllLocalLocales 637
getAllSystemLocales 638
getAllRpsMappings 638
getAppliedPatches 638
getBaseLocalesLocal 639
getCategory 639
getClass 639
getClassByNumber 639
getConstant 639
getConstantCategory 639
getControlClasses 640
getCurrentLocaleId 640
getDefaultLocale 640
getExternalDatabase 640
getFormatAnywhereInPath 640
getFormatAnywhereInPathLatest 641
getFormatAnywhereInSubs 641
getFormatAnywhereInSubsLatest 641
getFunction 641
getGlobalClass 641
getGlobalConstant 641
getHtmlDocumentSource 641
getImportedClass 642
getImportedJadeinterface 642
getInheritedFormats 642
getInheritedXlatableStrings 643
getJadeInterface 643
getLibrary 643
getLocalClass 643
getLocale 643
getLocaleCurrencyInfo 643
getLocaleDateInfo 644
getLocaleFullInfo 644
getLocaleInSubschemas 645
getLocaleLocal 645
getLocaleNameInfo 645
getLocaleNumericInfo 645
getLocaleTimeInfo 646
getLocalFormats 646
getLocalLocaleInSubschemas 646
getLocalPrimitive 647
getName 647
getOidForObject 647
getPrimitive 647

Contents xxviii

getRelationalView 647
getRpsMapping 647
getSchema 647
getSubschema 648
getSubschemas 648
getUserAppliedPatches 648
getUserFormat 649
getWebServiceConsumerNames 649
globalException 649
importWSDL 649
isLocalLocale 650
loadHTMLDocuments 650
makeLocaleNameFromId 651
nonGUIGlobalExceptionHandler 651
regenerateRelationalView 651
removeWebConsumer 652
reorgInProgress 652
reorgIsWaitingForTransition 652
resetUserAppliedPatches 652
setHtmlDocumentSource 653
withAllSubschemas 653
withAllSuperschemas 653
SchemaEntity Class 654
SchemaEntity Class Constants 654
SchemaEntity Properties 654
abstract 654
access 655
name 655
number 655
subAccess 655
text 656
SchemaEntity Methods 656
getName 656
getPatchNumber 656
SchemaEntityNumberDict Class 657
Script Class 658
Script Properties 658
compiledOK 658
errorCode 658
errorLength 659
errorPosition 659
status 659
warningCount 659
Script Methods 660
getSource 660
inError 660
notCompiled 660
Set Class 661
Set Methods 661
add 661
copy 662
createIterator 662
getStatistics 663
includes 664
indexNear 664
indexNear64 665
remove 665
SetMergeIterator Class 665
SetMergeIterator Property 666
ignoreDuplicates 666
SetMergeIterator Methods 666

Contents xxix

addCollection 667
back 667
current 667
getCollectionAt 668
getCollectionCount 668
getCurrentCollection 668
isValid 668
next 668
removeCollection 669
reset 669
startAtObject 669
SortActor Class 670
SortActor Class Constants 671
SortActor Properties 671
ascending 671
fieldNo 672
length 672
numeric 672
random 673
sortType 673
startPosition 674
SortActorArray Class 675
SortActorArray Properties 675
kway 675
lcid 675
maxMem 675
Sound Class 676
Sound Properties 676
data 676
format 676
name 677
Sound Methods 677
isPlayable 677
loadFromFile 677
play 677
StringArray Class 679
StringUtf8Array Class 680
System Class 681
System Properties 681
name 681
nodes 681
System Methods 681
activateDeltaDatabase 684
beginIndividualRequestsLogging 685
beginLockContentionStats 686
beginObjectTracking 686
beginSample 687
beginSampleGroupDefinition 687
clearLockContentionStats 688
createSystemSequenceNumber 688
disableRemoteSampling 689
dumpCharacterEntityTable 690
enableRemoteSampling 690
endIndividualRequestsLogging 690
endLockContentionStats 691
endObjectTracking 691
endSample 691
endSampleGroupDefinition 692
findCharacterEntityByName 692
findCharacterEntityByNumber 692
forceOffUser 693

Contents

getAllUsers 693
getClassAccessFrequencies 693
getDatabaseRole 695
getDatabaseStats 695
getDatabaseSubrole 696
getDbDiskCacheStats 697
getDeltaDatabaseStatus 698
getEnvironmentServerIdentity 698
getLockContentionInfo 698
getLockContentionStats 699
getLocks 699
getMostAccessedClasses 700
getNotes 701
getObjectLockProcesses 702
getObjectPartitionID 703
getQueuedLocks 703
getRequestStats 704
getRpcServerStatistics 705
getStatistics 706
getStatistics64 708
getSystemSequenceNumberNext 709
getTimeInTransactionState 710
interruptUser 710
isDatabaseEncryptionEnabled 711
isDbArchival 711
isRemoteSamplingEnabled 711
isValidProcess 711
logObjectCaches 712
logRequestStatistics 713
logUserCommand 713
processDumpAllNodes 714
queryLockContentionStats 714
removeNode 714
sdsAuditEnableSecondaryApps 715
verifyDbEncryptionMasterKey 715
SystemException Class 716
TcpIpConnection Class 717
TcpIpConnection Class Constants 717
TcpIpConnection Properties 717
authenticationLibrary 718
cryptLibrary 719
decryptMethod 719
encryptMethod 719
genAuthChallengeMethod 720
genAuthResponseMethod 720
localInterface 721
localIpAddress 721
localPort 721
networkProxy 721
port 722
protocolFamily 722
remoteIpAddress 722
remoteName 723
remotePort 723
resolveRemoteName 723
usePresentationClient 723
sslContext 723
userObject 724
verifyAuthResponseMethod 724
TcpIpConnection Methods 725
close 725

Contents xxxi

closeAsynch 726
getMaxMessageSize 726
listen 727
listenAsynch 727
listenContinuous 728
listenContinuousAsynch 729
open 730
openAsynch 731
readBinary 732
readBinaryAsynch 732
readUntil 733
readUntilAsynch 734
writeBinary 734
writeBinaryAsynch 735
TimeArray Class 736
TimeFormat Class 737
TimeFormat Properties 737
amText 737
ampmIsSuffix 737
format 738
is12HourFormat 738
pmText 738
separator 738
showLeadingZeros 738
showSeconds 738
TimeFormat Method 738
defineTimeFormat 739
TimeStampArray Class 740
TimeStampIntervalArray Class 741
TranslatableString Class 742
TranslatableString Properties 742
formBuildDataRefs 743
locale 743
TranslatableString Method 743
updateCompile 744
Type Class 745
Type Properties 745
consts 745
methods 745
schema 745
superschemaType 745
Type Methods 746
allMethods 746
findConstant 746
findConstantInSuperschema 746
findProperty 746
getConstant 747
getConstants 747
getConstantsInSchema 747
getMethod 747
getMethods 747
getName 748
getProperty 748
inheritsFrom 748
instancesExist 748
UserInterfaceException Class 749
WebSession Class 750
WebSession Class Constant 750
WebSession Properties 750
lastAccessTime 750
sessionId 751

startTime ... 751
usePageSequencing ... 751
WebSession Methods ... 751
browserType ... 752
createVirtualDirectoryFile ... 752
deleteVirtualDirectoryFile ... 753
getCurrentLocale ... 753
getHttpParam ... 754
getHttpString ... 754
getServerVariable ... 754
getSessionForm ... 756
getWebSessionCount ... 756
isVDFilePresent ... 756
processRequest ... 756
removeSession ... 757
removeSessionWithMessage ... 757
reply ... 758
setCurrentLocale ... 758
timerEvent ... 759
WebSocketException Class ... 760

Before You Begin

The *JADE Encyclopaedia of Classes* is intended as a major source of information when you are developing or maintaining JADE applications.

Who Should Read this Encyclopaedia

The main audience for the *JADE Encyclopaedia of Classes* is expected to be developers of JADE application software products.

What's Included in this Encyclopaedia

The *JADE Encyclopaedia of Classes* has two chapters, and is divided into three volumes.

Chapter 1	Gives a reference to system classes and the constants, properties, and methods that they provide
Chapter 2	Gives a reference to Window classes and the constants, properties, methods, and events that they provide

Note that this second volume contains system (non-GUI) classes in the range **JadeSkinApplication** class through **WebSession** class, inclusive. Volume 1 (that is, **EncycloSys1.pdf**) contains system (non-GUI) classes in the range **ActiveXAutomation** class through **JadeSkin** class, inclusive. Chapter 2 (**Window** class and subclasses) is contained in Volume 3 (that is, **EncycloWin.pdf**).

Related Documentation

Other documents that are referred to in this encyclopaedia, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

Title	Related to…
JADE Database Administration Guide	Administering JADE databases
JADE Development Environment Administration Guide	Administering JADE development environments
JADE Development Environment User's Guide	Using the JADE development environment
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Installation and Configuration Guide	Installing and configuring JADE
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Object Manager Guide	JADE Object Manager administration
JADE Report Writer User's Guide	Using the JADE Report Writer to develop and run reports
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)
JADE Thin Client Guide	Administering JADE thin client environments

Conventions

The *JADE Encyclopaedia of Classes* uses consistent typographic conventions throughout.

Convention	Description
Arrow bullet (»)	Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.
Bold	Items that must be typed exactly as shown. For example, if instructed to type **foreach**, type all the bold characters exactly as they are printed. File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.
Italic	Parameter values or placeholders for information that must be provided; for example, if instructed to enter *class-name*, type the actual name of the class instead of the word or words shown in italic type. Italic type also signals a new term. An explanation accompanies the italicized type. Document titles and status and error messages are also shown in italic type.
Blue text	Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the "Object Methods" cross-reference to display that topic.
Bracket symbols ([ ])	Indicate optional items.
Vertical bar (|)	Separates alternative items.
`Monospaced font`	Syntax, code examples, and error and status message text.
ALL CAPITALS	Directory names, commands, and acronyms.
SMALL CAPITALS	Keyboard keys.

Key combinations and key sequences appear as follows.

Convention	Description
KEY1+KEY2	Press and hold down the first key and then press the second key. For example, "press Shift+F2" means to press and hold down the Shift key and press the F2 key. Then release both keys.
KEY1,KEY2	Press and release the first key, then press and release the second key. For example, "press Alt+F,X" means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.

Chapter 1 System Classes

JADE provides *system* classes. System classes are standard classes whose instances provide properties and methods to encapsulate the behavior of objects in your JADE applications. This chapter contains the classes summarized in the following table, and is divided into two volumes.

Note This volume (Volume 2) contains system (non-GUI) classes in the range **JadeSkinApplication** class through **WebSession** class, inclusive. Volume 1 (that is, **EncycloSys1.pdf**) contains system (non-GUI) classes in the range **ActiveXAutomation** class through **JadeSkin** class, inclusive.

Class	Description
ActiveXAutomation	Provides a superclass for each subclass created when an ActiveX automation object is imported
ActiveXInterface	Provides a superclass for all interfaces of imported ActiveX automation and control objects
ActiveXInvokeException	Defines behavior for exceptions that occur as a result of accessing an ActiveX property or invoking an ActiveX method
Application	Common superclass in the **RootSchema** for **Application** classes defined in subschemas
ApplicationContext	Stores transient instances of the application, package, process, and schema for the main application in which a package is imported and for each package application when a process begins
Array	Encapsulates behavior required to access entries in an ordered collection of like objects in which the member objects are referenced by their position in the collection
BinaryArray	Stores and retrieves binaries in an array of **Binary** primitive types
BooleanArray	Stores and retrieves Boolean values in an array of **Boolean** primitive types
Btree	Encapsulates behavior required to access entries in a collection by a key (index)
ByteArray	Stores and retrieves characters in an array of **Byte** primitive types
CharacterArray	Stores and retrieves characters in an array of **Character** primitive types
Class	Metaclass of all other JADE classes; that is, contains the definition of all JADE classes
CMDialog	Encapsulates behavior for the common dialog subclasses
CMDColor	Enables access to the common Color dialog
CMDFileOpen	Enables access to the common File Open dialog
CMDFileSave	Enables access to the common File Save dialog
CMDFont	Enables access to the common Font dialog
CMDPrint	Enables access to the common Print dialogs
Collection	Defines the common protocol for all collection subclasses
Connection	Provides a generalized interface for communicating with external systems

Class	Description
ConnectionException	Defines behavior for exceptions that occur as a result of communicating with external systems
ConstantNDict	Stores references to instances of the **Constant** class (or instances of subclasses)
CurrencyFormat	Stores Windows locale currency information
Database	Encapsulates the definition of a database for a schema, including the database files and the class mappings to those files
DateArray	Stores and retrieves dates in an array of **Date** primitive types
DateFormat	Stores Windows locale date information
DbFile	Encapsulates the definition of a database file and provides methods to perform file-level operations
DbFileArray	Stores and retrieves objects from an array of database files
DeadlockException	Defines behavior for exceptions that occur as a result of deadlocks
DecimalArray	Stores and retrieves decimals in an array of **Decimal** primitive types
Dictionary	Encapsulates behavior for storing and retrieving objects in a collection by a user-defined key
DynaDictionary	Encapsulates the behavior required to access entries in member key dictionary subclasses (that is, in dictionaries in which the keys are properties in the member objects)
Exception	Defines the protocol for raising and responding to exception conditions
ExceptionHandlerDesc	Describes an exception handler that is currently armed
ExternalArray	Represents rows in a result set generated from an SQL query containing a sort specification
ExternalCollection	Provides the common protocol for external collection classes
ExternalDatabase	Represents a connection to an external database
ExternalDictionary	Represents the rows in a result set generated from an SQL query with an **ORDER BY** sort specification
ExternalIterator	Encapsulates behavior required to sequentially access elements of a collection
ExternalObject	Base class for all external database classes
ExternalSet	Represents rows in a result set generated from an SQL query that has no sort specification
ExtKeyDictionary	Encapsulates the behavior required to access entries in external key dictionary subclasses
FatalError	Encapsulates behavior required for serious internal faults
File	Enables you to read and write disk files, either sequentially or with random access
FileException	Defines behavior for exceptions that occur as a result of file handling

Class	Description
FileFolder	Contains a collection of files or subdirectories
FileNode	Contains the properties and methods common to the **File** class and **FileFolder** class
FileNodeArray	Stores and retrieves objects from an array of file nodes
Global	Provides a means by which application-specific data can be shared among users of an application
GUIClass	Metaclass containing the definition of all Graphical User Interface (GUI) classes
HugeStringArray	Stores and retrieves large strings in an array of **String** primitive types
IDispatch	Provides a superclass for all ActiveX automation and control classes created in JADE during the ActiveX type library import process
IDispatchArray	Stores and retrieves objects from an array of **IDispatch** objects
Integer64Array	Stores and retrieves integers in an array of **Integer64** primitive types
IntegerArray	Stores and retrieves integers in an array of **Integer** primitive types
IntegrityViolation	Defines the behavior of exceptions raised as a result of integrity rule violations
InternetPipe	Provides an interface for communicating with JADE applications from the Internet through an Internet server
Iterator	Encapsulates behavior required to sequentially access elements of a collection
IUnknown	Encapsulates behavior implemented by all COM objects and inherited by all ActiveX interfaces
JadeAuditAccess	Provides access to information recorded in database transaction journals in a form convenient for consumption by JADE applications
JadeBytes	Stores and retrieves instances of unstructured data of arbitrary size
JadeDatabaseAdmin	Provides an Application Programming Interface (API) to perform database operations
JadeDbFilePartition	Provides an administrative API for manipulating and querying the state of database partitions
JadeDotNetInvokeException	Defines behavior for exceptions that occur as a result of accessing a .NET property or invoking a .NET method
JadeDotNetType	Provides a superclass for all imported .NET non-GUI types
JadeDynamicObject	Encapsulates the behavior required to access entries in dynamic objects (that is, in objects that represent collection statistics)
JadeDynamicObjectArray	Stores and retrieves objects from an array of **JadeDynamicObject** objects
JadeDynamicPropertyCluster	Stores one or more dynamic properties used to extend a class
JadeGenericMessage	Encapsulates the building and analysis of messages
JadeGenericMessagingIF	Provides message arrival and queue management callback methods

Class	Description
JadeGenericQueue	Encapsulates a destination for the transmission and retrieval of messages
JadeGenericQueueManager	Encapsulates the management of a single messaging queue
JadeHTMLClass	Implements the interface that enables you to support HTML pages in your JADE applications
JadeHTTPConnection	Enables applications to access the standard Internet protocol HTTP
JadeIdentifierArray	Stores and retrieves strings with a maximum length of 100 characters, which is the maximum length of a JADE identifier
JadeInternetTCPIPConnection	Implements the interface defined by the **TcpIpConnection** class specifically for the Internet Transmission Control Protocol / Internet Protocol (TCP/IP) API
JadeJson	Standalone JSON functionality that is independent of the Representational State Transfer (REST) Application Programming Interface (API)
JadeLicenceInfo	Encapsulates behavior required to get license information
JadeLog	Encapsulates behavior required to create text log files in JADE applications
JadeMessagingException	Defines the behavior of exceptions that arise when using the messaging framework
JadeMessagingFactory	Encapsulates the behavior for creating and opening messaging queues
JadeMetadataAnalyzer	Encapsulates behavior required to analyze JADE metadata
JadeMethodContext	Provides an interface for invoking asynchronous method calls
JadeMultiWorkerTcpConnection	Provides an interface for sharing the messages arriving on client sockets among a pool of worker server JADE applications
JadeMultiWorkerTcpTransport	Encapsulates behavior required for multiple user TCP/IP connections between JADE systems
JadeMultiWorkerTcpTransportIF	Provides TCP/IP multiple worker connection event callback methods
JadePatchControlInterface	Encapsulates behavior required to dynamically access patch versioning information
JadePrintData	Encapsulates the behavior required for report output data subclasses (that is, for direct print or preview)
JadePrintDirect	Provides output report output to be sent directly to the printer
JadePrintPage	Encapsulates behavior required to hold a page of printed output for preview
JadeProfiler	Encapsulates behavior required to configure what is profiled and reported in the JADE Interpreter
JadeRelationalAttributeIF	Provides an interface to expose soft attributes
JadeRelationalEntityIF	Provides an interface to expose soft entities, which are mapped to a table in the relational view

Class	Description
JadeRelationalQueryProviderIF	Provides a search implementation that optimally finds and filters instances of a soft entity
JadeReport	Encapsulates behavior required to access an entire printed report
JadeReportWriterManager	Provides a superclass for each JADE Report Writer Configuration or Designer application
JadeReportWriterReport	Provides methods that enable you to dynamically override JADE Report Writer details at run time
JadeReportWriterSecurity	Provides a superclass for all user **JadeReportWriterSecurity** subclasses
JadeRestService	Defines the behavior of REST-style Web service applications
JadeRpsDataPumpIF	Provides an interface for managing output sent to a relational database from an RPS Datapump application
JadeSerialPort	Provides methods for communicating with external systems through a serial port
JadeSkin	Stores JADE skins and encapsulates behavior required to maintain JADE skins
JadeSkinApplication	Stores JADE skins for forms and controls in applications
JadeSkinArea	Encapsulates behavior required to define and maintain rectangular skin areas
JadeSkinCategory	Stores skin category definitions
JadeSkinControl	Encapsulates behavior required to define and maintain skins for controls
JadeSkinEntity	Encapsulates behavior required to define and maintain skin entities
JadeSkinForm	Encapsulates behavior required to define and maintain skins for forms
JadeSkinMenu	Encapsulates behavior required to define and maintain skins for menus
JadeSkinRoot	Stores dictionaries that reference skin entities
JadeSkinSimpleButton	Stores skin definitions for simple buttons in all four states (that is, up, down, disabled, and rollover)
JadeSkinWindow	Stores the defined image and category of all skins
JadeSkinWindowStateImage	Stores images of window areas for specific states (that is, up, down, disabled, and rollover)
JadeSOAPException	Defines the behavior of exceptions that occur as a result of Web services
JadeSSLContext	Implements the Secure Sockets Layer (SSL) protocol that supports digital certificates over secure connections
JadeTableCell	Internally created proxy class providing direct access to table cells
JadeTableColumn	Internally created proxy class providing direct access to table columns
JadeTableElement	Internally created proxy class encapsulating behavior required to directly access table elements
JadeTableRow	Internally created proxy class providing direct access to table rows
JadeTableSheet	Internally created proxy class providing direct access to table sheets

Class	Description
JadeTcpIpProxy	Implements TCP/IP network proxy support that enables you to open a TCP/IP network connection through a proxy host
JadeTestCase	Provides unit testing functionality for user-written test subclasses
JadeTestListenerIF	Provides callback methods on the progress and results of unit testing
JadeTestRunner	Enables you to run unit test methods in subclasses of the **JadeTestCase** class
JadeTransactionTrace	Enables you to identify objects that are updated, created, and deleted within a transaction
JadeUserCollClass	Enables you to create a user collection class at run time
JadeWebService	Maintains all Web service information
JadeWebServiceConsumer	Defines the behavior of Web service consumers loaded into your application
JadeWebServiceProvider	Defines the behavior of Web service provider applications
JadeWebServiceSoapHeader	Defines the behavior of SOAP headers in Web service provider applications
JadeWebServiceUnknownHeader	Represents an unknown SOAP header in a Web service provider application
JadeWebSocket	Base class for handling a WebSocket connection
JadeWebSocketServer	Handles all incoming TCP/IP connections from the **JadeWebsocketIISNativeModule** on a specific interface and TCP port
JadeX509Certificate	Stores digital certificates in X509 format for use with the **JadeSSLContext** class that provides secure connections
JadeXMLAttribute	Represents an attribute of an XML element in an XML document tree
JadeXMLCDATA	Represents a CDATA section in an XML document tree
JadeXMLCharacterData	Abstract superclass of character-based nodes in an XML document tree
JadeXMLComment	Represents a comment in an XML document tree
JadeXMLDocument	Represents an XML document as a tree of nodes
JadeXMLDocumentParser	Represents the interface for parsing XML documents into a tree of objects
JadeXMLDocumentType	Represents the document type declaration in an XML document tree
JadeXMLElement	Represents an XML element in an XML document tree
JadeXMLException	Defines behavior for exceptions that occur as a result of XML processing
JadeXMLNode	Abstract superclass of all nodes in an XML document tree
JadeXMLParser	Abstract transient-only class that provides the interface for parsing XML documents
JadeXMLProcessingInstruction	Represents a processing instruction in an XML document tree
JadeXMLText	Represents the textual content within an XML document tree

Class	Description
List	Encapsulates behavior required to reference objects by their position in the collection
Locale	Defines the locales (languages) supported by a schema
LocaleFormat	Defines the common protocol for locale format information
LocaleFullInfo	Provides Windows locale information for the current workstation
LocaleNameInfo	Provides Windows locale name information for the current workstation
Lock	Describes the lock requests maintained by the system
LockArray	Stores and retrieves objects in an array of locks
LockContentionInfo	Stores information about lock contentions for a target persistent object
LockException	Defines the behavior of exceptions raised as a result of locking conflicts
MemberKeyDictionary	Encapsulates the behavior required to access entries in member key dictionary subclasses
MenuItem	Contains the definition of each menu command (item) on a menu
MergeIterator	Encapsulates behavior required to sequentially access elements of two or more compatible dictionaries
MethodCallDesc	Provides information at run time about currently active method calls
MultiMediaType	Provides the behavior for all types of multimedia subclasses
NamedPipe	Provides a generalized interface for communicating with external systems
Node	Class for which an instance exists for each node in a system
NormalException	Superclass of all non-fatal exceptions
Notification	Superclass for objects that describe the notifications maintained by the system
NotificationArray	Stores and retrieves objects from an array of notifications
NotificationException	Defines behavior for exceptions that occur as a result of notifications
NumberFormat	Stores Windows locale numeric information
Object	Defines default behavior for all other classes in the schema
ObjectArray	Stores and retrieves objects in an array
ObjectByObjectDict	Encapsulates the behavior required to map one object to another object
ObjectLongNameDict	Encapsulates the behavior for accessing the long names of objects
ObjMethodCallDesc	Provides information at run time about currently active method calls made to object methods (that is, methods defined on classes as opposed to primitive types)
ObjectSet	Stores and retrieves objects in a set
ODBCException	Defines behavior for exceptions that occur as a result of ODBC communications
OleObject	Stores the Object Linking and Editing (OLE) object images for the **OleControl** class

Class	Description
PointArray	Stores and retrieves points in an array of **Point** primitive types
PrimMethodCallDesc	Provides information at run time about currently active methods calls made to primitive methods
Printer	Handles printing
Process	Class for which an instance exists for each process in the system
ProcessDict	Encapsulates the behavior required to access process objects in a dictionary
ProcessStackArray	Encapsulates the behavior required to access method calls in the process stack array
RealArray	Stores and retrieves Real values in an array of **Real** primitive types
Rectangle	Encapsulates the dimensions of a rectangle
RelationalView	Enables views to be defined for use by the RPS **Datapump** application and to allow relational tools to access JADE
RootSchemaSession	Defines the common protocol for all Web session classes in subschemas
Schema	Represents the object model for a specific application domain
SchemaEntity	Superclass of a number of classes that participate in the definition of a schema
SchemaEntityNumberDict	Stores references to instances of subclasses of the **SchemaEntity** class
Script	Encapsulates the behavior of schema entities that have source code
Set	Encapsulates the behavior of collection set classes
SetMergeIterator	Encapsulates behavior required to sequentially access elements of two or more sets
SortActor	Contains properties that enable you to specify the precedence of records in the **File** class
SortActorArray	Container for **SortActor** objects
Sound	Contains the properties and methods for the sound multimedia type
StringArray	Stores and retrieves strings in an array of **String** primitive types
StringUtf8Array	Stores and retrieves strings in an array of **StringUtf8** primitive types
System	One instance of this class exists, representing an entire JADE system (that is, the installed JADE environment)
SystemException	Superclass of all exceptions relating to errors detected by the JADE kernel
TcpIpConnection	Implements the interface defined by the **Connection** class specifically for the TCP/IP API
TimeArray	Stores and retrieves times in an array of **Time** primitive types
TimeFormat	Stores Windows locale time information
TimeStampArray	Stores and retrieves timestamps in an array of **TimeStamp** primitive types

Class	Description
TimeStampIntervalArray	Stores and retrieves timestamp intervals in an array of **TimeStampInterval** primitive types
TranslatableString	Stores locale-dependent text to be displayed when a client is running an application
Type	Superclass of all class, primitive type, and interface meta classes
UserInterfaceException	Defines behavior for exceptions relating to the handling of windows
WebSession	Maintains Internet session information
WebSocketException	Defines behavior for WebSocket protocol exceptions

For details about user-interface (GUI) classes and their associated constants, properties, methods, and events, see Chapter 2, "Window Classes", in Volume 3.

JadeSkinApplication Class

The **JadeSkinApplication** class stores JADE skins for forms and controls in applications.

An application skin definition consists of a collection of form and control skins. You can define a skin with no form skins (that is, with control skins only) and the reverse.

For details about the properties defined in the **JadeSkinApplication** class, see "JadeSkinApplication Properties" and "JadeSkinApplication Method", in the following subsections. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinEntity

Inherited By: (None)

JadeSkinApplication Properties

The properties defined in the **JadeSkinApplication** class are summarized in the following table.

Property	Collection of…
myFormSkins	Form skins to be applied to the application
myControlSkins	Control skins to be applied to the an application

myFormSkins

Type: JadeSkinFormNameDict

Availability: Read or write at any time

The **myFormSkins** property of the **JadeSkinApplication** class contains a reference to a collection of form skins to be applied to an application.

This collection can contain multiple form skins where each form skin references a different skin category.

Each form skin is applied only to forms that have the same defined skin category (for details, see the **Window** class **skinCategoryName** property).

myControlSkins

Type: JadeSkinControlNameDict

Availability: Read or write at any time

The **myControlSkins** property of the **JadeSkinApplication** class contains a reference to a collection of the control skins to be applied to an application.

Each control class that can be skinned has an equivalent associated skin class. The **myControlSkins** collection can include multiple skins for each control type when they reference a different skin category.

A control skin is applied only to controls of the same type with the same defined skin category (for details, see the **Window** class **skinCategoryName** property).

JadeSkinApplication Method

The method defined in the **JadeSkinApplication** class is summarized in the following table.

Method	Description
updateSkinTimeStamp	Resets the instance timestamp and causes the skin build data to be rebuilt.

updateSkinTimeStamp

Signature `updateSkinTimeStamp() updating;`

The **updateSkinTimeStamp** method defined in the **JadeSkinApplication** class resets the instance timestamp and causes the skin build data to be rebuilt.

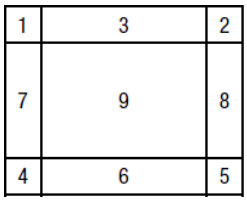
You would typically call this method if you updated a skin by any other means than using the **JadeSkinMaintenance** form or by loading a form and data definition (.**ddb** or .**ddx**) file.

JadeSkinArea Class

The **JadeSkinArea** class is the abstract class that defines the way in which a rectangular area is drawn.

Note Before you can define a skin area, a picture file (for example, a .**gif**, .**png**, .**bmp**, or .**jpg** file) must exist for each of the images that you want to specify.

The following image illustrates the layout of a skin, which is made up of eight border segments and an inner segment.



A **JadeSkinArea** is drawn as follows.

Segment 1

The top-left image (**imgBorderTopLeft**) is drawn at actual size. The top-left of the image is positioned at the top-left of the control.

The height drawn is usually the minimum of the top-left image height and the top-center strip height.

The exception is if the top-left image height is greater than the top-center strip height and the top-left image width is less than the left-center strip width. In that case, the top-left image height is used.

The top-left image can be higher than the top-center strip if the left-center strip is at least the same width.

Segment 2

The top-right image (**imgBorderTopRight**) is drawn at actual size. The top-right of the image is positioned at the top-right of the control.

The height drawn is usually the minimum of the top-right image height and the top-center strip height.

The exception is if the top-right image height is greater than the top-center strip height and the top-right image width is less than the right-center strip width. In that case, the top-right image height is used.

The top-right image can be higher than the top-center strip if the right-center strip is at least the same width.

Segment 3

The top center strip (**imgBorderTopStrip**) is drawn at actual image height and stretched horizontally between the top-left and top-right images.

Segment 4

The bottom-left image (**imgBorderBottomLeft**) is drawn at actual size. The bottom-left of the image is positioned at the bottom-left of the control.

The height drawn is usually the minimum of the bottom-left image height and the bottom-center strip height.

The exception is if the bottom-left image height is greater than the bottom-center strip height and the bottom-left image width is less than the left-center strip width. In that case, the bottom-left image height is used.

This allows the bottom-left image to be higher than the bottom-center strip if the left-center strip is at least the same width.

Segment 5

The bottom-right image (**imgBorderBottomRight**) is drawn at actual size. The bottom-right of the image is positioned at the bottom-right of the control.

The height drawn is usually the minimum of the bottom-right image height and the bottom-center strip height.

The exception is if the bottom-right image height is greater than the bottom-center strip height and the bottom-right image width is less than the right-center strip width. In that case, the bottom-right image height is used.

This allows the bottom-right image to be higher that bottom-center strip if the right-center strip is at least the same width.

Segment 6

The bottom center strip (**imgBorderBottomStrip**) is drawn at actual image height and stretched horizontally between the bottom-left and bottom-right images.

Segment 7

The left-center strip (**imgBorderLeftStrip**) is drawn at actual image width and stretched vertically between the top-left and the bottom-left images.

Segment 8

The right-center strip (**imgBorderRightStrip**) is drawn at actual image width and stretched vertically between the top-right and the bottom-right images

Segment 9

The center image (**imgInner**) is drawn stretched from the left-center image to the right-center image and from the top-center image to the bottom-center image. If there is no center image, it is filled with the background color specified for the skin.

Notes Segments 3 and 6 determine the respective top and bottom heights of the border. Segments 7 and 8 determine the respective left and right widths of the border.

Unexpected results may occur if an image has a size that is inappropriate or does not correspond to a specific area.

A corner segment is drawn to its full height if the width is the same as the corresponding left or right strip. For example, segment 1 can be higher that segment 3, provided that segment 1 is the same width as segment 7. You can use this to achieve rounded border effects. For details, see "JadeSkinWindow Class", later in this chapter.

The following characteristics of the **JadeSkinArea** class are affected by additional subclass property values.

- If optional border images (that is, areas 1 through 8 in the above image) are not present, the inner area of the skin is the entire area.
- You can define the optional inner image by setting the value of the **imgInner** property to a brush that is repeatedly drawn over the entire inner area or an image that is drawn centered in the inner area.
- A **backColor** property value is used only if the inner image (that is, the **imgInner** property) is not defined or it is not a brush.

For details about the **JadeSkinArea** class constants and the properties defined in the **JadeSkinArea** class, see "JadeSkinArea Class Constants" and "JadeSkinArea Properties", in the following subsections. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinEntity

Inherited By: JadeSkinMenu, JadeSkinWindow, JadeSkinWindowStateImage

JadeSkinArea Class Constants

The constants provided by the **JadeSkinArea** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
BorderStyle_3DRaised	3	BorderStyle_3DSunken	2
BorderStyle_Images	4	BorderStyle_None	0
BorderStyle_Single	1		

JadeSkinArea Properties

The properties defined in the **JadeSkinArea** class are summarized in the following table.

Property	Description
backColor	Background color of the area if the inner image is not supplied or it is not a brush
imgBorderBottomLeft	Optional bottom left of the area
imgBorderBottomRight	Optional bottom right of the area
imgBorderBottomStrip	Optional bottom strip of the area
imgBorderLeftStrip	Optional left strip of the area
imgBorderRightStrip	Optional right strip of the area
imgBorderTopLeft	Optional top left of the area
imgBorderTopRight	Optional top right of the area
imgBorderTopStrip	Optional top strip of the area
imgInner	Optional inner image for the area
innerIsBrush	Specifies whether the optional inner image is a brush for the entire area or a centered image (set to **true** by default)

backColor

Type: Integer

Availability: Read or write at any time

The **backColor** property of the **JadeSkinArea** class contains the global background color of the area if the inner image is not defined or it is not a brush. The default value of **Default_Color** for this property means that the defined value of the **Window** class **backColor** property is used, subject to the following rules.

The rules for the **backColor** of a form are:

- If the form has a **JadeSkinForm** skin set, the **JadeSkinForm.backColor** value is not **Default_Color**, and the **backColor** of the form is **3D Face**, the **backColor** value of the skin is used.
- If the above does not apply, the **backColor** value of the form is used.

The rules for the **backColor** of a control are:

- If the control is transparent, the background area of the control is not erased and **backColor** is ignored.
- If the **backColor** of the control is set to **Default_Color**, the effective **backColor** of the first parent whose **backColor** is not **Default_Color** is used, regardless of whether a skin is applied.
- If the control has its default **backColor** value set *or* if the skin was individually set on the control, as shown in the following example.

```
label1.setSkin(myJadeSkinLabel);
```

- If the control has a skin and the **backColor** of the skin is not **Default_Color**, the **backColor** of the skin is used.
- If the default **backColor** of the control is **3D Face**, the control is not a button or browse button, and the form has a **JadeSkinForm** skin set, the **JadeSkinForm.backColor** value is used unless its value is **Default_Color**.
- If either of the above does not apply, the **backColor** value of the control is used.

For more details about the **backColor** property, see the **Window** class **backColor** property.

imgBorderBottomLeft

Type: Binary

Availability: Read or write at any time

The **imgBorderBottomLeft** property of the **JadeSkinArea** class contains the optional image for the bottom left area of the skin. This image is drawn unstretched.

imgBorderBottomRight

Type: Binary

Availability: Read or write at any time

The **imgBorderBottomRight** property of the **JadeSkinArea** class contains the optional image for the bottom right area of the skin. This image is drawn unstretched.

imgBorderBottomStrip

Type: Binary

Availability: Read or write at any time

The **imgBorderBottomStrip** property of the **JadeSkinArea** class contains the optional image for the bottom strip of the skin. This image is drawn stretched.

imgBorderLeftStrip

Type: Binary

Availability: Read or write at any time

The **imgBorderLeftStrip** property of the **JadeSkinArea** class contains the optional image for the left strip of the skin. This image is drawn stretched.

imgBorderRightStrip

Type: Binary

Availability: Read or write at any time

The **imgBorderRightStrip** property of the **JadeSkinArea** class contains the optional image for the right strip of the skin. This image is drawn stretched.

imgBorderTopLeft

Type: Binary

Availability: Read or write at any time

The **imgBorderTopLeft** property of the **JadeSkinArea** class contains the optional image for the top left area of the skin. This image is drawn unstretched.

imgBorderTopRight

Type: Binary

Availability: Read or write at any time

The **imgBorderTopRight** property of the **JadeSkinArea** class contains the optional image for the top right area of the skin. This image is drawn unstretched.

imgBorderTopStrip

Type: Binary

Availability: Read or write at any time

The **imgBorderTopStrip** property of the **JadeSkinArea** class contains the optional image for the top strip of the skin. This image is drawn stretched.

imgInner

Type: Binary

Availability: Read or write at any time

The **imgInner** property of the **JadeSkinArea** class contains the optional image for the inner area of the skin.

You can define a brush that is repeatedly drawn over the entire inner area or an image that is drawn centered in the inner area.

If you do not define an image for this property or it is not a brush, the **backColor** property value is used for the inner area of the skin.

innerIsBrush

Type: Boolean

Availability: Read or write at any time

The **innerIsBrush** property of the **JadeSkinArea** class specifies whether the optional inner image (defined in the **imgInner** property) is a brush for the entire area or a centered image.

This property is set to **true** by default.

JadeSkinCategory Class

The **JadeSkinCategory** class holds the skin category definitions for applications, forms, and controls.

For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinEntity

Inherited By: (None)

JadeSkinControl Class and Subclasses

The **JadeSkinControl** class is the abstract superclass that provides the definition of elements common for each control that can be skinned.

Use the properties of the **JadeSkinArea** class to define the image drawn for the active border and the inner (client area) of each control. When erasing the inner area of a control:

1. If the control **backBrush** property of the form is not **null**, the inner area is erased using that brush.
2. If the control skin has a **JadeSkinArea** class **imgInner** property value that is a brush, the inner area is erased using that brush.
3. If the **backColor** property of the skin area is not **Default_Color** and the **backColor** property of the control is the default value or the skin was set by using the **Control** class **setSkin** method, erase using the **backColor** property value of the skin.
4. If the value of the **backColor** property of the control is **Color_3DFace** and the form of the control has a skin whose **backColor** property is not set to **Default_Color**, the inner area is erased using the **backColor** property value of the form's skin.
5. Erase using the **backColor** property value of the control.
6. If the control was erased using a color and the skin of the control has an inner image defined in the **JadeSkinArea** class **imgInner** property that is not a brush (that is, the **JadeSkinArea** class **innerIsBrush** property is set to **false**), that image is drawn centered in the inner area.

Skins do not apply to the **ActiveXControl**, **MultiMedia**, and **Ocx** control classes, as these are totally drawn by the controls themselves.

Note If an application is active with a skin set, a second application initiated from the same **jade.exe** executable is now drawn using the current Windows theme if that application does not have a skin set.

For details about the class constants and properties defined in the **JadeSkinControl** class, see "JadeSkinControl Class Constants", and "JadeSkinControl Properties", in the following subsections. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinWindow

Inherited By: JadeSkinBaseControl, JadeSkinBrowseButtons, JadeSkinButton, JadeSkinCheckBox, JadeSkinComboBox, JadeSkinFolder, JadeSkinFrame, JadeSkinGroupBox, JadeSkinJadeDockBase, JadeSkinJadeEditMask, JadeSkinJadeMask, JadeSkinJadeRichText, JadeSkinLabel, JadeSkinListBox, JadeSkinOleControl, JadeSkinOptionButton, JadeSkinPicture, JadeSkinProgressBar, JadeSkinScrollBar, JadeSkinSheet, JadeSkinStatusLine, JadeSkinTable, JadeSkinTextBox

JadeSkinControl Class Constants

The constants provided by the **JadeSkinControl** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
ApplyCondition_3D	2	ApplyCondition_All	0
ApplyCondition_Border	1		

JadeSkinControl Properties

The properties defined in the **JadeSkinControl** class and inherited by all subclasses are summarized in the following table.

Property	Description
applyCondition	Determines whether the border area of a control uses a skin
borderStyle	Contains the type of border to be drawn
focusBackColor	Contains the color to be used for the background of the control when the control has focus
focusForeColor	Contains the color to be used for drawing the text of the control when the control has focus
fontBold	Specifies whether the control font is bold when the control uses the default application font
fontItalic	Specifies whether the control font is italicized when the control uses the default application font
fontName	Font with which the control is drawn (the default null value indicates the control uses its own default font)
fontSize	Specifies the size of the control font when the control uses the default application font
fontStrikethru	Specifies whether the control font is strikethrough when the control uses the default application font
fontUnderline	Specifies whether the control font is underlined when the control uses the default application font
foreColor	Contains the color to be used for drawing the text of the control
foreColorDisabled	Contains the color to be used for drawing the text of the control when it is disabled

applyCondition

Type: Integer

Availability: Read or write at any time

The **applyCondition** property of the **JadeSkinControl** class determines whether the border area of the control uses a skin.

The **applyCondition** property values are listed in the following table.

Class Constant	Integer Value	Description
ApplyCondition_3D	2	The skin is applied to the control but the border area of the skin is displayed only if the **borderStyle** property for the control is set to **BorderStyle_3DSunken** (2) or **BorderStyle_3DRaised** (3). If the **borderStyle** property of the control is set to **BorderStyle_None** (0), the skin is displayed without showing a border. If the **borderStyle** property of the control is set to **BorderStyle_Single** (1), the single border is displayed and the rest of the control is displayed with a skin.

Class Constant	Integer Value	Description
ApplyCondition_All	0	The skin is applied to the control, including the border area definition of the skin.
ApplyCondition_Border	1	The skin is applied to the control but the border area of the skin is displayed only if the **borderStyle** property of the control is set to a value other than **BorderStyle_None** (0).

The default value is **ApplyCondition_Border** (1) for **BaseControl**, **Frame**, **JadeDockBar**, **JadeDockContainer**, **Label**, **ListBox**, **OleControl**, **Picture**, **StatusLine**, **Table**, and **TextBox** controls. For all other controls, the default value is **ApplyCondition_All** (0). For example, you can define a skin for a **Label** control with a border and if the **applyCondition** property of the skin is not **ApplyCondition_All** (0), the border displays depends on the value of the **borderStyle** property of the label.

borderStyle

Type: Integer

Availability: Read or write at any time

The **borderStyle** property of the **JadeSkinControl** class contains the type of border to be drawn on the control skin. The default value is **BorderStyle_Images** (**4**).

The **borderStyle** property values are listed in the following table.

JadeSkinArea Class Constant	Integer Value	Description
BorderStyle_3DRaised	3	Raised three-dimensional border (two pixels).
BorderStyle_3DSunken	2	Sunken three-dimensional border (two pixels).
BorderStyle_Images	4	Border is drawn using the supplied images of the **JadeSkinArea** class. If there are no images, the control does not have a border.
BorderStyle_None	0	No border is drawn.
BorderStyle_Single	1	Fixed single-line border.

If you set the **borderStyle** property to a value other than the default **BorderStyle_Images** (**4**), the defined border is drawn and the border images defined in the **JadeSkinArea** class are ignored.

For more details about control borders, see the **Window** class **borderStyle** property and the **JadeSkinControl** class **applyCondition** property.

focusBackColor

Type: Integer

Availability: Read or write at any time

The **focusBackColor** property of the **JadeSkinControl** class contains the background color of a control when the control has focus or a child of the control has focus. You can use the **focusBackColor** and **focusForeColor** properties to give the user a better visual prompt as to which control has focus.

When a skin is assigned to a control that uses this property, the value is used when the control has focus or a child of the control has focus if all of the following are true.

- The value is not **Black** (**0**)
- The value of the equivalent **focusBackColor** property of the control is **Black** (**0**)
- The equivalent **backColor** value of the control is the default for the control

Note If the control or the control's skin has a brush defined to erase the background area of the control, the effective value of the **focusBackColor** property is used instead.

The default value of zero (**Black**) indicates that the property is always ignored when drawing the control.

JADE uses the RGB scheme for colors. The valid range for a normal RGB color is zero (**0**) through **16,777,215** (**#FFFFFF**). The high byte of a number in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

When the value of the **focusBackColor** property is not **Black**, that property value is used instead of the value of the **backColor** property to erase the control area when the control has focus or a child of the control has focus.

Note If the control is transparent, the value of the **focusBackColor** property is not used. (The control area is not erased as part of the painting of the control.)

The **focusBackColor** property is not relevant to all controls. The controls that make use of this property must be capable of gaining the focus, they can be control parents, and they are not external controls such as .NET controls.

The controls that use the **focusBackColor** property are:

- BaseControl
- Button
- CheckBox
- ComboBox
- Folder
- Frame
- GroupBox
- Sheet
- ListBox
- OptionButton
- Picture
- StatusLine
- Table
- TextBox
- JadeMask

- JadeDockBar
- JadeDockContainer
- JadeEditMask
- JadeRichText.

In addition, the **focusBackColor** property is defined in the **Control** class so that you can specify it for individual controls in development and at run time.

focusForeColor

Type: Integer

Availability: Read or write at any time

The **focusForeColor** property of the **JadeSkinControl** class contains the foreground color used to display text associated with a control when the control has focus or a child of the control has focus. You can use the **focusForeColor** and **focusBackColor** properties to give the user a better visual prompt as to which control has focus.

When a skin is assigned to a control that uses this property, the value is used when the control has focus or a child of the control has focus if all of the following are true.

- The value is not **Black** (**0**)
- The value of the equivalent **focusForeColor** property of the control is **Black** (**0**)
- The equivalent **foreColor** value of the control is the default for the control

The default value of zero (**Black**) indicates that the property is always ignored when drawing the control.

JADE uses the RGB scheme for colors. The valid range for a normal RGB color is zero (**0**) through **16,777,215** (**#FFFFFF**). The high byte of a number in this range equals **0**; the lower three bytes (from least to most significant byte) determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is **128**, JADE uses the system colors, as defined in the Control Panel of the user. To determine the **Integer** value of a color from the RGB values:

```
int:= RedValue + (GreenValue * 256) + (BlueValue * 256 * 256);
```

When the value of the **focusForeColor** property is not **Black**, that property value is used instead of the value of the **foreColor** property to draw the text associated with the control when the control has focus or a child of the control has focus.

The **focusForeColor** property is not relevant to all controls. The controls that make use of this property must be capable of gaining the focus, they can be control parents, and they are not external controls such as .NET controls.

The controls that use the **focusForeColor** property are:

- BaseControl
- Button
- CheckBox
- ComboBox
- Folder

- Frame
- GroupBox
- Sheet
- ListBox
- OptionButton
- Picture
- StatusLine
- Table
- TextBox
- JadeMask
- JadeDockBar
- JadeDockContainer
- JadeEditMask
- JadeRichText.

In addition, the **focusForeColor** property is defined in the **Control** class so that you can specify it for individual controls in development and at run time.

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **JadeSkinControl** class specifies whether the font style of the control skin is bold. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontBold** property are listed in the following table.

Value	Description
true	Turns on the bold formatting
false	Turns off the bold formatting (the default)

Use the **fontBold** property to format text in a control skin, either in the JADE development environment or at run time by using logic.

If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method.

Note The font uses the application font if the **fontName** property for the control skin is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontItalic

Type: Boolean

Availability: Read or write at any time

The **fontItalic** property of the **JadeSkinControl** class specifies whether the font style of the control skin is italics. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontItalic** property are listed in the following table.

Value	Description
true	Turns on the italic formatting
false	Turns off the italic formatting (the default)

Use the **fontItalic** property to format text in a control skin, either in the JADE development environment or at run time by using logic.

If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method.

Note The font uses the application font if the **fontName** property for the control skin is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontName

Type: String[31]

Availability: Read or write at any time

The **fontName** property of the **JadeSkinControl** class contains the font used to display text in a control skin. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method. At run time, the **fontName** property returns an empty string if the control is using the default application font. Use **app.fontName** to obtain the actual font name.

The default value for the **fontName** property is determined by the system. Fonts that are available with JADE vary, according to your system configuration, display devices, and printing devices.

Notes Changing the **fontName** property to an empty string causes the control skin to use the default font. The **fontBold** and **fontItalic** properties revert to the font of the application.

If a control is using the default font for the application (that is, this property contains the null value (**""**), changing any font property of the control causes the control to use a local font constructed by using the application font values with the changed font attribute.

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property of the **JadeSkinControl** class contains the size of the font used for text displayed in a control skin. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

Use the **fontSize** property to format text in a control skin, either in the JADE development environment or at run time by using logic.

Use the **fontSize** property to format text in the required font size. The default value (**0**) is determined by the system. To change the default, specify the size of the font in points. If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method.

Note The font uses the application font if the **fontName** property for the control skin is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontStrikethru

Type: Boolean

Availability: Read or write at any time

The **fontStrikethru** property of the **JadeSkinControl** class specifies whether the font style is strikethrough. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontStrikethru** property are listed in the following table.

Value	Description
true	Turns on the strikethru formatting
false	Turns off the strikethrough formatting (the default)

Use the **fontStrikethru** property to format text in a control skin, either in the JADE development environment or at run time by using logic.

If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method.

Note The font uses the application font if the **fontName** property for the control skin is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontUnderline

Type: Boolean

Availability: Read or write at any time

The **fontUnderline** property of the **JadeSkinControl** class specifies whether the font style of the control skin is underlined. This property is defined for all subclasses of the **JadeSkinControl** class, but it has no meaning in some cases. For example, a **ScrollBar** control has no text, and therefore the font is not relevant.

The settings for the **fontUnderline** property are listed in the following table.

Value	Description
true	Turns on the underline formatting
false	Turns off the underline formatting (the default)

Use the **fontUnderline** property to format text in a control skin, either in the JADE development environment or at run time by using logic.

If the **fontName** property is null (the default), the control continues to use its own defined font. The skin font is ignored by any control that has its own defined font unless the skin of the control has been set by using the **Control** class **setSkin** method.

Note The font uses the application font if the **fontName** property for the control skin is set to **Default** during painting or to an empty string at run time. The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

foreColor

Type: Integer

Availability: Read or write at any time

The **foreColor** property of the **JadeSkinControl** class contains the foreground color used to draw text in a control skin. JADE uses the RGB scheme for colors.

If the skin **captionActiveForeColor** is not the **Default_Color** default value, it is still ignored if the value of the **foreColor** property of the control is anything other than its own standard default value.

The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that a color is not set and that the control uses its own defined value of the **foreColor** property.

foreColorDisabled

Type: Integer

Availability: Read or write at any time

The **foreColorDisabled** property of the **JadeSkinControl** class contains the foreground color used to draw text in a disabled control skin. JADE uses the RGB scheme for colors.

The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that a color is not set and that the control draws its text as normal.

JadeSkinBaseControl Class

The **JadeSkinBaseControl** class holds the definition of a skin for subclasses of the **BaseControl** class.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinBrowseButtons Class

The **JadeSkinBrowseButtons** class holds the definition of a skin for **BrowseButtons** controls.

If a button image is not supplied for a non-up state, the up image is used. For painting to be successful, the skin requires all of the up images to be supplied. For details about the properties defined in the **JadeSkinBrowseButtons** class, see "JadeSkinBrowseButtons Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinBrowseButtons Properties

The properties defined in the **JadeSkinBrowseButtons** class are summarized in the following table.

Property	Reference to the …
myFirstButton	Full image and up, disabled, down, and rollover states of the **First** button
myLastButton	Full image and up, disabled, down, and rollover states of the **Last** button
myNextButton	Full image and up, disabled, down, and rollover states of the **Next** button
myPriorButton	Full image and up, disabled, down, and rollover states of the **Prior** button

myFirstButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myFirstButton** property of the **JadeSkinBrowseButtons** class contains a reference to the full image and the up, disabled, down, and rollover states of the **First** button.

myLastButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myLastButton** property of the **JadeSkinBrowseButtons** class contains a reference to the full image and the up, disabled, down, and rollover states of the **Last** button.

myNextButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myNextButton** property of the **JadeSkinBrowseButtons** class contains a reference to the full image and the up, disabled, down, and rollover states of the **Next** button.

myPriorButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myPriorButton** property of the **JadeSkinBrowseButtons** class contains a reference to the full image and the up, disabled, down, and rollover states of the **Prior** button.

JadeSkinButton Class

The **JadeSkinButton** class holds the definition of a skin for **Button** controls.

Each state can consist of up to eight border segments and an inner image, an inner image only, or no images (in which case the background color is used to fill the non-border area). These images are drawn inside any defined border area. If you do not define a specific state, the **myButtonUp** image is used.

The following image on the left is an example of a button with a raised three-dimensional effect, and the image on the right is an example of a button with a sunken three-dimensional effect.

For details about the properties defined in the **JadeSkinButton** class, see "JadeSkinButton Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinButton Properties

The properties defined in the **JadeSkinButton** class are summarized in the following table.

Property	Description
createRegionFromMask	Specifies whether the **JadeSkinWindow**::**myImageMask** property is used to create a region
myButtonDisabled	Reference to the image drawn for the disabled button state
myButtonDown	Reference to the image drawn for the down button state
myButtonFocus	Reference to the image drawn for the focus button state
myButtonFocusDown	Reference to the image drawn for the focus down button state
myButtonRollOver	Reference to the image drawn for the rollover button state

Property	Description
myButtonRollUnder	Reference to the image drawn for the roll-under button state
myButtonUp	Reference to the image drawn for the up button state (the default state)

createRegionFromMask

Type: Boolean

Availability: Read or write at any time

The **createRegionFromMask** property of the **JadeSkinButton** class specifies whether the **JadeSkinWindow** class **myImageMask** property is used to create a region for the **Button** control.

If you set this property to **true**, the **myImageMask** property is used to create a region to be applied to the control.

The default value of **false** indicates that the full rectangular button area is drawn using the skin. The region defined by the **myImageMask** property then applies only to any mouse actions. For example, if the button is an unusual shaped image on a background, the button then only displays the rollover and click images when the mouse is over that special area.

myButtonDisabled

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonDisabled** property of the **JadeSkinButton** class contains a reference to the image drawn for the disabled button state.

myButtonDown

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonDown** property of the **JadeSkinButton** class contains a reference to the image drawn for the down button state.

myButtonFocus

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonFocus** property of the **JadeSkinButton** class contains a reference to the image drawn for the focus button state.

myButtonFocusDown

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonFocusDown** property of the **JadeSkinButton** class contains a reference to the image drawn for the focus down button state.

myButtonRollOver

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonRollOver** property of the **JadeSkinButton** class contains a reference to the image drawn for the rollover button state.

myButtonRollUnder

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonRollUnder** property of the **JadeSkinButton** class contains a reference to the image drawn for the roll-under button state that is a rollover state when the button is down.

myButtonUp

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myButtonUp** property of the **JadeSkinButton** class contains a reference to the image drawn for the up (default) button state.

JadeSkinCheckBox Class

The **JadeSkinCheckBox** class holds the definition of a skin for **CheckBox** controls.

If you do not supply a specific state, the appropriate up image is used. If the check box button image of the skin is higher than the check box control using that skin, the check box control is enlarged in height to display the entire button image.

If you do not supply the appropriate up image, the default check box image is drawn. For example, the following image is an example of a **CheckBox** control with the **Control** class **borderStyle** property set to **BorderStyle_ 3DSunken** and the **Control** class **backBrush** property set.

For details about the properties defined in the **JadeSkinCheckBox** class, see "JadeSkinCheckBox Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinCheckBox Properties

The properties defined in the **JadeSkinCheckBox** class are summarized in the following table.

Property	Reference to the …
myFalseImage	Full image and up, disabled, down, and rollover states of the **false** value of check boxes
myTrueImage	Full image and up, disabled, down, and rollover states of the **true** value of check boxes

myFalseImage

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myFalseImage** property of the **JadeSkinCheckBox** class contains a reference to the full image and the up, disabled, down, and rollover states of the **false** value of check box controls.

myTrueImage

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myTrueImage** property of the **JadeSkinCheckBox** class contains a reference to the full image and the up, disabled, down, and rollover states of the **true** value of check box controls.

JadeSkinComboBox Class

The **JadeSkinComboBox** class holds the definition of a skin for **ComboBox** controls. When defining the skin for a **ComboBox** control, note the following points.

- A simple combo box is a text box followed by a list box covering the whole combo area.
- A spin box ignores any defined buttons for the combo box skin. The skin of the vertical scroll bar for the application is used to draw the spin box over the top of any defined border.
- When the combo box has a text box portion, the text box allows only a solid background color and does not successfully handle any defined brush.
- A combo box skin consists of a border definition and a button that is placed on the right side of the border area. The way in which it is painted depends on the defined border, as follows.
  - If the combo box skin has the **JadeSkinControl** class **borderStyle** property set to **BorderStyle_UseImages**, the button image is centered vertically and offset from the right-hand edge of the combo box by the value of the **buttonRightOffset** property, as shown in the following example.

- If the combo box skin has the **borderStyle** property set to a value other than **BorderStyle_UseImages**, the button is drawn inside whatever border is defined (against right inner edge of the border and stretched vertically), as shown in the following example.

For details about the properties defined in the **JadeSkinComboBox** class, see "JadeSkinComboBox Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinComboBox Properties

The properties defined in the **JadeSkinComboBox** class are summarized in the following table.

Property	Description
buttonRightOffset	Contains the number of pixels from the right edge to position the right edge of the combo box button.
imgComboButtonDownRollOver	Contains the extra button state image required for the combo box button rollover state in the down position.
myComboButton	Reference to the images for the up (normal), down (list box displayed), rollover (when in the up position), and disabled states.
myListBoxSkin	Reference to the skin used to draw the list box part of the combo box. The list box is not drawn with a skin if the value of this property is null.
mySimpleComboTextBoxSkin	Reference to the skin used to draw the text box part of a simple combo box.

buttonRightOffset

Type: Integer

Availability: Read or write at any time

The **buttonRightOffset** property of the **JadeSkinComboBox** class contains the number of pixels from the right edge to position the right edge of the combo box button.

This property is ignored if the value of the **JadeSkinControl** class **borderStyle** property is not **BorderStyle_Images** (**4**).

The button is centered vertically.

imgComboButtonDownRollOver

Type: Binary

Availability: Read or write at any time

The **imgComboButtonDownRollOver** property of the **JadeSkinComboBox** class contains the extra button rollover button state image for the combo box button in the down position.

myComboButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myComboButton** property of the **JadeSkinComboBox** class contains a reference to the images for the up (normal), down (list box displayed), rollover (when in the up position), and disabled states of the combo box button.

myListBoxSkin

Type: JadeSkinListBox

Availability: Read or write at any time

The **myListBoxSkin** property of the **JadeSkinComboBox** class contains a reference to the skin used to draw the list box area of the combo box.

If this property has a null value, the list box is not drawn with a skin.

mySimpleComboTextBoxSkin

Type: JadeSkinTextBox

Availability: Read or write at any time

The **mySimpleComboTextBoxSkin** property of the **JadeSkinComboBox** class contains a reference to the skin used to draw the text box area of a simple combo box (that is, a text box followed by a list box covering the whole combo box area).

JadeSkinFolder Class

The **JadeSkinFolder** class holds the definition of a skin for **Folder** controls.

For details about the properties defined in the **JadeSkinFolder** class, see "JadeSkinFolder Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinFolder Properties

The properties defined in the **JadeSkinFolder** class are summarized in the following table.

Property	Description
myTabsButton	Reference to the skin used to draw the tabs of a folder with the **tabsStyle** property set to **TabsStyle_Buttons**.
tabActiveColor	Contains the color used to draw the background area of active tabs.
tabHeight	Contains the height of the tabs in a folder that uses the default tab height.
tabInactiveColor	Contains the color used to draw the background area of inactive tabs.

myTabsButton

Type: JadeSkinButton

Availability: Read or write at any time

The **myTabsButton** property of the **JadeSkinFolder** class contains a reference to the skin used to draw the tabs of a folder that has the **tabsStyle** property set to **TabsStyle_Buttons** (1).

If this property has a null value, no skin is applied to tab buttons.

Note A button reference may be provided with a sheet skin (for details, see the **JadeSkinSheet** class **myTabButton** property). A sheet skin button image overrides any button image provided by the folder.

Defining several sheet skins with different categories enables you to have different images and colors for the tabs of a folder. To achieve this, set the **Window**::**skinCategoryName** property on each sheet to match the category of the sheet skin that you require.

---

tabActiveColor

Type: Integer

Availability: Read or write at any time

The **tabActiveColor** property of the **JadeSkinFolder** class contains the color used to draw the background area of the active tab. JADE uses the RGB scheme for colors. The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that the normal color of the folder is used.

The **tabActiveColor** property values for the skin are ignored for any sheet that has a **backColor** property value other than **Color_3Dface**, because the tab of the sheet is then drawn using that color.

tabHeight

Type: Integer

Availability: Read or write at any time

The **tabHeight** property of the **JadeSkinFolder** class contains the height of the tabs in a folder that uses the default tab height determination (that is, the **Folder** class **tabsHeight** property is set to the default value of **0**).

If the height of the tabs for the folder has been specifically set, the **tabHeight** property of the skin is ignored. By default, the **tabHeight** property is set to zero (**0**), indicating that the default height is the calculated text height using the font of the folder. If the **tabHeight** property is set to a positive value, each tab is drawn the specified number of pixels high.

Note If the **tabHeight** property is zero (**0**) and a button image is provided (by the **JadeSkinSheet** class **myTabButton** property), the height of the tabs is incremented by the height of the top and bottom border areas of the button and by the height of its skin top and bottom images (that is, the **JadeSkinButton** class **myButtonUp** property **imgBorderBottomStrip** and **imgBorderTopStrip** values).

---

tabInactiveColor

Type: Integer

Availability: Read or write at any time

The **tabInactiveColor** property of the **JadeSkinFolder** class contains the color used to draw the background area of the inactive tab. JADE uses the RGB scheme for colors.

The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that the normal color of the folder is used.

The values of the **tabInactiveColor** property for the skin are ignored for any sheet that has a **backColor** property value other than **Color_3Dface**, because the tab of the sheet is then drawn using that color.

JadeSkinFrame Class

The **JadeSkinFrame** class holds the definition of a skin for **Frame** controls.

Inherits From: JadeSkinControl

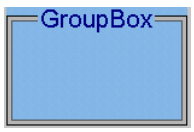
Inherited By: (None)

JadeSkinGroupBox Class

The **JadeSkinGroupBox** class holds the definition of a skin for **GroupBox** controls. As a group box control has no non-client area (border area) and the entire skin image is drawn in the client area, children may be positioned anywhere within that control and cover the drawn images.

The group box skin images are drawn over the entire control area. The drawing of the caption depends on whether the skin definition includes a skin label reference (**myLabelSkin**). If there is no skin label reference, the caption is drawn transparently over the skin image. If the skin references a label definition, that label skin is drawn on top of the group box skin image. The label is sized so that there is a three-pixel gap from the border area of the label to the left, right, top, and bottom of the caption. The caption or the label is drawn at the position indicated by the **captionPosition**, **captionPositionLeftOffset**, and **captionPositionTopOffset** properties.

The following image is an example of two group boxes. The example at the left has only the top and right border strips set.



For details about the constants and properties defined in the **JadeSkinGroupBox** class, see "JadeSkinGroupBox Class Constants" and "JadeSkinGroupBox Properties", in the following subsections.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinGroupBox Class Constants

The constants provided by the **JadeSkinGroupBox** class are listed in the following table.

Constant	Integer Value	Description
CaptionPosition_Left_Top	0	Left-justified at the top
CaptionPosition_Left_Middle	1	Left-justified and centered vertically
CaptionPosition_Left_Bottom	2	Left-justified at the bottom
CaptionPosition_Right_Top	3	Right-justified at the top
CaptionPosition_Right_Middle	4	Right-justified and centered vertically
CaptionPosition_Right_Bottom	5	Right-justified at the bottom
CaptionPosition_Center_Top	6	Centered horizontally at the top
CaptionPosition_Center_Middle	7	Centered horizontally and vertically
CaptionPosition_Center_Bottom	8	Centered horizontally at the bottom

JadeSkinGroupBox Properties

The properties defined in the **JadeSkinGroupBox** class are summarized in the following table.

Property	Contains…
captionPosition	The position of the caption.
captionPositionLeftOffset	The value that is added to the calculated left position resulting from the **captionPosition** property value.
captionPositionTopOffset	The value that is added to the calculated top position resulting from the **captionPosition** property value.
myLabelSkin	A reference to the **JadeSkinLabel** class.

captionPosition

Type: Integer

Availability: Read or write at any time

The **captionPosition** property of the **JadeSkinGroupBox** class contains the position of the caption (or the value of the **myLabelSkin** property) using the constant values listed in the following table.

Class Constant	Integer Value	Description
CaptionPosition_Left_Top	0	Left-justified at the top (the default)
CaptionPosition_Left_Middle	1	Left-justified and centered vertically
CaptionPosition_Left_Bottom	2	Left-justified at the bottom
CaptionPosition_Right_Top	3	Right-justified at the top
CaptionPosition_Right_Middle	4	Right-justified and centered vertically

Class Constant	Integer Value	Description
CaptionPosition_Right_Bottom	5	Right-justified at the bottom
CaptionPosition_Center_Top	6	Centered horizontally at the top
CaptionPosition_Center_Middle	7	Centered horizontally and vertically
CaptionPosition_Center_Bottom	8	Centered horizontally at the bottom

captionPositionLeftOffset

Type: Integer

Availability: Read or write at any time

The **captionPositionLeftOffset** property of the **JadeSkinGroupBox** class contains the value that is added to the calculated left position resulting from the **captionPosition** property value.

For example, to position the caption nine pixels from the top right of the group box control, set the value of the **captionPositionLeftOffset** to **-9** and the value of the **captionPosition** property to **CaptionPosition_Right_Top** (3).

The value of the **captionPositionLeftOffset** property is ignored if the position causes the caption to fall outside the group box area.

The default value of zero (**0**) indicates that the left position of the group box caption is not offset.

captionPositionTopOffset

Type: Integer

Availability: Read or write at any time

The **captionPositionTopOffset** property of the **JadeSkinGroupBox** class contains the value that is added to the calculated top position resulting from the **captionPosition** property value.

For example, to position the caption nine pixels from the bottom right of the group box control, set the value of the **captionPositionTopOffset** to **-9** and the value of the **captionPosition** property to **CaptionPosition_Right_Bottom** (5).

The value of the **captionPositionTopOffset** property is ignored if the position causes the caption to fall outside the group box area.

The default value of zero (**0**) indicates that the top position of the group box caption is not offset.

myLabelSkin

Type: JadeSkinLabel

Availability: Read or write at any time

The **myLabelSkin** property of the **JadeSkinGroupBox** class contains a reference to the **JadeSkinLabel** class.

If this property is set, this skin is used to draw the text as though it were a label so that the text portion of the group box can have its own border and background color, brush, or image.

If this property has a null value, no skin is applied to the group box.

JadeSkinHScroll Class

The **JadeSkinHScroll** class holds the definition of a skin for **HScroll** subclasses of the **ScrollBar** control.

Note This skin is also used for drawing the horizontal scroll bar for any control or form in the application.

The height of the border area and the height of the **myLeftButton.imgUp** image determine the height of the scroll bar. For a scroll bar control, the image is stretched vertically to fit the area inside the borders, as shown in the example in the following image.

If you do not supply the image for a specific state, the appropriate up image is used. The default scroll button image is drawn if you do not supply the up image.

For details about the properties defined in the **JadeSkinHScroll** class, see "JadeSkinHScroll Properties", in the following subsection.

Inherits From: JadeSkinScrollBar

Inherited By: (None)

JadeSkinHScroll Properties

The properties defined in the **JadeSkinHScroll** class are summarized in the following table.

Property	Reference to the image used to draw the …
myLeftButton	Left button of a horizontal scroll bar in its various states (the button is placed inside the scroll bar borders)
myRightButton	Right button of a horizontal scroll bar in its various states (the button is placed inside the scroll bar borders)

myLeftButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myLeftButton** property of the **JadeSkinHScroll** class contains a reference to the image used to draw the left button of a horizontal scroll bar in its up, disabled, down, and rollover states. (The button is placed inside the scroll bar borders.)

myRightButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myRightButton** property of the **JadeSkinHScroll** class contains a reference to the image used to draw the right button of a horizontal scroll bar in its up, disabled, down, and rollover states. (The button is placed inside the scroll bar borders.)

JadeSkinJadeDockBar Class

The **JadeSkinJadeDockBar** class holds the definition of a skin for **JadeDockBar** controls.

Inherits From: JadeSkinJadeDockBase

Inherited By: (None)

JadeSkinJadeDockBase Class

The **JadeSkinJadeDockBase** class is the abstract class that defines elements of a skin for docking controls, described in the **JadeDockBase** class, in Chapter 2.

For details about the properties defined in the **JadeSkinJadeDockBase** class, see "JadeSkinJadeDockBase Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: JadeSkinJadeDockBar, JadeSkinJadeDockContainer

JadeSkinJadeDockBase Properties

The properties defined in the **JadeSkinJadeDockBase** class are summarized in the following table.

Property	Contains the image drawn for a …
myHorizontalGripBar	Horizontal grip bar for a vertically aligned docking control
myHorizontalResizeBar	Horizontal resize bar drawn on the bottom border of a docking control
myVerticalGripBar	Vertical grip bar for a horizontally aligned docking control
myVerticalResizeBar	Vertical resize bar drawn on the right border of a docking control

myHorizontalGripBar

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myHorizontalGripBar** property of the **JadeSkinJadeDockBase** class contains a reference to the image drawn horizontally for the grip bar of a vertically aligned docking control (that is, the grip bar is the two horizontal lines at the top of the image).

The image is stretched to fit the width of the docking control. The standard docking control grip is drawn if you do not supply an image.

myHorizontalResizeBar

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myHorizontalResizeBar** property of the **JadeSkinJadeDockBase** class contains a reference to the image drawn horizontally for a resize bar drawn on the bottom border of a vertical docking control. In the previous example, the horizontal resize bar is the dark border at the bottom of the image. The image is stretched to fit the width of the docking control. The standard docking control resize bar is drawn if you do not supply an image.

myVerticalGripBar

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myVerticalGripBar** property of the **JadeSkinJadeDockBase** class contains a reference to the image drawn vertically for the grip bar for a horizontally aligned docking control (that is, the grip bar is the two vertical lines at the left of the image).

The image is stretched to fit the height of the docking control. The standard docking control grip is drawn if you do not supply an image.

myVerticalResizeBar

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myVerticalResizeBar** property of the **JadeSkinJadeDockBase** class contains a reference to the image drawn vertically for a resize bar drawn on the right border of a horizontal docking control. In the previous example, the vertical resize bar is the dark border at the right of the image.

The image is stretched to fit the height of the docking control. The standard docking control resize bar is drawn if you do not supply an image.

JadeSkinJadeDockContainer Class

The **JadeSkinJadeDockContainer** class holds the definition of a skin for **JadeDockContainer** controls.

Inherits From: JadeSkinJadeDockBase

Inherited By: (None)

JadeSkinJadeEditMask Class

The **JadeSkinJadeEditMask** class holds the definition of a skin for **JadeEditMask** controls.

Note The background area of a **JadeEditMask** control outside of the text box children is always drawn using the value of the **backColor** property of its parent and is unaffected by the skin.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinJadeMask Class

The **JadeSkinJadeMask** class holds the definition of a skin for **JadeMask** controls.

For details about the property defined in the **JadeSkinJadeMask** class, see "JadeSkinJadeMask Property", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinJadeMask Property

The property defined in the **JadeSkinJadeMask** class is summarized in the following table.

Property	Reference to the …
myButtonSkin	Skin used to draw any **JadeMask** controls that will be treated like a button (with no picture images defined)

myButtonSkin

Type: JadeSkinButton

Availability: Read or write at any time

The **myButtonSkin** property of the **JadeSkinJadeMask** class contains a reference to the image used to draw any **JadeMask** controls that are treated like a button (with no defined picture images).

If the value of this property is null, **JadeMask** controls treated like a button are not skinned.

JadeSkinJadeRichText Class

The **JadeSkinJadeRichText** class holds the definition of a skin for **JadeRichText** controls.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinLabel Class

The **JadeSkinLabel** class holds the definition of a skin for **Label** controls.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinListBox Class

The **JadeSkinListBox** class holds the definition of a skin for **ListBox** controls.

Note The images defined in these properties replace the equivalent image only if the standard JADE image has not been replaced in the list box.

For details about the properties defined in the **JadeSkinListBox** class, see "JadeSkinListBox Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinListBox Properties

The properties defined in the **JadeSkinListBox** class are summarized in the following table.

Property	Contains the …
alternatingRowBackColor	Specifies an alternate entry background color
alternatingRowBackColorCount	Specifies the number of visible list box entries at which the alternating background color of each visible entry is displayed
imgPictureClosed	Closed image that replaces a list box with the **hasPictures** or **hasPlusMinus** property set to **true**
imgPictureLeaf	Leaf image that replaces a list box with the **hasPictures** or **hasPlusMinus** property set to **true**
imgPictureMinus	Minus image that replaces a list box with the **hasPictures** or **hasPlusMinus** property set to **true**
imgPictureOpen	Open image that replaces a list box with the **hasPictures** or **hasPlusMinus** property set to **true**
imgPicturePlus	Plus image that replaces a list box with the **hasPictures** or **hasPlusMinus** property set to **true**
selectionColor	Specifies the background color of the selected item in a skinned list box
selectionColorText	Specifies the text color of the selected item in a skinned list box

alternatingRowBackColor

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColor** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies an alternate row background color. By default, alternating list box and table rows have a background color of **Azure**. When you set this property to a value other than **Azure**, the specified value is used as the default background color of each alternate non-fixed row.

If the value of the **alternatingRowBackColorCount** property is **2**, the first, third, and so on non-fixed row default background color is the **backColor** property value of the list box or table sheet. The second, fourth, and so on non-fixed row default background color is the **alternatingRowBackColor** property value when it is not the default value (otherwise the **backColor** property value of the list box or sheet is used).

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the entry or cell is ignored and the specific value of the **backColor** property is used.

Note When a list entry or cell is drawn, the **backColor** property value is overridden by any specified **backColor** value set for that list entry, cell, its row, or its column.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;
```

```
listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

alternatingRowBackColorCount

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColorCount** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the number of list box or table rows at which the alternating background color of each visible list entry row, non-fixed row, and non-fixed cell is displayed.

If the value of the **alternatingRowBackColorCount** property is:

- Less than or equal to zero (**0**), the background color of each list entry or non-fixed cell defaults to the value of the **backColor** property of the list box or sheet, or of the list box or table itself if the value of the sheet is not specifically set. The **alternatingRowBackColor** property value is ignored.
- Greater than zero (**0**), for each visible **alternatingRowBackColorCount** list entry, non-fixed row, and non-fixed cell, the background color defaults to the value of the **alternatingRowBackColor** property.

For example, if the count is **2**, the first, third, fifth, and so on, list box entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **backColor** property of the list box or sheet, while the second, fourth, sixth, and so on list entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **alternatingRowBackColor** property.

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the row or cell is ignored and the specific value of the **backColor** property is used.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;

listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

imgPictureClosed

Type: Binary

Availability: Read or write at any time

The **imgPictureClosed** property of the **JadeSkinListBox** class contains the closed image that replaces a list box with the **ListBox** class **hasPictures** or **hasPlusMinus** property set to **true**.

The defined image replaces the equivalent image only if the standard JADE image has not been replaced in the list box.

imgPictureLeaf

Type: Binary

Availability: Read or write at any time

The **imgPictureLeaf** property of the **JadeSkinListBox** class contains the leaf image that replaces a list box with the **ListBox** class **hasPictures** or **hasPlusMinus** property set to **true**.

The defined image replaces the equivalent image only if the standard JADE image has not been replaced in the list box.

imgPictureMinus

Type: Binary

Availability: Read or write at any time

The **imgPictureMinus** property of the **JadeSkinListBox** class contains the minus image that replaces a list box with the **ListBox** class **hasPictures** or **hasPlusMinus** property set to **true**.

The defined image replaces the equivalent image only if the standard JADE image has not been replaced in the list box.

imgPictureOpen

Type: Binary

Availability: Read or write at any time

The **imgPictureOpen** property of the **JadeSkinListBox** class contains the open image that replaces a list box with the **ListBox** class **hasPictures** or **hasPlusMinus** property set to **true**.

The defined image replaces the equivalent image only if the standard JADE image has not been replaced in the list box.

imgPicturePlus

Type: Binary

Availability: Read or write at any time

The **imgPicturePlus** property of the **JadeSkinListBox** class contains the plus image that replaces a list box with the **ListBox** class **hasPictures** or **hasPlusMinus** property set to **true**.

The defined image replaces the equivalent image only if the standard JADE image has not been replaced in the list box.

selectionColor

Type: Integer

Availability: Read or write at any time

The **selectionColor** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the background color that is used to draw a selected item in a skinned list box or table.

The default value of **#80000000** (that is, transparent) means that the default selection color defined by Windows is used.

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the selected list item, cell, row, or column is ignored and the specific value of the **backColor** property is used.

selectionColorText

Type: Integer

Availability: Read or write at any time

The **selectionColorText** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the text color of a selected item in a skinned list box or table.

The default value of **#80000000** (that is, transparent) means that the default selection color defined by Windows is used.

If the value of the **foreColor** property of a selected list entry, cell, row, or column is specifically set and it is not **#80000000**, the default value of the list item, cell, row, or column text is ignored and the specific value of the **foreColor** property is used.

JadeSkinOleControl Class

The **JadeSkinOleControl** class holds the definition of a skin for **OleControl** controls.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinOptionButton Class

The **JadeSkinOptionButton** class holds the definition of a skin for **OptionButton** controls. If a specific state is not supplied, the appropriate up image is used. If the up image is not supplied, the default option button image is drawn.

If the option button image of the skin is higher than the option button control using that skin, the option button control is enlarged in height to display the entire image.

For details about the properties defined in the **JadeSkinOptionButton** class, see "JadeSkinOptionButton Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinOptionButton Properties

The properties defined in the **JadeSkinOptionButton** class are summarized in the following table.

Property	Reference to the …
myFalseImage	Full image and up, disabled, down, and rollover states of the **false** value of option buttons
myTrueImage	Full image and up, disabled, down, and rollover states of the **true** value of option buttons

myFalseImage

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myFalseImage** property of the **JadeSkinOptionButton** class contains a reference to the full image and the up, disabled, down, and rollover states of the **false** value of **OptionButton** controls.

If a specific state is not supplied, the appropriate up image is used. If the up image is not supplied, the default option button image is drawn.

myTrueImage

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myTrueImage** property of the **JadeSkinOptionButton** class contains a reference to the full image and the up, disabled, down, and rollover states of the **true** value of **OptionButton** controls. If a specific state is not supplied, the appropriate up image is used. If the up image is not supplied, the default option button image is drawn.

JadeSkinPicture Class

The **JadeSkinPicture** class holds the definition of a skin for **Picture** controls.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinProgressBar Class

The **JadeSkinProgressBar** class holds the definition of a skin for **ProgressBar** controls. For details about the property defined in the **JadeSkinProgressBar** class, see "JadeSkinProgressBar Property", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinProgressBar Property

The property defined in the **JadeSkinProgressBar** class is summarized in the following table.

Property	Reference to the …
myProgressImage	Skin used to draw the completed progress part of **ProgressBar** controls.

myProgressImage

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myProgressImage** property of the **JadeSkinProgressBar** class contains a reference to the image used to draw the completed progress part of **ProgressBar** controls. The progress bar is initially drawn by using the **JadeSkinControl** definition for the border and incomplete parts of the progress bar.

The progress percentage portion is then drawn by using the **myProgressImage** property definition. If the value of the **myProgressImage** property is **null**, the progress percentage portion is drawn by using the value of the **foreColor** property of the **JadeSkinControl** superclass. The percentage text is drawn centered in the control, using the value of the **backColor** property for any part of the text over the progress percentage portion and the value of the **foreColor** property for any text over the other area.

JadeSkinScrollBar Class

The **JadeSkinScrollBar** class is the abstract class that contains information common to vertical and horizontal **ScrollBar** controls. For details about the properties defined in the **JadeSkinScrollBar** class, see "JadeSkinScrollBar Properties", in the following section.

Inherits From: JadeSkinControl

Inherited By: JadeSkinHScroll, JadeSkinVScroll

JadeSkinScrollBar Properties

The properties defined in the **JadeSkinScrollBar** class are summarized in the following table.

Property	Description
imgHighLightBrush	Contains the image for the brush used when the user clicks on the scroll bar stem itself (that is, *not* the thumb track or the arrows).
myThumbTrack	Reference to the **JadeSkinWindowStateImage** object that defines the thumb track in the up position.
myThumbTrackDisabled	Reference to the **JadeSkinWindowStateImage** object that defines the disabled thumb track.
myThumbTrackDown	Reference to the **JadeSkinWindowStateImage** object that defines the clicked thumb track.
myThumbTrackRollOver	Reference to the **JadeSkinWindowStateImage** object that defines how to the thumb track in the rollover state.

imgHighLightBrush

Type: Binary

Availability: Read or write at any time

The **imgHighLightBrush** property of the **JadeSkinScrollBar** class contains the image for the brush used when the user clicks on the scroll bar stem itself (that is, *not* the thumb track or the arrows).

When the mouse is down in this situation, that portion of the scroll bar is highlighted. If you supply this brush image, highlighting is drawn using this brush, or it is drawn with a black brush if you do not supply a highlight brush.

myThumbTrack

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myThumbTrack** property of the **JadeSkinScrollBar** class contains a reference to the **JadeSkinWindowStateImage** object that defines the thumb track in the up position. If this property has a null value, the thumb track is drawn as normal.

myThumbTrackDisabled

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myThumbTrackDisabled** property of the **JadeSkinScrollBar** class contains a reference to the **JadeSkinWindowStateImage** object that defines the disabled thumb track.

If this property has a null value, the thumb track is drawn using the value of the **myThumbTrack** property.

myThumbTrackDown

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myThumbTrackDown** property of the **JadeSkinScrollBar** class contains a reference to the **JadeSkinWindowStateImage** object that defines the clicked thumb track.

If this property has a null value, the thumb track is drawn using the value of the **myThumbTrack** property.

myThumbTrackRollOver

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myThumbTrackRollOver** property of the **JadeSkinScrollBar** class contains a reference to the **JadeSkinWindowStateImage** object that defines how to draw the thumb track in the rollover state.

If this property has a null value, the thumb track is drawn using the value of the **myThumbTrack** property.

JadeSkinSheet Class

The **JadeSkinSheet** class holds the definition of a skin for **Sheet** controls.

For details about the property defined in the **JadeSkinSheet** class, see "JadeSkinSheet Property", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinSheet Property

The property defined in the **JadeSkinSheet** class is summarized in the following table.

Property	Description
myTabButton	Contains a reference to the skin used to draw the tab of a folder.

myTabButton

Type: JadeSkinButton

Availability: Read or write at any time

The **myTabButton** property of the **JadeSkinSheet** class contains a reference to the skin used to draw the tab of a folder for a sheet that has the **tabsStyle** property set to **TabsStyle_Buttons** (1).

If this property is null, any skin button of the folder is used instead. If this property is not null, any skin button setting of the folder is ignored.

Note The size of the tab area of a folder is defined by using the **JadeSkinFolder** skin type. Any tab defined for a sheet has no impact on the determination of the tab height.

The main use of defining a button skin for a sheet is to enable each tab of a folder to be drawn with different images and colors. To achieve this, define several sheet skins with different categories and then set the **Window**::**skinCategoryName** property on the sheets that you want to use each specific sheet skin.

JadeSkinStatusLine Class

The **JadeSkinStatusLine** class holds the definition of a skin for **StatusLine** controls.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinTable Class

The **JadeSkinTable** class holds the definition of a skin for **Table** controls.

For details about the properties defined in the **JadeSkinTable** class, see "JadeSkinTable Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinTable Properties

The properties defined in the **JadeSkinTable** class are summarized in the following table.

Property	Description
alternatingRowBackColor	Specifies an alternate row background color

Property	Description
alternatingRowBackColorCount	Specifies the number of table rows at which the alternating background color of each visible non-fixed row and non-fixed cell is displayed
fixed3D	Specifies whether the table fixed cells are drawn in three-dimensional style
fixedColumnsBackColor	Specifies the color with which the background of fixed columns is drawn
fixedRowColorHasPrecedence	Specifies whether cells that are in both a fixed row and a fixed column are drawn using the value of the **fixedRowsBackColor** property
fixedRowsBackColor	Specifies the color with which the background of fixed rows is drawn
myCheckBoxSkin	Specifies the check box skin that is used when drawing a cell that has the **Table** class **inputType** property set to **InputType_CheckBox** or a cell control set to a **CheckBox** control
selectionColor	Specifies the background color of the selected item in a skinned table
selectionColorText	Specifies the text color of the selected item in a skinned table
tabActiveColor	Contains the color used to draw the background area of the active tab.
tabInactiveColor	Contains the color used to draw the background area of inactive tabs.

alternatingRowBackColor

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColor** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies an alternate row background color. By default, alternating list box and table rows have a background color of **Azure**. When you set this property to a value other than **Azure**, the specified value is used as the default background color of each alternate non-fixed row.

If the value of the **alternatingRowBackColorCount** property is **2**, the first, third, and so on non-fixed row default background color is the **backColor** property value of the list box or table sheet. The second, fourth, and so on non-fixed row default background color is the **alternatingRowBackColor** property value when it is not the default value (otherwise the **backColor** property value of the list box or sheet is used).

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the entry or cell is ignored and the specific value of the **backColor** property is used.

Note When a list entry or cell is drawn, the **backColor** property value is overridden by any specified **backColor** value set for that list entry, cell, its row, or its column.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;

listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

alternatingRowBackColorCount

Type: Integer

Availability: Read or write at any time

The **alternatingRowBackColorCount** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the number of list box or table rows at which the alternating background color of each visible list entry row, non-fixed row, and non-fixed cell is displayed.

If the value of the **alternatingRowBackColorCount** property is:

- Less than or equal to zero (**0**), the background color of each list entry or non-fixed cell defaults to the value of the **backColor** property of the list box or sheet, or of the list box or table itself if the value of the sheet is not specifically set. The **alternatingRowBackColor** property value is ignored.
- Greater than zero (**0**), for each visible **alternatingRowBackColorCount** list entry, non-fixed row, and non-fixed cell, the background color defaults to the value of the **alternatingRowBackColor** property.

For example, if the count is **2**, the first, third, fifth, and so on, list box entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **backColor** property of the list box or sheet, while the second, fourth, sixth, and so on list entries, non-fixed rows, and the non-fixed cells in that row default to the value of the **alternatingRowBackColor** property.

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the row or cell is ignored and the specific value of the **backColor** property is used.

Note that when the list box or table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;

listbox1.alternatingRowBackColorCount := 3;
listbox1.alternatingRowBackColor := DarkGray;
```

fixed3D

Type: Integer

Availability: Read or write at any time

The **fixed3D** property of the **JadeSkinTable** class specifies whether the skinned table fixed cells are drawn in three-dimensional style. The property can be set to one of the following values.

Integer Value	Description
0 (the default)	A table that uses this skin does not draw the fixed cells as 3D elements. (The value of the **Table** class **fixed3D** property value is ignored.)
1 (true)	A table that uses this skin draws the fixed cells as 3D elements. (The value of the **Table** class **fixed3D** property is ignored.)
2 (use the **Table** control value)	Ignores this skin property and uses the value of the **Table** class **fixed3D** property.

fixedColumnsBackColor

Type: Integer

Availability: Read or write at any time

The **fixedColumnsBackColor** property of the **JadeSkinTable** class specifies the color with which the background of fixed columns is drawn.

The default value of **#80000000** means that this property is ignored. The background color of fixed columns for any other value is drawn using the specified value of this property.

Note If a fixed cell has a specific background color set via a cell, row, column **Table** class **backColor** property, the skin background color value specified by this parameter is ignored.

fixedRowColorHasPrecedence

Type: Boolean

Availability: Read or write at any time

The **fixedRowColorHasPrecedence** property of the **JadeSkinTable** class specifies whether cells that are in both a fixed row and a fixed column are drawn using the value of the **fixedRowsBackColor** property. The default value is **true**.

When **true**, cells that are in both a fixed row and a fixed column are drawn using the value of the **fixedRowsBackColor** property. When **false**, cells that are in both a fixed row and a fixed column are drawn using the value of the **fixedColumnsBackColor** property.

Note If a fixed cell has a specific background color set via a cell, row, column **Table** class **backColor** property, the skin background color value specified by this parameter is ignored.

fixedRowsBackColor

Type: Integer

Availability: Read or write at any time

The **fixedRowsBackColor** property of the **JadeSkinTable** class specifies the color with which the background of fixed rows is drawn.

The default value of **#80000000** means that this property is ignored. The background color of fixed rows for any other value is drawn using the specified value of this property.

Note If a fixed cell has a specific background color set via a cell, row, column **Table** class **backColor** property, the skin background color value specified by this parameter is ignored.

myCheckBoxSkin

Type: JadeSkinCheckBox

Availability: Read or write at any time

The **myCheckBoxSkin** property of the **JadeSkinTable** class specifies the check box skin that is used when drawing a cell that has the **Table** class **inputType** property set to **InputType_CheckBox** or a cell control set to a **CheckBox** control.

The value for a check box skin is defined in the **Table** control types on the **Controls** sheet of the Jade Skin Maintenance dialog. The default value of **<none>** indicates that check boxes in table cells are drawn without a skin.

selectionColor

Type: Integer

Availability: Read or write at any time

The **selectionColor** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the background color that is used to draw a selected item in a skinned list box or table.

The default value of **#80000000** (that is, transparent) means that the default selection color defined by Windows is used.

If the value of the **backColor** property of a list entry, cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the selected list item, cell, row, or column is ignored and the specific value of the **backColor** property is used.

selectionColorText

Type: Integer

Availability: Read or write at any time

The **selectionColorText** property of the **JadeSkinListBox** class and **JadeSkinTable** class specifies the text color of a selected item in a skinned list box or table.

The default value of **#80000000** (that is, transparent) means that the default selection color defined by Windows is used.

If the value of the **foreColor** property of a selected list entry, cell, row, or column is specifically set and it is not **#80000000**, the default value of the list item, cell, row, or column text is ignored and the specific value of the **foreColor** property is used.

tabActiveColor

Type: Integer

Availability: Read or write at any time

The **tabActiveColor** property of the **JadeSkinTable** class contains the color used to draw the background area of the active tab on **Table** controls. JADE uses the RGB scheme for colors.

The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that the normal color of the table is used.

tabInactiveColor

Type: Integer

Availability: Read or write at any time

The **tabInactiveColor** property of the **JadeSkinTable** class contains the color used to draw the background area of the inactive tabs on **Table** controls. JADE uses the RGB scheme for colors.

The default value specified by the **JadeSkinEntity** class **Default_Color** constant indicates that the normal color of the table is used.

JadeSkinTextBox Class

The **JadeSkinTextBox** class holds the definition of a skin for **TextBox** controls.

Note Text box controls do not successfully handle back brushes and non-solid colors for the **backColor** property.

For details about the properties defined in the **JadeSkinTextBox** class, see "JadeSkinTextBox Properties", in the following subsection.

Inherits From: JadeSkinControl

Inherited By: (None)

JadeSkinTextBox Properties

The properties defined in the **JadeSkinTextBox** class are summarized in the following table.

Property	Description
hintBackColor	Specifies the background color of hint text
hintForeColor	Specifies the foreground color of hint text

hintBackColor

Type: Integer

Availability: Read or write at any time

The **hintBackColor** property of the **JadeSkinTextBox** class specifies the color with which the text box background is displayed when hint text is displayed. (JADE uses the RGB scheme for colors.)

The default value of **#80000000** means that the property is ignored and the defined **TextBox** class **hintBackColor** property value is used.

This value applies only if the **TextBox** class **hintText** property is not null (**""**), the hint text is displayed (that is, the text box is empty), and the **JadeSkinTextBox** class hint value is not **#80000000** (the default).

The property value is also ignored when the hint text is displayed and the text box text is disabled, in which case the text box is drawn in its disabled state (with the hint text still displayed).

For details about the **TextBox** class hint background color, see the **TextBox** class **hintBackColor** property in Volume 3 of the *JADE Encyclopedia of Classes*.

hintForeColor

Type: Integer

Availability: Read or write at any time

The **hintForeColor** property of the **JadeSkinTextBox** class specifies the color with which the text box foreground (text) is displayed when hint text is displayed. (JADE uses the RGB scheme for colors.)

The default value of **#80000000** means that the property is ignored and the defined **TextBox** class **hintForeColor** property value is used.

This value applies only if the **TextBox** class **hintText** property is not null (**""**), the hint text is displayed (that is, the text box is empty), and the **JadeSkinTextBox** class hint value is not **#80000000** (the default).

The property value is also ignored when the hint text is displayed and the text box text is disabled, in which case the text box is drawn in its disabled state (with the hint text still displayed).

For details about the **TextBox** class hint foreground color, see the **TextBox** class **hintForeColor** property in Volume 3 of the *JADE Encyclopedia of Classes*.

JadeSkinVScroll Class

The **JadeSkinVScroll** class holds the definition of a skin for **VScroll** subclasses of the **ScrollBar** control. The width of the border area and the width of the **myTopButton.imgUp** image determine the width of the scroll bar.

For a scroll bar control, the image is stretched horizontally to fit the area inside the borders.

Note This skin is also used for drawing the vertical scroll bar for any control or form in the application.

If you do not supply the image for a specific state, the appropriate up image is used. The default scroll button image is drawn if you do not supply the up image.

For details about the properties defined in the **JadeSkinVScroll** class, see "JadeSkinHScroll Properties", in the following subsection.

Inherits From: JadeSkinScrollBar

Inherited By: (None)

JadeSkinVScroll Properties

The properties defined in the **JadeSkinVScroll** class are summarized in the following table.

Property	Reference to the image used to draw the …
myBottomButton	Top button of a vertical scroll bar in its various states (the button is placed inside the scroll bar borders)
myTopButton	Bottom button of a vertical scroll bar in its various states (the button is placed inside the scroll bar borders)

myBottomButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myBottomButton** property of the **JadeSkinVScroll** class contains a reference to the image used to draw the bottom button of a vertical scroll bar in its up, disabled, down, and rollover states. (The button is placed inside the scroll bar borders.)

myTopButton

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myTopButton** property of the **JadeSkinVScroll** class contains a reference to the image used to draw the top button of a vertical scroll bar in its up, disabled, down, and rollover states. (The button is placed inside the scroll bar borders.)

JadeSkinEntity Class

The **JadeSkinEntity** class is the abstract superclass of the skin entities.

The **JadeSkinEntity** class contains the JADE skins defined for your applications, forms, controls, and menus, and encapsulates the behavior required to define and maintain JADE skins using the **JadeSkinMaintenance** and **JadeSkinSelection** forms provided by the JADE RootSchema. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of your *JADE Runtime Application Guide*.

Note If an application is active with a skin set, a second application initiated from the same **jade.exe** executable is now drawn using the current Windows theme if that application does not have a skin set.

For details about the constant and properties defined in the **JadeSkinEntity** class, see "JadeSkinEntity Class Constant" and "JadeSkinEntity Properties", in the following subsections.

Inherits From: Object

Inherited By: JadeSkinApplication, JadeSkinArea, JadeSkinCategory, JadeSkinSimpleButton

JadeSkinEntity Class Constant

The **JadeSkinEntity** class provides the constant listed in the following table.

JadeSkinEntity Class Constant	Integer Value	Description
Default_Color	#80000000	Color is not set

JadeSkinEntity Properties

The **JadeSkinEntity** class provides the properties summarized in the following table.

Property	Description
description	Text that can be used for documentation purposes
myOwners	Automatic collection of other skin entities that reference this object
mySkinRoot	Inverse reference to the instance of the **JadeSkinRoot** class
name	Name used to identify the skin entity

description

Type: String

Availability: Read or write at any time

The **description** property of the **JadeSkinEntity** class contains the text that can be used for documentation purposes for the skin.

myOwners

Type: JadeSkinEntityNameDict

Availability: Read or write at any time

The **myOwners** property of the **JadeSkinEntity** class contains a reference to the automatic collection of other skin entities that reference this object.

mySkinRoot

Type: JadeSkinRoot

Availability: Read or write at any time

The **mySkinRoot** property of the **JadeSkinEntity** class contains an inverse reference to the instance of the **JadeSkinRoot** class.

name

Type: String

Availability: Read or write at any time

The **name** property of the **JadeSkinEntity** class contains the name used to identify the skin entity.

JadeSkinForm Class

The **JadeSkinForm** class contains the JADE skins defined for forms in your applications and encapsulates the behavior required to define and maintain JADE skins using the **JadeSkinMaintenance** and **JadeSkinSelection** forms provided by the JADE RootSchema.

Use the **JadeSkinArea** class to define the image drawn for the active border and the inner (client area) of the form. If the form has a backdrop picture (set by using the **Form** class **setBackDrop** method), this image is drawn.

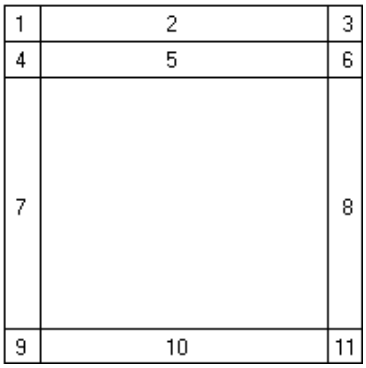
When drawing an inactive form image, if an image is not provided for a border segment, the equivalent image for the active form is drawn instead.

When erasing the inner area of the form:

1. If the form **backBrush** property is not null, the inner area is erased using that brush.
2. If the form skin has a **JadeSkinArea** class **imgInner** property value that is a brush, the inner area is erased using that brush.
3. If the form **backColor** property is not **Color_3DFace**, the inner area is erased using the background color of the form.
4. If the **backColor** property of the skin area is not **Default_Color**, erase using the **backColor** property value of the skin.
5. The **Color_3DFace** value is used when erasing.
6. If the form was erased using a color and the skin of the form has a **JadeSkinArea** class **imgInner** property value that is not a brush, that image is drawn centered in the inner area of the form skin.

The set of skin images used by JADE is provided with the product release so that you can use these skins in your applications, if required. (By default, skins are not used.)

The form border for a skin is made up of 11 images, as shown in the following image.



The following is a description of the form border areas.

- Images 1, 3, 4, 6, 9, and 11 are shown at actual size.
- Images 2, 5, 7, 8, and 10 are stretched to fit the width or height of the form.
- Images 1, 2, and 3 must have the same height to enable the form to display correctly.
- Images 4, 5, and 6 must have the same height to enable the form to display correctly.

- Images 9, 10, and 11 must have the same height to enable the form to display correctly.
- The whole of image 1 is treated as the control menu area for the form.

  If the menu does not fit on the menu line, the menu is extended to include additional lines, as required. Each line is drawn with the same skin images as the first menu line.
- When an MDI child is maximized, the whole of image 4 is treated as the system menu area for the MDI child.
- Form icons are placed adjacently at the top right hand edge of the area defined by image 3.
- MDI child form icons are placed adjacently at the top right hand edge of the area defined by image 6.
- Form icons that are disabled are not displayed if there is no disabled image.

The following areas are not affected by using a skin.

- Only JADE forms adopt the skin presentation. Windows forms such as message boxes, common dialogs, and the JADE exception dialogs are unchanged.
- When a form is resized, Windows draws the standard form image while the resize is occurring.
- A minimized MDI form displays the standard image, as there is normally insufficient room to display the skinned image for that short caption line.
- Windows-drawn menu items are unchanged by the skin. This includes the system menus.
- Any changes made to the skin do not affect any current users of that skin.
- Windows sounds do not occur when forms are minimized, maximized, and so on, as the form buttons are not in the Windows standard positions and their actions must be performed programmatically by JADE. (Windows does not issue those sounds when such actions are performed programmatically.)

For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*. For details about using JADE skins in your JADE development environment, see "Specifying Your JADE Installation Preferences", in Chapter 2 of the *JADE Installation and Configuration Guide*. For details about the properties defined in the **JadeSkinForm** class, see "JadeSkinForm Properties", in the following subsection.

Inherits From: JadeSkinWindow

Inherited By: (None)

JadeSkinForm Properties

The **JadeSkinForm** class provides the properties summarized in the following table.

Property	Description
captionActiveForeColor	Color used to draw caption text when the form is active.
captionFontBold	Specifies whether the caption of the form is bold.
captionFontItalic	Specifies whether the caption of the form is italics.
captionFontName	Font with which the form caption is displayed.
captionFontSize	Size of the font with which the form caption is displayed.
captionInactiveForeColor	Color used to draw the caption text of the form when the form is inactive.

Property	Description
captionLeft	Starting left position of the caption text of the form.
captionTop	Starting right position of the caption text of the form.
centerCaption	Specifies whether the caption is centered within the top strip area of the skin of the form.
drawMenuSelectionFlat	Specifies how the menu line items are drawn when the skinned menu line item is selected.
imgInactiveBorderBottomLeft	Border area image to be drawn for the bottom left of the inactive form.
imgInactiveBorderBottomRight	Border area image to be drawn for the bottom right of the inactive form.
imgInactiveBorderBottomStrip	Border area image to be drawn for the bottom strip of the inactive form.
imgInactiveBorderLeftStrip	Border area image to be drawn for the left strip of the inactive form.
imgInactiveBorderRightStrip	Border area image to be drawn for the right strip of the inactive form.
imgInactiveBorderTopLeft	Border area image to be drawn for the top left of the inactive form.
imgInactiveBorderTopRight	Border area image to be drawn for the top right of the inactive form.
imgInactiveBorderTopStrip	Border area image to be drawn for the top strip of the inactive form.
imgMenuLeft	Image drawn for the left of the menu line of the form.
imgMenuRight	Image drawn for the right of the menu line of the form.
imgMenuStrip	Image drawn for the strip of the menu line of the form.
menuBackColor	Background color for the menu line if the skin has no defined **imgMenuStrip** property value.
menuBackColorSelected	Background color of the border drawn around the selected menu using the Windows three-dimensional colors or the default color used to draw the background area for selected menu items for drop-down or popup menus.
menuFontBold	Specifies whether menu line item, drop-down menu, and popup menu captions are bold.
menuFontItalic	Specifies whether menu line item, drop-down menu, and popup menu captions are italics.
menuFontName	Font with which menu line items, drop-down menus, and popup menus are displayed.
menuFontSize	Size of the font with which menu line items, drop-down menus, and popup menus are displayed.
menuForeColor	Color used to draw the text for non-selected and enabled menu line items.
menuForeColorDisabled	Color used to draw the text for disabled menu line items and the default disabled text color for drop-down and popup menus.
menuForeColorSelected	Default selected text color for drop-down and popup menus.
menuLeftPosition	Starting left position of the form menus.
menuTopPosition	Offset of the top position of the form menu drawn on the skin.
myChildMinimizeBtn	Reference to the simple button images drawn for an MDI child minimize button in its four states (up, down, rollover, and disabled).

Property	Description
myChildRestoreBtn	Reference to the simple button images drawn for an MDI child restore button in its four states.
myChildTerminateBtn	Reference to the simple button images drawn for an MDI child terminate button in its four states.
myMaximizeBtn	Reference to the simple button images drawn for an MDI child maximize button in its four states.
myMaximizedBtn	Reference to the simple button images drawn for an MDI child maximized button in its four states.
myMenuSkin	Reference to the menu definition of the form.
myMinimizeBtn	Reference to the simple button images to be drawn for the form minimize button in its four states.
myTerminateBtn	Reference to the simple button images to be drawn for the form terminate button in its four states.
showMenuLineAlways	Specifies whether the menu line of the skin is always drawn, regardless of whether the form has a menu.
transparentColorForButtons	Transparent color to be applied to maximize, minimize, and terminate buttons drawn for the skin of the form.
useMenuLineSkinForMenus	Specifies whether the menu line definition of a form skin is used to draw menus when the value of the **myMenuSkin** property is null.

captionActiveForeColor

Type: Integer

Availability: Read or write at any time

The **captionActiveForeColor** property of the **JadeSkinForm** class contains the foreground color used to draw the caption text on active forms.

JADE uses the RGB scheme for colors.

The **JadeSkinEntity** class **Default_Color** constant default value for this property indicates that no color is set in the form skin, and the desktop active caption text color defined by the user is used.

captionFontBold

Type: Boolean

Availability: Read or write at any time

The **captionFontBold** property of the **JadeSkinForm** class specifies whether the font style of the form skin text is bold.

Use this property to format text in a form skin, either in the JADE development environment or at run time by using logic.

The settings for the **captionFontBold** property are listed in the following table.

Value	Description
true	Turns on the bold formatting (the default)
false	Turns off the bold formatting

If the **captionFontName** property is null (the default), the form caption is drawn using the desktop active caption font defined by the user.

captionFontItalic

Type: Boolean

Availability: Read or write at any time

The **captionFontItalic** property of the **JadeSkinForm** class specifies whether the font style of the form skin text is italics.

Use this property to format text in a form skin, either in the JADE development environment or at run time by using logic.

The settings for the **captionFontItalic** property are listed in the following table.

Value	Description
true	Turns on the italic formatting
false	Turns off the italic formatting (the default)

If the **captionFontName** property is null (the default), the form caption is drawn using the desktop active caption font defined by the user.

captionFontName

Type: String[31]

Availability: Read or write at any time

The **captionFontName** property of the **JadeSkinForm** class contains the name of the font used to display text on form skins.

Use this property to format text in a form skin, either in the JADE development environment or at run time by using logic.

The default value is **Tahoma**.

Note The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

captionFontSize

Type: Real

Availability: Read or write at any time

The **captionFontSize** property of the **JadeSkinForm** class contains the size of the font used for text displayed on a form skin.

Use this property to format text in a form skin, either in the JADE development environment or at run time by using logic.

The default value for the **captionFontSize** property is **8.25**.

If the **captionFontName** property is null (the default), the form caption is drawn using the desktop active caption font defined by the user.

captionInactiveForeColor

Type: Integer

Availability: Read or write at any time

The **captionInactiveForeColor** property of the **JadeSkinForm** class contains the foreground color used to draw the caption text on inactive forms. JADE uses the RGB scheme for colors.

The **JadeSkinEntity** class **Default_Color** constant default value for this property indicates that no color is set in the skin and the text color of the desktop inactive caption defined by the user is used.

captionLeft

Type: Integer

Availability: Read or write at any time

The **captionLeft** property of the **JadeSkinForm** class contains the starting left position of the caption text on the form skin. The default value is zero (**0**).

This property is ignored if the value of the **centerCaption** property is **true** unless the caption cannot fit within the top strip area of the form skin.

captionTop

Type: Integer

Availability: Read or write at any time

The **captionTop** property of the **JadeSkinForm** class contains the top position where the caption is drawn on the form skin. The default value is zero (**0**).

The form caption top position is adjusted downwards at run time if the value is greater than zero (**0**) and the caption text will overflow off the bottom of the caption area of the form's skin.

Under different workstation font setups, the height of the caption text can be larger than the image that defines the height of the caption.

centerCaption

Type: Boolean

Availability: Read or write at any time

The **centerCaption** property of the **JadeSkinForm** class specifies whether the caption is centered within the top strip area of the form skin. The default value is **false**.

If the caption cannot fit within this top strip area, the value of this property is ignored.

drawMenuSelectionFlat

Type: Boolean

Availability: Read or write at any time

The **drawMenuSelectionFlat** property of the **JadeSkinForm** class specifies how the menu line items are drawn when the skinned menu line item is selected. If the value is:

- **False** (the default), a selected menu line item is drawn using the effective menu item selection background and foreground (text) colors with a border that is the same color as the text of the menu item.

  In addition, when a menu is dropped down directly below a menu item (the menu does not have to be moved to fit on the current display), the selected menu item is drawn as though it is part of the dropped-down menu, using the same background color and text color defined for the popup menu.

- **True**, the selected menu line item is always drawn in a flat manner using the effective menu item selection background and foreground (text) colors with no surrounding border.

  In addition, the menu line item is not drawn as though it is part of the popup menu.

Control the **drawMenuSelectionFlat** property with the **Draw Flat Selection** check box on the **Forms** sheet of the Jade Skin Maintenance dialog.

imgInactiveBorderBottomLeft

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderBottomLeft** property of the **JadeSkinForm** class contains the bottom left border area for inactive form skins. This image is drawn unstretched.

When drawing an inactive form image, if you do not supply the bottom left border image, the equivalent image for the active form is drawn instead.

imgInactiveBorderBottomRight

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderBottomRight** property of the **JadeSkinForm** class contains the bottom right border area for inactive form skins. This image is drawn unstretched.

When drawing an inactive form image, if you do not supply the bottom right border image, the equivalent image for the active form is drawn instead.

imgInactiveBorderBottomStrip

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderBottomStrip** property of the **JadeSkinForm** class contains the bottom strip for inactive form skins. This image is drawn stretched.

When drawing an inactive form image, if you do not supply the bottom border strip image, the equivalent image for the active form is drawn instead.

imgInactiveBorderLeftStrip

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderLeftStrip** property of the **JadeSkinForm** class contains the left strip for inactive form skins. This image is drawn stretched.

When drawing an inactive form image, if you do not supply the left border strip image, the equivalent image for the active form is drawn instead.

imgInactiveBorderRightStrip

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderRightStrip** property of the **JadeSkinForm** class contains the right strip for inactive form skins. This image is drawn stretched.

When drawing an inactive form image, if you do not supply the right border strip image, the equivalent image for the active form is drawn instead.

imgInactiveBorderTopLeft

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderTopLeft** property of the **JadeSkinForm** class contains the top left border area for inactive form skins. This image is drawn unstretched.

When drawing an inactive form image, if you do not supply the top left border image, the equivalent image for the active form is drawn instead.

imgInactiveBorderTopRight

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderTopRight** property of the **JadeSkinForm** class contains the top right border area for inactive form skins. This image is drawn unstretched.

When drawing an inactive form image, if you do not supply the top right border image, the equivalent image for the active form is drawn instead.

imgInactiveBorderTopStrip

Type: Binary

Availability: Read or write at any time

The **imgInactiveBorderTopStrip** property of the **JadeSkinForm** class contains the top strip for inactive form skins. This image is drawn stretched.

When drawing an inactive form image, if you do not supply the top border strip image, the equivalent image for the active form is drawn instead.

imgMenuLeft

Type: Binary

Availability: Read or write at any time

The **imgMenuLeft** property of the **JadeSkinForm** class contains the left menu area for form skins. This image is drawn unstretched.

imgMenuRight

Type: Binary

Availability: Read or write at any time

The **imgMenuRight** property of the **JadeSkinForm** class contains the right menu area for form skins. This image is drawn unstretched.

imgMenuStrip

Type: Binary

Availability: Read or write at any time

The **imgMenuStrip** property of the **JadeSkinForm** class contains the menu strip area for form skins. This image is drawn stretched.

menuBackColor

Type: Integer

Availability: Read or write at any time

The **menuBackColor** property of the **JadeSkinForm** class contains the background color for the menu line if the skin has no defined **imgMenuStrip** property value.

This property also contains the default color used to draw the background of non-selected and enabled drop-down and popup menu items.

The default value of **Default_Color** indicates that the default color of the window is used.

menuBackColorSelected

Type: Integer

Availability: Read or write at any time

The **menuBackColorSelected** property of the **JadeSkinForm** class contains the default color used to draw the background area for selected menu items for drop-down and popup (context) menus.

The default value of **Default_Color** indicates that the default color of the window is used.

menuFontBold

Type: Boolean

Availability: Read or write at any time

The **menuFontBold** property of the **JadeSkinForm** class specifies whether the bold font attribute is applied to menu line item, drop-down menu, and popup menu captions. The default value is **false**.

menuFontItalic

Type: Boolean

Availability: Read or write at any time

The **menuFontItalic** property of the **JadeSkinForm** class specifies whether the italic font attribute is applied to menu line item, drop-down menu, and popup menu captions. The default value is **false**.

menuFontName

Type: String

Availability: Read or write at any time

The **menuFontName** property of the **JadeSkinForm** class contains the name of the font with which menu line items, drop-down menus, and popup menus are displayed. This property is set to null (**""**) by default, and the standard menu font is used.

menuFontSize

Type: Real

Availability: Read or write at any time

The **menuFontSize** property of the **JadeSkinForm** class contains the size of the font with which menu line items, drop-down menus, and popup menus are displayed. The default value is zero (**0**).

menuForeColor

Type: Integer

Availability: Read or write at any time

The **menuForeColor** property of the **JadeSkinForm** class contains the color used to draw the text for non-selected and enabled menu line items and the default text color of any non-selected and enabled drop-down and popup menus.

The default value of **Default_Color** indicates that the Windows-defined menu text color is used.

menuForeColorDisabled

Type: Integer

Availability: Read or write at any time

The **menuForeColorDisabled** property of the **JadeSkinForm** class contains the color used to draw the text for disabled menu line items and the default disabled text color for drop-down and popup menus.

The default value of **Default_Color** indicates that the Windows-defined disabled menu text color is used.

menuForeColorSelected

Type: Integer

Availability: Read or write at any time

The **menuForeColorSelected** property of the **JadeSkinForm** class contains the default selected text color for drop-down and popup menus.

The default value of **Default_Color** indicates that the Windows-defined selected menu text color is used.

menuLeftPosition

Type: Integer

Availability: Read or write at any time

The **menuLeftPosition** property of the **JadeSkinForm** class contains the starting left position of menus on the form skin. The default value is zero (**0**).

menuTopPosition

Type: Integer

Availability: Read or write at any time

The **menuTopPosition** property of the **JadeSkinForm** class contains the starting top position of menus on the form skin. The default value is zero (**0**).

The menu top position is adjusted downwards at run time if the value is greater than zero (**0**) and the menu text will overflow off the bottom of the form menu area.

Under different workstation font set-ups, the height of the menu text can be larger than the image that defines the height of the menu.

myChildMinimizeBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myChildMinimizeBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for an MDI child minimize button in its four states (that is, up, down, rollover, and disabled).

If this reference is null or the up image is null, the default minimize button is drawn. If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myChildRestoreBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myChildRestoreBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for an MDI child restore button in its four states (that is, up, down, rollover, and disabled).

If this reference is null or the up image is null, the default restore button is drawn. If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myChildTerminateBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myChildTerminateBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for an MDI child terminate button in its four states (that is, up, down, rollover, and disabled).

If this reference is null or the up image is null, the default terminate button is drawn. If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myMaximizeBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myMaximizeBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for a form maximize button in its four states (that is, up, down, rollover, and disabled).

If this reference is null or the up image is null, the default maximize button is drawn. If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myMaximizedBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myMaximizedBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for a form maximized button in its four states (that is, up, down, rollover, and disabled).

If this reference is null or the up image is null, the default maximized button is drawn. If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myMenuSkin

Type: JadeSkinMenu

Availability: Read or write at any time

The **myMenuSkin** property of the **JadeSkinForm** class contains a reference to the menu definition that applies to the form for drop-down and popup menus. If this property is null, drop-down and popup menus use the **JadeSkinForm** class menu properties to draw the menu.

myMinimizeBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myMinimizeBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for a form minimize button in its four states (that is, up, down, rollover, and disabled). If this reference is null or the up image is null, the default minimize button is drawn.

If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

myTerminateBtn

Type: JadeSkinSimpleButton

Availability: Read or write at any time

The **myTerminateBtn** property of the **JadeSkinForm** class contains a reference to the simple button images drawn for a form terminate button in its four states (that is, up, down, rollover, and disabled). If this reference is null or the up image is null, the default terminate button is drawn.

If the down or rollover states are not provided, the up image is drawn. If the disabled image is not provided, the button is not drawn when it is disabled.

showMenuLineAlways

Type: Boolean

Availability: Read or write at any time

The **showMenuLineAlways** property of the **JadeSkinForm** class specifies whether the menu line of the skin is always drawn, regardless of whether the form has a menu.

The default value of **false** indicates that the menu line is not drawn if the form does not have a menu or if the form is an MDI child.

transparentColorForButtons

Type: Integer

Availability: Read or write at any time

The **transparentColorForButtons** property of the **JadeSkinForm** class contains the transparent color applied to maximize, minimize, and terminate buttons drawn for the form skin.

JADE uses the RGB scheme for colors.

The **JadeSkinEntity** class **Default_Color** constant default value for this property indicates that no transparent color is set.

useMenuLineSkinForMenus

Type: Boolean

Availability: Read or write at any time

The **useMenuLineSkinForMenus** property of the **JadeSkinForm** class specifies whether the menu line definition of a form skin is used to draw menus when the value of the **myMenuSkin** property is null (**""**).

If the value of the **useMenuLineSkinForMenus** property is:

- **False**and the value of the **myMenuSkin** property is null (**""**), menus are not skinned.
- **True** and the value of the **myMenuSkin** property is null (**""**, menus are drawn using the menu line properties (that is, font, colors, and so on).

Control the **useMenuLineSkinForMenus** property with the **Use Menu Line Options For Menus** check box on the **Forms** sheet of the Jade Skin Maintenance dialog. If a popup menu skin is set for the form, the **Use Menu Line Options For Menus** check box is set to **false** (that is, unchecked) and disabled.

JadeSkinMenu Class

The **JadeSkinMenu** class holds the skin definitions drop-down and popup menus.

Notes Windows system menus are drawn by Windows and are not skinned.

If the **JadeSkinForm** class **myMenuSkin** property is null (a reference to the **JadeSkinMenu** class), drop-down and popup menus are drawn using the skin menu color and font properties of the form, with a border style of **BorderStyle_3DRaised** (3).

You can skin drop-down menus and popup (context) menus, as follows.

- Type of border in terms of the **borderStyle** property or by using eight images
- Inner image, **backBrush**, or **backColor**
- **backColor** of selected items
- Menu text, selected, and disabled foreground colors
- Font
- Check box, separator, and right arrow images

The **JadeSkinArea** class **backColor** property defines the background color to be used to draw the background of non-selected and enabled menu items in the drop-down or popup menu. The default value of **Default_Color** for this property indicates that the default background color is defined by the form skin **menuBackColor** property. If that property is also set to **Default_Color**, the Windows default background color is used.

For details about the properties defined in the **JadeSkinMenu** class, see "JadeSkinMenu Properties", in the following subsection. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinArea

Inherited By: (None)

JadeSkinMenu Properties

The **JadeSkinMenu** class provides the properties summarized in the following table.

Property	Description
backColorSelected	Contains the color used to draw the background of selected and enabled menu items in the drop-down or popup menu
borderStyle	Contains the type of border to be drawn
drawMenuSelectionFlat	Specifies how menu items are drawn in the popup menu when a menu item is selected (which is **false** by default)
fontBold	Specifies whether the menu font is bold (which is **false** by default)
fontItalic	Specifies whether the menu font is bold (which is **false** by default)
fontName	Contains the name of the menu font ("Tahoma" is the default value)
fontSize	Contains the size of the menu font (8.25 points is the default value)

Property	Description
foreColor	Contains the color used to draw the text for non-selected and enabled menu items in the drop-down or popup menu
foreColorDisabled	Contains the color used to draw the text for disabled menu items in the drop-down or popup menu
foreColorSelected	Contains the color used to draw the text for selected menu items in the drop-down or popup menu
imgCheckMark	Contains the image used to draw for checked menu items
imgRightArrow	Contains the image used to draw the right arrow submenu indicator
imgSeparator	Contains the image used to draw menu separators
lineHeight	Contains the height in pixels of each menu line item
pixelsAfterCheckMark	Contains the amount of space left after the check mark column
pixelsAfterPicture	Contains the amount of space left after the picture image column
pixelsBeforeAccelerator	Contains the amount of space left before the accelerator text column
pixelsBeforeCheckMark	Contains the amount of space left before the check mark column
pixelsBeforeRightArrow	Contains the amount space left before the right arrow column

backColorSelected

Type: Integer

Availability: Read or write at any time

The **backColorSelected** property of the **JadeSkinMenu** class contains the color used to draw the background of selected and enabled menu items in the drop-down or popup menu.

The default value of **Default_Color** indicates that the defined default background color of the **JadeSkinForm** class **menuBackColorSelected** property is used. If that property is also set to **Default_Color**, the Windows default value is used.

borderStyle

Type: Integer

Availability: Read or write at any time

The **borderStyle** property of the **JadeSkinMenu** class contains the type of border to be drawn for a drop-down or popup menu. The default value is **BorderStyle_3DRaised** (3).

The **borderStyle** property values are listed in the following table.

JadeSkinArea Class Constant	Integer Value	Description
BorderStyle_3DRaised	3	Raised three-dimensional border (two pixels).
BorderStyle_3DSunken	2	Sunken three-dimensional border (two pixels).

JadeSkinArea Class Constant	Integer Value	Description
BorderStyle_Images	4	Border is drawn using the supplied images of the JadeSkinArea class. If there are no images, the control does not have a border.
BorderStyle_None	0	No border is drawn.
BorderStyle_Single	1	Fixed single-line border.

If this property is not set to **BorderStyle_Images** (4), the defined border is drawn and the border images are ignored (see the **JadeSkinArea** class properties).

drawMenuSelectionFlat

Type: Boolean

Availability: Read or write at any time

The **drawMenuSelectionFlat** property of the **JadeSkinMenu** class specifies how menu items are drawn in the popup menu when a menu item is selected. If the value is:

- **False** (the default), the selected menu line item is drawn using the effective menu item selection background and foreground colors with a border that is the same color as the text of the menu item.
- **True**, the selected menu line item is always drawn in a flat manner using the defined menu line selection colors (with no surrounding border).

Control the **drawMenuSelectionFlat** property with the **Draw Flat Selection** check box on the **Menus** sheet of the Jade Skin Maintenance dialog.

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **JadeSkinMenu** class specifies whether the bold font attribute is applied to text on drop-down or popup menus.

The settings for the **fontBold** property are listed in the following table.

Value	Description
true	Turns on the bold formatting
false	Turns off the bold formatting (the default)

fontItalic

Type: Boolean

Availability: Read or write at any time

The **fontItalic** property of the **JadeSkinMenu** class specifies whether the italic font attribute is applied to text on drop-down or popup menus.

The settings for the **fontItalic** property are listed in the following table.

Value	Description
true	Turns on the italic formatting
false	Turns off the italic formatting (the default)

fontName

Type: String[31]

Availability: Read or write at any time

The **fontName** property of the **JadeSkinMenu** class contains the name of the font used to display menu text on drop-down or popup menus.

The default value is null (**""**), meaning that the font defined by the **JadeSkinForm** class **menuFontName**, **menuFontSize**, **menuFontBold**, and **menuFontItalic** properties is used.

Note The fonts that are available in JADE vary, according to your system configuration, display devices, and printing devices.

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property of the **JadeSkinMenu** class contains the size of the font used for menu text displayed on drop-down or popup menus. The default value is **0**.

foreColor

Type: Integer

Availability: Read or write at any time

The **foreColor** property of the **JadeSkinMenu** class contains the color used to draw the text for non-selected and enabled menu items of drop-down or popup menus.

The default value of **Default_Color** indicates that the defined default background color of the **JadeSkinForm** class **menuForeColor** property is used. If that property is also set to **Default_Color**, the Windows default value is used.

foreColorDisabled

Type: Integer

Availability: Read or write at any time

The **foreColorDisabled** property of the **JadeSkinMenu** class contains the color used to draw the text for disabled menu items on drop-down and popup menus.

The default value of **Default_Color** indicates that the defined default background color of the **JadeSkinForm** class **menuForeColorDisabled** property is used. If that property is also set to **Default_Color**, the Windows default value is used.

foreColorSelected

Type: Integer

Availability: Read or write at any time

The **foreColorSelected** property of the **JadeSkinMenu** class contains the color used to draw the text for selected items on drop-down and popup menus.

The default value of **Default_Color** indicates that the defined default background color of the **JadeSkinForm** class **menuForeColorSelected** property is used. If that property is also set to **Default_Color**, the Windows default value is used.

imgCheckMark

Type: Binary

Availability: Read or write at any time

The **imgCheckMark** property of the **JadeSkinMenu** class contains the image used to draw checked menu items. The default value of **null** indicates that the default check mark image is drawn.

Note If the image is a monochrome bitmap, the image is drawn using the background and foreground colors of the menu item.

imgRightArrow

Type: Binary

Availability: Read or write at any time

The **imgRightArrow** property of the **JadeSkinMenu** class contains the image used to draw the right arrow submenu indicator. The default value of **null** indicates that the default right arrow image is drawn.

Note If the image is a monochrome bitmap, the image is drawn using the background and foreground colors of the menu item.

imgSeparator

Type: Binary

Availability: Read or write at any time

The **imgSeparator** property of the **JadeSkinMenu** class contains the image used to draw menu separators. The default value of **null** indicates that the default menu separator image is drawn. This image is stretched horizontally to fit.

Note If the image is a monochrome bitmap, the image is drawn using the background and foreground colors of the menu item.

lineHeight

Type: Integer

Availability: Read or write at any time

The **lineHeight** property of the **JadeSkinMenu** class contains the height of each menu line item.

The default value of zero (**0**) indicates that the default menu spacing is used.

If the value of this property is not zero (**0**) and it is less than the height of the font text + 2, the height that is used is the height of font text height + 2.

pixelsAfterCheckMark

Type: Integer

Availability: Read or write at any time

The **pixelsAfterCheckMark** property of the **JadeSkinMenu** class contains the amount of space left after the check mark column.

The default value is zero (**0**), because the default check mark image includes that spacing in the image. The check mark column is always displayed.

pixelsAfterPicture

Type: Integer

Availability: Read or write at any time

The **pixelsAfterPicture** property of the **JadeSkinMenu** class contains the amount of space left after the picture image column. The default value is **5** pixels.

Note If none of the displayed menu items has a picture image, the picture column and the value of the **pixelsAfterPicture** property are ignored.

pixelsBeforeAccelerator

Type: Integer

Availability: Read or write at any time

The **pixelsBeforeAccelerator** property of the **JadeSkinMenu** class contains the amount of space left before the accelerator text column. The default value is **5** pixels.

Note If none of the displayed menu items has an accelerator, the accelerator column and the value of the **pixelsBeforeAccelerator** property are ignored.

pixelsBeforeCheckMark

Type: Integer

Availability: Read or write at any time

The **pixelsBeforeCheckMark** property of the **JadeSkinMenu** class contains the amount of space left before the check mark column. The default value is zero (**0**), because the default check mark image includes that spacing in the image.

The check mark column is always displayed.

pixelsBeforeRightArrow

Type: Integer

Availability: Read or write at any time

The **pixelsBeforeRightArrow** property of the **JadeSkinMenu** class contains the amount of space left before the right arrow column. The default value is **5** pixels.

Note If none of the displayed menu items has a submenu, the right arrow column and the value of the **pixelsBeforeRightArrow** property are ignored.

JadeSkinRoot Class

The **JadeSkinRoot** class is the root class for all of the skin entities. This class contains a series of dictionaries that enable you to reference the skin entities. Obtain the **JadeSkinRoot** class instance as follows.

```
root := JadeSkinRoot.firstInstance;
```

The **JadeSkinRoot** class properties are automatically maintained member key dictionaries that use the **name** property of the **JadeSkinEntity** class as the key. For details about the properties defined in the **JadeSkinRoot** class, see "JadeSkinRoot Properties", in the following subsection. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinEntity

Inherited By: (None)

JadeSkinRoot Properties

The **JadeSkinRoot** class provides the properties summarized in the following table.

Property	Reference to the dictionary containing the name of all...
allApplicationSkins	Application skins
allControlSkins	Control skins
allFormSkins	Form skins
allMenuSkins	Menu skins
allSimpleButtonSkins	Simple button skins
allSkinCategories	Skin categories
allSkinEntities	Skin entities
allWindowStateImages	Window state images

allApplicationSkins

Type: JadeSkinApplicationNameDict

Availability: Read or write at any time

The **allApplicationSkins** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all application skins.

The key of the member key dictionary containing all application skins is the **JadeSkinEntity** class **name** property.

allControlSkins

Type: JadeSkinControlNameDict

Availability: Read or write at any time

The **allControlSkins** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all control skins.

The key of the member key dictionary containing all control skins is the **JadeSkinEntity** class **name** property.

allFormSkins

Type: JadeSkinFormNameDict

Availability: Read or write at any time

The **allFormSkins** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all form skins.

The key of the member key dictionary containing all form skins is the **JadeSkinEntity** class **name** property.

allMenuSkins

Type: JadeSkinMenuNameDict

Availability: Read or write at any time

The **allMenuSkins** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all menu skins.

The key of the member key dictionary containing all menu skins is the **JadeSkinEntity** class **name** property.

allSimpleButtonSkins

Type: JadeSkinSimpleButtonNameDict

Availability: Read or write at any time

The **allSimpleButtonSkins** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all simple button skins.

The key of the member key dictionary containing all simple button skins is the **JadeSkinEntity** class **name** property.

allSkinCategories

Type: JadeSkinCategoryNameDict

Availability: Read or write at any time

The **allSkinCategories** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all skin categories.

The key of the member key dictionary containing all skin categories is the **JadeSkinEntity** class **name** property.

allSkinEntities

Type: JadeSkinEntityNameDict

Availability: Read or write at any time

The **allSkinEntities** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all skin entities.

The key of the member key dictionary containing all skin entities is the **JadeSkinEntity** class **name** property.

allWindowStateImages

Type: JadeSkinWindowStateNameDict

Availability: Read or write at any time

The **allWindowStateImages** property of the **JadeSkinRoot** class contains an automatic reference to the collection of all window state images.

The key of the member key dictionary containing all window state images is the **JadeSkinEntity** class **name** property.

JadeSkinSimpleButton Class

The **JadeSkinSimpleButton** class holds the skin definitions for a simple button and its various states, which are:

- Up
- Down
- Disabled
- Rollover

Each state is defined by using a single image.

Simple buttons are used to define the form buttons such as the **Maximize** and **Minimize** buttons and the buttons for **CheckBox**, **ComboBox**, **OptionButton**, and **ScrollBar** controls.

For details about the properties defined in the **JadeSkinSimpleButton** class, see "JadeSkinSimpleButton Properties", in the following subsection. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinEntity

Inherited By: (None)

JadeSkinSimpleButton Properties

The **JadeSkinSimpleButton** class provides the properties summarized in the following table.

Property	Image displayed...
imgDisabled	For the disabled state.
imgDown	For the down state when the button is clicked. If you do not supply the image, the **imgUp** image is used.
imgRollOver	When the mouse is over the button in the up state. If you do not supply the image, the **imgUp** image is used.
imgUp	For the up or normal state. If you do not supply the image, the default button is drawn.

imgDisabled

Type: Binary

Availability: Read or write at any time

The **imgDisabled** property of the **JadeSkinSimpleButton** class contains the image that is displayed for the disabled state of simple buttons.

If you do not supply the disabled image, the image contained in the **imgUp** property is used.

imgDown

Type: Binary

Availability: Read or write at any time

The **imgDown** property of the **JadeSkinSimpleButton** class contains the image that is displayed for the down state of simple buttons.

If you do not supply the down image, the image contained in the **imgUp** property is used.

imgRollOver

Type: Binary

Availability: Read or write at any time

The **imgRollOver** property of the **JadeSkinSimpleButton** class contains the image that is displayed for the rollover state of simple buttons.

If you do not supply the rollover image, the image contained in the **imgUp** property is used.

imgUp

Type: Binary

Availability: Read or write at any time

The **imgUp** property of the **JadeSkinSimpleButton** class contains the image that is displayed for the up state of simple buttons.

If you do not supply the up image, the default button is drawn.

JadeSkinWindow Class

The **JadeSkinWindow** class, which is the abstract superclass of all **Window** class skins, contains the defined image and category of the skins. For details about the properties defined in the **JadeSkinWindow** class, see "JadeSkinWindow Properties", in the following subsection. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinArea

Inherited By: JadeSkinControl, JadeSkinForm

JadeSkinWindow Properties

The **JadeSkinWindow** class provides the properties summarized in the following table.

Property	Reference to the …
myImageMask	**JadeSkinWindowStateImage** object used to construct a Windows region for the window
mySkinCategory	Skin category that applies to the skin

myImageMask

Type: JadeSkinWindowStateImage

Availability: Read or write at any time

The **myImageMask** property of the **JadeSkinWindow** class contains an optional reference to the **JadeSkinWindowStateImage** object used to construct a Windows region for the window. (See also the **JadeSkinButton** class **createRegionFromMask** property.)

By default, all windows are rectangular. Applying a region to a window enables it to be of any shape and include 'holes' inside it.

Note Any part of any child outside the defined region is not displayed.

To understand the way in which a region is constructed, consider that the image specified by the **myImageMask** property is drawn (stretched) over the top of the window, including any border area.

Only black pixels are considered part of the window when it is painted or clicked on. The result could be a window with rounded corners, a window with holes in it, and so on.

Because the mask image is built to the same size as the actual window, if border masks are defined, the only reasonable region that could be constructed is one where the corners as shaped. As all other areas are stretched to fit, it is likely that they would not provide a suitable result. You would achieve a better result by defining only an inner image for the mask that is stretched.

mySkinCategory

Type: JadeSkinCategory

Availability: Read or write at any time

The **mySkinCategory** property of the **JadeSkinWindow** class contains an optional reference to the skin category that applies to the skin.

If the **mySkinCategory** property is set to **"Company Logo"**, for example, the skin is applied only to a window of the appropriate type that has the **skinCategoryName** property also set to **"Company Logo"**.

In addition, you can define a skin category for a **Control** subclass (for example, each **BaseControl** subclass) and associate a different category name with each of those skins. The constructor of each control subclass can then set the appropriate category name on the control so that the correct **JadeSkinBaseControl** skin is then applied.

JadeSkinWindowStateImage Class

The **JadeSkinWindowStateImage** class defines the image of a window area for a specific state (that is, for the up, down, rollover, or disabled state).

For details about the properties defined in the **JadeSkinWindowStateImage** class, see "JadeSkinWindowStateImage Properties", in the following subsection. For details about defining and maintaining skins, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: JadeSkinArea

Inherited By: (None)

JadeSkinWindowStateImage Properties

The **JadeSkinWindowStateImage** class provides the properties summarized in the following table.

Property	Description
foreColor	Specifies the color of the text for the defined window state
isImageMask	Specifies whether this image is to be used for a window region mask.

foreColor

Type: Integer

Availability: Read or write at any time

The **foreColor** property of the **JadeSkinWindowStateImage** class contains the foreground color used to draw text for the defined window state. This property is used for the text for a defined window state of a **Button**, **Folder** control sheet tab, and **JadeMask** control when a **JadeSkinButton** skin is assigned.

The default value of **#80000000** means that no foreground color is applied to the text color of the control by the window state so that the text color is defined by the value of the **foreColor** property of the **JadeSkinButton** class or **Control** class; otherwise the value is used to draw the text when a specific window state is active. For example, when the mouse is moved over a button and an **imgRollOver** property value is defined for that state, the text is drawn using the **foreColor** property value when it is not **#80000000**.

This behavior is not affected by any **foreColor** property value assigned to the control. It would therefore normally not be used to draw the text in its normal state, because otherwise a skinned control will not show a **foreColor** value assigned by logic to the control.

Use the **Default foreColor** check box on the **Window State Image** sheet of the Jade Skin Maintenance form to specify the rollover foreground color. When you uncheck this check box, the common Color dialog is displayed, to enable you to select or define the foreground color you require for the **rollOver** state of the skin.

isImageMask

Type: Boolean

Availability: Read or write at any time

The **isImageMask** property of the **JadeSkinWindowStateImage** class specifies whether the image is to be used for a window region mask. This causes the images to be built in a special way for region handling.

It is also used to filter out images not suitable for region masks (that is, icons, cursors, and meta files).

JadeSOAPException Class

The **JadeSOAPException** class is the transient class that defines behavior for exceptions that occur as a result of Web service handling.

For details about Web services exceptions, see the error messages in the range 11000 through 11052 in "Error Messages and System Messages", in the **JADEMsgs.pdf** file.

Inherits From: NormalException

Inherited By: (None)

JadeSSLContext Class

The **JadeSSLContext** class implements the behavior required for secure connections using a Secure Sockets Layer (SSL) library protocol instead of the Transmission Control Protocol / Internet Protocol (TCP/IP) protocol when the **TcpIpConnection** class **sslContext** property contains a reference to a **JadeSSLContext** transient object.

SSL is a secure communication protocol on top of an already established TCP/IP connection. SSL libraries are generated from publicly available third-party sources, maintained by the OpenSSL Group (http://www.openssl.org). JADE supports TLS (Transport Layer Security) version 1, TLS version 1.1, and TLS version 1.2.

JadeSSLContext connections use digital certificates in X509 format, which must exist on disk in Privacy-Enhanced Electronic Mail (PEM)-encoded certificate (PEM) format.

The method in the following example opens an outgoing SSL connection.

```
vars
    tcpip      : TcpIpConnection;
    sslContext : JadeSSLContext;
    x509       : JadeX509Certificate;
begin
    create x509 transient;
    x509.readCertificateDataFromFile("c:\Certificates\client.pem");
    x509.readPrivateKeyDataFromFile("c:\Certificates\client.key", "myPassword");
    create sslContext transient;
    sslContext.methodType := JadeSSLContext.MethodTLSv1_2;
    sslContext.caFile := "c:\Certificates\serverCAcerts.pem";
    sslContext.x509 := x509;
    create tcpip transient;
    tcpip.name := "mySSLNode";
    tcpip.port := 8097;
    tcpip.sslContext := sslContext;
    tcpip.open;
    // ... send and receive some data
    tcpip.close;
epilog
    delete x509;
    delete sslContext;
    delete tcpip;
end;
```

The method in the following example listens for an incoming SSL connection request.

```
vars
    tcpip : TcpIpConnection;
    sslContext : JadeSSLContext;
    x509: JadeX509Certificate;
begin
    create x509 transient;
    x509.readCertificateDataFromFile("c:\Certificates\server.pem");
    x509.readPrivateKeyDataFromFile("c:\Certificates\server.key", "mySrvPassword");
    create sslContext transient;
    sslContext.methodType := JadeSSLContext.MethodTLSv1_2;
    sslContext.caFile := "c:\Certificates\clientCAcerts.pem";
    sslContext.x509 := x509;
```

```
        create tcpip transient;
        tcpip.port := 8097;
        tcpip.sslContext := sslContext;
        tcpip.listen;
        // ... send and receive some data
        tcpip.close;
    epilog
        delete x509;
        delete sslContext;
        delete tcpip;
    end;
```

For details about the constants, properties and methods defined in the **JadeSSLContext** class, see "JadeSSLContext Class Constants", "JadeSSLContext Properties" and "JadeSSLContext Methods", in the following subsections. For details about returning the type of encryption being used by a JADE thin client TCP/IP connection in the current application, see the **Application** class **getThinClientEncryptionType** method.

Inherits From: Object

Inherited By: (None)

JadeSSLContext Class Constants

The constants provided by the **JadeSSLContext** class are listed in the following table.

JadeSSLContext Class Constant	Integer Value
MethodSSLNone	0
MethodSSLv2 (superseded)	1
MethodSSLv23 (superseded)	2
MethodSSLv3 (superseded)	3
MethodTLSv1	4
MethodTLSv1_1	5
MethodTLSv1_2	6

JadeSSLContext Properties

The properties defined in the **JadeSSLContext** class are summarized in the following table.

Property	Description
caFile	Contains the name of the file containing the digital certificate or certificates of Certificate Authorities
caPath	Contains the absolute path of the directory containing digital Certificate Authority files
cipherList	Contains a colon-separated list of ciphers that can be used for the connection object
methodType	Contains the indicator to the SSL library of which secure protocol to use
verifyDepth	Contains the maximum verification depth in a chain of certificates

Property	Description
verifyRemoteCertificate	Specifies whether the peer certificate is verified when the SSL connection is established
x509	Contains a reference to the **JadeX509Certificate** object

caFile

Type: String

The **caFile** property of the **JadeSSLContext** class contains the name of the Certificate Authority file containing the Privacy-Enhanced Electronic Mail (PEM)-encoded digital certificate or certificates used by the SSL libraries to validate the issuers of the peer certificate.

Local operating system file naming conventions apply.

This file is located in the directory specified in the **caPath** property.

caPath

Type: String

The **caPath** property of the **JadeSSLContext** class contains the absolute path of the directory or directories containing the master public certificates for Certificate Authority files (for example, **Verisign**) used by the SSL libraries to validate the issuers of the peer certificate or certificates. Each file in the specified directory can contain only a single certificate.

Local operating system file naming conventions apply. If you specify more than one directory, separate each path name with a semicolon character (**;**). The path name can contain a drive letter.

Note Support of OpenSSL requires file names need to be hashed and the OpenSSL-supplied Practical Extraction and Report Language (Perl) script for hashing does not work on Windows operating systems because Perl does not support symbolic linking.

cipherList

Type: String

The **cipherList** property of the **JadeSSLContext** class contains a colon-separated list of ciphers that can be used for the connection object. Separate each cipher with a colon character (**:**).

As the default value is null (**""**), you must specify this property before establishing a connection if you want to use a list. When SSL detects a null value, it finds the strongest matching cipher. The connection fails if both ends (that is, the server and client nodes) do not have at least one cipher in common.

SSL selects the strongest common cipher available to both ends of the connection. For details, see "Secure Sockets Layer (SSL) Security", in Chapter 2 of the *JADE Object Manager Guide*.

methodType

Type: Integer

The **methodType** property of the **JadeSSLContext** class contains the indicator to the SSL library of which Transport Layer Security (TLS) version that the application server and presentation clients use to communicate.

The **methodType** property can contain one of the **JadeSSLContext** class constants listed in the following table.

JadeSSLContext Class Constant	Integer Value
MethodSSLNone (the default value)	0
MethodSSLv2 (superseded)	1
MethodSSLv23 (superseded)	2
MethodSSLv3 (superseded)	3
MethodTLSv1	4
MethodTLSv1_1	5
MethodTLSv1_2	6

verifyDepth

Type: Integer

The **verifyDepth** property of the **JadeSSLContext** class contains the maximum verification depth in a chain of certificates. This value specifies how far back in the certificate chain checking for the Certificate Authority signature goes. A certificate is signed by a Certificate Authority (CA) certificate. The CA certificate is signed by a more-trusted CA or it is signed by itself.

If you know the maximum depth of certificate chain, set the property to that value, or depth. A large value allows more checking but too small a value may not verify the complete certificate chain. Specify zero (**0**) if you want unlimited depth checking. The default value is **9**.

verifyRemoteCertificate

Type: Boolean

The **verifyRemoteCertificate** property of the **JadeSSLContext** class specifies whether the peer certificate is verified when the SSL connection is established. (The remote program must have a file defined in the **caFile** property for that node.) The default value of **false** indicates that the remote peer certificate is not verified.

x509

Type: JadeX509Certificate

The **x509** property of the **JadeSSLContext** class contains a reference to the **JadeX509Certificate** object of the digital certificate to be used for identification when establishing an SSL connection. The default value is null; that is, there is no X509 certificate reference.

JadeSSLContext Methods

The methods defined in the **JadeSSLContext** class are summarized in the following table.

Method	Returns …
getActiveCipher	The active cipher for the specified TCP/IP connection
getPeerCertificate	Reference to the certificate object used by the connection peer

getActiveCipher

Signature `getActiveCipher(tcp: TcpIpConnection): String;`

The **getActiveCipher** method of the **JadeSSLContext** class returns a string containing the active cipher for the TCP/IP connection specified in the **tcp** parameter.

If the specified connection is not currently connected, this method returns null (**""**).

getPeerCertificate

Signature `getPeerCertificate(tcp: TcpIpConnection): JadeX509Certificate;`

The **getPeerCertificate** method of the **JadeSSLContext** class returns a read-only reference to a **JadeX509Certificate** object that contains information about the certificate used by the connection peer if an SSL connection has been successfully established or it returns null (**""**) if an SSL connection has not been established.

JadeTableCell Class

The **JadeTableCell** class provides access to an internally created cell object that you can use to directly reference the properties and methods of the table cell. This object is created on the first call to the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method for each **Table** control.

Using instances of the **JadeTableCell** class is equivalent to setting the **accessMode** property of the **Table** control to the **Table.AccessMode_Cell** value without having to set the **row**, **column**, or **sheet** property.

One **JadeTableCell** object only is created for each **Table** control, as there would be too much overhead required in creating an object for each cell. This object is a proxy object holding the reference to the cell that was last accessed by using the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method.

Accessing a cell by using the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method also sets the **accessedCell** property in the **Table** class, allowing subsequent access to that table cell.

The following code fragments show examples of accessing the last table elements that were accessed.

```
table1.accessCell(2, 3).inputType := Table.InputType_TextBox;
table1.accessedCell.foreColor := Red;

table1.accessSheet(2).accessCell(1, 4).text := "Company";
table1.accessedCell.alignment := Table.Alignment_Right_Middle;
```

Storing a reference to a returned cell causes problems unless you take a copy of that cell, as shown in the following example in which both **cell1** and **cell2** refer to the same object, which is referencing **cell(3, 4)**.

```
cell1 := table1.accessCell(2, 3);
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

In the following example, **cell1** has been cloned and still refers to **cell(2, 3)**.

```
cell1 := table1.accessCell(2, 3).cloneSelf(true);
// the cloned cell must be deleted by your logic
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

For details about the properties and methods defined in the **JadeTableCell** class, see "JadeTableCell Properties" and "JadeTableCell Methods", in the following subsections.

For details about the superclass that encapsulates the behavior required to directly access the properties and methods of a table cell, see "JadeTableElement Class", later in this chapter, and for details about the table control and the constants, properties, methods, and events that it provides, see "Table Class", in Chapter 2.

Inherits From: JadeTableElement

Inherited By: (None)

JadeTableCell Properties

The properties defined in the **JadeTableCell** class are summarized in the following table.

Property	Description
column	Contains the column number of the accessed cell

Property	Description
comboIndex	Contains the index of a combo box in the cell
hyperLink	Specifies whether the cell contains a hyperlink string that is programmatically attached to the cell
mergeCells	Contains a value representing the type of cell merging that can be performed, if any
picture	Contains a graphic to be displayed in the cell
row	Contains the row number of the accessed cell
sheet	Contains the sheet number of the accessed cell
text	Contains the text in the cell

column

Type: Integer

Availability: Read or write at any time

The **column**, **row**, and **sheet** properties of the **JadeTableCell** class define the cell referenced by this object. These properties are set when the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method is called.

You can also set this property manually, allowing your logic to dynamically modify the cell that is being referenced.

The following example shows the use of the **column** property.

```
tableCell := table1.accessCell(2, 10);
counter   := 10;
while counter >= 1 do
    tableCell.column   := counter;
    tableCell.selected := true;
    counter            := counter - 1;
endwhile;
```

comboIndex

Type: Integer

Availability: Read or write at run time only (for a cell with **inputType** set to **3** only)

The **comboIndex** property of the **JadeTableCell** class contains the index of a combo box in a cell of a **Table** cell referenced by this object.

This property applies only to a cell that has the **inputType** property of the **Table** class set to **InputType_ ComboBox** (3).

For a description of this property, see the **Table** class **comboIndex** property.

The code fragment in the following example shows the use of the **comboIndex** property.

```
table1.accessCell(2, 3).comboIndex := listBox.listIndex;
```

hyperLink

Type: Boolean

Availability: Read or write at run time only

The **hyperLink** property of the **JadeTableCell** class specifies whether the cell referenced by this object contains a hyperlink string that is programmatically attached to the cell; that is, it holds the value for the **Table** class **isHyperlinkSet**, **resetHyperlinkCell**, or **setHyperlinkCell** method.

Note The **hyperLink** property is ignored for JADE applications that are Web-enabled. For these applications, you should use the **hyperlinkColumn** array property of the **Table** class.

The code fragment in the following example shows the use of the **hyperlink** property.

```
table1.accessCell(2, 4).hyperLink := checkbox.value;
```

mergeCells

Type: Integer

Availability: Read or write at run time only

The **mergeCells** property of the **JadeTableCell** class contains a value that specifies the type of cell merging that can be performed, if any. You can set the **mergeCells** property to one of the **Table** class constants listed in the following table.

Table Class Constant	Value	Description
MergeCells_Available	0	Cell available for merging if empty (the default value)
MergeCells_Merge	1	Merge all following empty cells
MergeCells_MergeSelectable	2	Merge all following empty cells (cells still selectable)
MergeCells_NotAvailable	3	Current cell not available for merging

The **mergeCells** property enables you to extend a cell over adjacent cells, as shown in the following image.

In this example, the table holds entries divided into months, with the month heading centered across three cells and an extended fixed-column cell further dividing each month.

Setting the **mergeCells** property of a cell to **MergeCells_MergeSelectable** (2) or **MergeCells_Merge** (1) causes that cell to be drawn across adjacent cells of the same row. The drawing stops prior to a cell in which one of the following applies.

- The **mergeCells** property of the cell is not set to **MergeCells_Available** (0).
- The cell has a non-null text value.

- The cell has a non-null picture value.
- The cell is the current cell.

The **mergeCells** property can be set to a value represented by a **Table** control class constant, as follows.

- **MergeCells_Available** (0)

  If the cell has no text or picture, it can be merged into a preceding cell. (This is the default value for all cells.)

- **MergeCells_Merge** (1)

  The cells following the cell will be merged into this cell up to the end of the row or up to (but not including):

  - A cell that contains a text or picture
  - The **mergeCells** property of the cell is not **MergeCells_Available** (0)
  - The cell is the current cell

  Clicking anywhere in the expanded cell is treated as a **click** event in that cell and the hidden cell or cells cannot be accessed. Similarly, the left and right arrow keys ignore the covered cells.

  The **getCellFromPosition** method also returns the merged cell for any position within the whole of the drawn cell.

- **MergeCells_MergeSelectable** (2)

  Cells are merged in exactly the same way as they are when the property is set to the **MergeCells_Merge** value, but the covered cells can be 'brought back to life' by clicking on the expanded cell in the position corresponding to the hidden cell. Similarly, the left and right arrow keys step into the previously hidden cells rather than skipping to the start or end of the merged cell.

  The **getCellFromPosition** method also returns the cell that corresponds to the position, ignoring any merged cell size.

- **MergeCells_NotAvailable** (3)

  This setting has no effect on that cell except to specifically terminate any merging process; that is, this cell cannot be merged.

  This value is required only to terminate the merging process prior to an empty cell.

If the cells are automatically resized (by using the **Table** class **autoSize** property) and the cell contents do not fit within the whole of the merged columns, the first column of the merged columns is enlarged. (This size is calculated by determining the size of the merged columns using the above rules, except that the current cell does not terminate the merging of the cells.)

The following example shows the use of the **mergeCells** property.

```
table1.accessCell(2, 2).mergeCells := Table.MergeCells_Merge;
    // column 2 merges following cells
table1.accessedCell.text           := "Heading 1";
table1.accessedCell.alignment      := Table.Alignment_Center_Middle;
    // center
table1.accessCell(2, 7).mergeCells := Table.MergeCells_NotAvailable;
    // ensure merging ended
```

Notes Merged cells do not affect the values of the **Table** control class **columnWidth** property.

A merged cell uses the total displayed width for data entry when used with a cell that has the **inputType** property set or that has a **JadeTableElement** class **cellControl** property.

If the horizontal alignment of the cell is not left aligned, the alignment is performed based on the total width of the merged cell.

You can also merge cells belonging to a fixed row and column to allow, for example, headings that span more than one column. If you merge fixed cells, the moving and resizing processes of columns and rows are also affected by whether the **MergeCells_MergeSelectable** (2) or **MergeCells_Merge** (1) value applies. For the **MergeCells_Merge** value, the hidden cells cannot be moved or resized, while they can be for the **MergeCells_MergeSelectable** value.

See also the **JadeTableCell** class **getCellWidth** method.

picture

Type: Binary

Availability: Read or write at run time only

The **picture** property of the **JadeTableCell** class contains a graphic to be displayed in the cell referenced by this object. For a description of this property, see the **Table** class **picture** property.

The code fragment in the following example shows the use of the **picture** property.

```
table1.accessCell(2, 3).picture := app.loadPicture("c:\images\company.ico");
```

row

Type: Integer

Availability: Read or write at any time

The **row**, **column**, and **sheet** properties of the **JadeTableCell** class define the cell referenced by this object. These properties are set when the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method is called.

You can also set this property manually, allowing your logic to dynamically modify the cell that is being referenced, as shown in the following example.

```
tableCell := table1.accessCell(10, 2);
counter   := 10;
while counter >= 1 do
    tableCell.row      := counter;
    tableCell.selected := true;
    counter            := counter - 1;
endwhile;
```

sheet

Type: Integer

Availability: Read or write at any time

The **sheet**, **column**, and **row** properties of the **JadeTableCell** class define the cell referenced by this object. These properties are set when the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method is called.

You can also set this property manually, allowing your logic to dynamically modify the cell that is being referenced.

text

Type: String

Availability: Read or write at run time only

The **text** property of the **JadeTableCell** class contains the text of the cell referenced by this object. For a description of this property, see the **Table** class **text** property.

The code fragment in the following example shows the use of the **text** property.

```
table1.accessCell(1, 2).text := "Company";
```

The code fragment in the following example uses concatenation with the **Tab** character to store text in cells to the right of the specified cell.

```
// Set up column headings
table1.accessCell(1, 1).text := "Name" & Tab & "Address" & Tab & "Phone";
```

JadeTableCell Methods

The methods defined in the **JadeTableCell** class are summarized in the following table.

Property	Description
delete	Deletes the **JadeTableCell** object but has no impact on the contents of the table
getCellWidth	Returns the logical width of the cell in pixels
positionLeft	Returns the displayed left position of the cell in pixels
positionTop	Returns the displayed top position of the cell in pixels
setPictureDescription	Assigns a description to a picture in a cell

delete

Signature `delete() updating;`

The **delete** method deletes the **JadeTableCell** object but has no impact on the contents of the **Table** control.

A new **JadeTableCell** object is created on the next call of the **Table** class **accessCell** method or the **JadeTableSheet** class **accessCell** method.

getCellWidth

Signature `getCellWidth(): Integer;`

The **getCellWidth** method of the **JadeTableCell** class returns the logical width of the cell in pixels.

For a merged cell where the **mergeCells** property is set to the **MergeCells_MergeSelectable** (2) or **MergeCells_Merge** (1) value, the width is the size after the merging process.

For a cell that is hidden by the merge process, this method returns zero (**0**).

For a non-merged cell, this method returns the same value as the **Table** control class **columnWidth** property of the cell.

The following example, which ensures that a merged cell is always totally displayed, shows the use of the **getCellWidth** method.

```
// If the first cell is a hidden cell of the merge, move the table
// to the start of a merged cell
while table1.leftColumn > 2 and table1.accessCell(table1.row,
                  table1.leftColumn).getCellWidth = 0 do
    table1.leftColumn := table1.leftColumn - 1;
endwhile;
```

See also the **JadeTableCell** class **mergeCells** property.

positionLeft

Signature `positionLeft(): Integer;`

The **positionLeft** property of the **JadeTableCell** class returns the displayed position in pixels of the cell referenced by this object, relative to the client area of the **Table** control (the area inside borders).

The **sheet**, **row**, and **column** properties define the current cell.

If the current cell is not visible, one or both of the **positionLeft** or **positionTop** methods returns **-1**.

The code fragment in the following example shows the use of the **positionLeft** method.

```
if table1.accessCell(2, 3).positionLeft >= 0 then
    myTextBox.left := table1.accessedCell.positionLeft;
endif;
```

positionTop

Signature `positionTop(): Integer;`

The **positionTop** property of the **JadeTableCell** class returns the displayed position in pixels of the cell referenced by this object, relative to the client area of the **Table** control (the area inside borders).

If the current cell is not visible, one or both of the **positionLeft** or **positionTop** methods returns **-1**.

The code fragment in the following example shows the use of the **positionTop** method.

```
if table1.accessCell(2, 3).positionTop >= 0 then
    myTextBox.top := table1.accessedCell.positionTop;
endif;
```

setPictureDescription

Signature `setPictureDescription(desc: String) updating;`

The **setPictureDescription** property of the **JadeTableCell** class assigns a description to a picture in a cell. This description is used when accessibility screen-reading software (for example, Scientific Freedom Jaws) reads the contents of a cell that has no cell text.

The assignment associates the description with the picture, not with the cell, so if the same picture is assigned to more than one cell, the last description set for the image applies to all cells.

If the method is called for a cell that does not have an assigned image, an exception is raised. Ensure that the picture is assigned to the cell before the description is attached, as in the following example.

```
vars
    jtc : JadeTableCell;
begin
    jtc := table1.accessCell(2,1);
    jtc.picture := app.loadPicture("C:\bridge.jpg");
    jtc.setPictureDescription("London Bridge");
end;
```

JadeTableColumn Class

The **JadeTableColumn** class provides access to an internally created column object that you can use to directly reference the properties and methods of the table column. This object is created on the first call to the **Table** class **accessColumn** method or the **JadeTableSheet** class **accessColumn** method for each **Table** control.

Using instances of the **JadeTableColumn** class is equivalent to setting the **accessMode** property of the **Table** control to the **Table.AccessMode_Column** value without having to set the **sheet** or **column** property.

One **JadeTableColumn** object only is created for each **Table** control, as there would be too much overhead required in creating an object for each column. This object is a proxy object that holds the reference to the column that was last accessed by using the **Table** class **accessColumn** method or **JadeTableSheet** class **accessColumn** method.

Accessing a column by using the **Table** class **accessColumn** method or **JadeTableSheet** class **accessColumn** method sets the **accessedColumn** property in the **Table** class, allowing subsequent access to that table column.

Storing a reference to a returned column causes problems unless you take a copy of that column, as there is only one such object. (Your logic must delete the cloned column.)

For details about the properties and methods defined in the **JadeTableColumn** class, see "JadeTableColumn Properties" and "JadeTableColumn Methods", in the following subsections. For details about the table control and the constants, properties, methods, and events that it provides, see "Table Class", in Chapter 2.

Inherits From: JadeTableElement

Inherited By: (None)

JadeTableColumn Properties

The properties defined in the **JadeTableColumn** class are summarized in the following table.

Property	Description
column	Contains the column number of the accessed column
maxColumnWidth	Contains the maximum width (in pixels) for a column when determining the width during the column width auto-size processing
sheet	Contains the sheet number of the accessed column
sortAsc	Specifies whether the sorting is ascending or descending
sortCased	Specifies whether sorting is case-sensitive
sortOrder	Specifies the precedence of the column for sorting (1 through 6) or zero to remove sorting on the column
sortType	Specifies the type of data the cell text represents during sorting
visible	Specifies the visibility of the column
width	Contains the size of the column
widthPercent	Contains the width of a column to a percentage of the client width of the table

column

Type: Integer

Availability: Read or write at any time

The **column** and **sheet** properties of the **JadeTableColumn** class define the column referenced by this object. These properties are set when the **Table** class **accessColumn** method or the **JadeTableSheet** class **accessColumn** method is called.

You can also set this property manually, allowing your logic to dynamically modify the column that is being referenced, as shown in the following example.

```
tableColumn := table1.accessColumn(10);
counter     := 10;
while counter >= 1 do
    tableColumn.column    := counter;
    tableColumn.inputType := Table.InputType_Numeric;
    counter               := counter - 1;
endwhile;
```

maxColumnWidth

Type: Integer

Availability: Read or write at any time

The **maxColumnWidth** property of the **JadeTableColumn** class specifies the maximum width (in pixels) for a column when determining the width during the column width auto-size processing.

The default value is zero (**0**), with values in the range zero (**0**) through **32767** pixels permitted. The default value of zero (**0**) means that there is no maximum width and the column will be as wide as required by the content if the column width is auto-sized. If the cell contains a long text string, the column will be as wide as is required to display the entire string.

If the value of the **maxColumnWidth** property is greater than zero (**0**) and column width is determined by the **autoSize** process, the width is restricted to a maximum value of the **maxColumnWidth** property.

The **maxColumnWidth** property is used only during the **autoSize** process and is ignored if the:

- **Table** class **autoSize**property value is not one of **AutoSize_Both**, **AutoSize_BothColumnMinimum**, **AutoSize_Column**, or **AutoSize_ColumnMinimum**.
- Column width has been set by logic (that is, set by the **Table** class **columnWidth** property or the **JadeTableColumn** class **width** property).
- Value of the **JadeTableColumn** class **widthPercent** property is not zero (**0**).

The **maxColumnWidth**property value does not prevent logic from setting a larger column width, nor does it prevent the user from resizing the column width to be larger than the value of the**width**property.

sheet

Type: Integer

Availability: Read or write at any time

The **sheet** and **column** properties of the **JadeTableColumn** class define the column referenced by this object. These properties are set when the **Table** class **accessColumn** or **JadeTableSheet** class **accessColumn** method is called.

You can also set this property manually, allowing your logic to dynamically modify the column that is being referenced.

sortAsc

Type: Boolean

Availability: Read or write at run time only

The **sortAsc** property of the **JadeTableColumn** class controls whether the column referenced by this object is sorted in ascending or descending order. The **sortAsc** property defaults to **true**.

This property is dependent on the column already being recorded as a sort column by the **sortOrder** property. For a description of this property, see the **Table** class **sortAsc** property.

The code fragment in the following example shows the use of the **sortAsc** property.

```
table1.accessColumn(2).sortOrder := 1;
table1.accessedColumn.sortAsc    := false;
```

sortCased

Type: Boolean

Availability: Read or write at run time only

The **sortCased** property of the **JadeTableColumn** class controls whether the column referenced by this object is sorted according to case. The **sortCased** property defaults to **false**.

This property is dependent on the column already being recorded as a sort column by the **sortOrder** property.

For a description of this property, see the **Table** control **sortCased** property.

The code fragment in the following example shows the use of the **sortCased** property.

```
table1.accessColumn(2).sortOrder := 1;
table1.accessedColumn.sortCased  := true;
```

sortOrder

Type: Integer

Availability: Read or write at run time only

The **sortOrder** property of the **JadeTableColumn** class contains the precedence of the column referenced by this object when sorting, in the range **1** through **3**, or it contains zero (**0**) to remove sorting on the current column.

For a description of this property, see the **Table** control **sortColumn** property. See also the **JadeTableColumn** class **sortAsc**, **sortCased**, and **sortType** properties, which are dependent on the column already being recorded as a sort column by the **sortOrder** property.

The code fragment in the following example shows the use of the **sortOrder** property.

```
table1.accessColumn(2).sortOrder := 1;   // first column in sort
table1.accessColumn(4).sortOrder := 2;   // second column
table1.accessColumn(5).sortOrder := 3;   // third column
```

sortType

Type: Integer

Availability: Read or write at run time only

The **sortType** property of the **JadeTableColumn** class contains the type of sorting that is performed on the column referenced by this object. This property is dependent on the column already being recorded as a sort column by the **sortOrder** property. For a description of this property, see the **Table** control **sortType** property.

The code fragment in the following example shows the use of the **sortType** property.

```
table1.accessColumn(2).sortOrder := 1; // first column in sort
table1.accessedColumn.sortType   := Table.SortType_Numeric;
```

visible

Type: Boolean

Availability: Read or write at any time

The **visible** property of the **JadeTableColumn** class specifies whether the column referenced by this object is displayed or hidden, or the visibility status to be obtained. For a description of this property, see the **Table** control **columnVisible** property.

The code fragment in the following example shows the use of the **visible** property.

```
if table1.accessColumn(indx).visible then
    table1.accessedColumn.backColor := Red;
endif;
```

width

Type: Integer

Availability: Read or write at any time

The **width** property of the **JadeTableColumn** class enables the size of the column referenced by this object to be accessed. For a description of this property, see the **Table** control **columnWidth** property.

The code fragment in the following example shows the use of the **width** property.

```
myTextBox.width := table1.accessColumn(table1.column).width;
```

widthPercent

Type: Real

Availability: Read or write at any time

The **widthPercent** property of the **JadeTableColumn** class contains the width of a column to a percentage of the client width of the table, as shown in the following example.

```
table.accessColumn(2).widthPercent := 20;
```

The default value of zero (**0.0**) indicates that the property does not apply, and the width of a column is then the default (set by the **Table**::**columnWidth** or **JadeTableColumn**::**width** property) or automatically calculated, by using the **Table** class **autoSize** property.

If a column has a positive **widthPercent** value, the width of the column is set to that percentage of the client width of the table. If the table width changes, this value is recalculated accordingly.

If all of the visible columns of a table have a non-zero **widthPercent** value and those values add up to 100 percent, the table is guaranteed not to have a horizontal scroll bar and the columns exactly fill the width of the table. It is not necessary for the values to add up to 100 percent or for all columns to have a value set.

To use mixed column width values in a table (that is, fixed-width and percent-width values), use the **widthPercentStyle** property of the **JadeTableSheet** class.

You can use this property in conjunction with the **Table** class **wordWrap** and **autoSize** properties. If word wrap applies to the cells of a column, setting the **Table** class **autoSize** property to **AutoSize_Row**, **AutoSize_Both**, or **AutoSize_BothColumnMinimum** determines the row height by using the column width for the word-wrapped text.

Notes Setting the **Table** class **columnWidth** property for a column resets the value of the **widthPercent** property to zero for that column.

If the user resizes a column manually, the value of the **widthPercent** property for that column is set to zero.

Setting the value of the **widthPercent** property to any value, including zero, clears any manual resizing of the column and causes the automatic width processes to apply again for that column.

JadeTableColumn Methods

The methods defined in the **JadeTableColumn** class are summarized in the following table.

Property	Description
delete	Deletes the **JadeTableColumn** object and the column reference in the table
findObject	Returns the row of the cell holding the specified object
findString	Searches the cells in a column of a table for the specified string
restoreAutoSize	Recalculates column widths, ignoring a width set by logic or by any user resize of that column

delete

Signature `delete() updating;`

The **delete** method deletes the column referenced by this object from the **Table** control.

A new **JadeTableColumn** object is created on the next call of the **Table** class **accessColumn** method or the **JadeTableSheet** class **accessColumn** method.

findObject

Signature `findObject(object: Object;`
`row: Integer io): Boolean;`

The **findObject** method of the **JadeTableColumn** class searches the **itemObject** property value of every cell in the column referenced by this object for the value specified in the **object** parameter, starting from the row specified in the **row** parameter value. If the **row** parameter contains a zero (**0**) value indicating that it is not specified, the row is treated as the first row; that is, the **row** value is set to **1**.

If the specified object is found, this method returns **true** and the row of the cell that contains the specified object. If the object is not found, this method returns **false**.

The code fragment in the following example shows the use of the **findObject** method.

```
bool := table1.accessColumn(table1.column).findObject(obj, row);
```

findString

Signature
```
findString(str:           String;
           row:           Integer io;
           caseSensitive: Boolean;
           exact:         Boolean): Boolean;
```

The **findString** method of the **JadeTableColumn** class searches the cells in a column of a **Table** control for the string specified in the **str** parameter. The search starts at the cell row specified by the **row** parameter or at the first row if the parameter value is less than **1**.

The case-sensitivity of the search is determined by the value of the **caseSensitive** parameter.

If the **exact** parameter value is **false**, the search matches any cell text with the specified prefix. If the **exact** parameter value is **true**, the search matches only cells whose text is an exact match.

If the search does not find a cell containing the string with the required **caseSensitive** and **exact** parameter values, this method returns **false**. If the search finds a cell containing the string with the required case-sensitive and exact options, this method returns **true** and the row of the located cell is returned; for example:

```
row := 0;
while table1.accessColumn(2).findString("city", row, false, false) do
    ....
    row := row + 1; // repeat the search starting in the next row
endwhile;
```

restoreAutoSize

Signature `restoreAutoSize();`

The **restoreAutoSize** method of the **JadeTableColumn** class results in the column width being re-evaluated, ignoring a width set by logic or by any user resize of that column.

JadeTableElement Class

The **Table** control provides access to each of an internally created sheet, row, column, and cell object; that is, the **JadeTableElement** class and its subclasses.

The **JadeTableElement** class encapsulates the behavior required to directly access the properties and methods of a table object (that is, a cell, column, row, or sheet) without using the **accessMode** property.

Using instances of the **JadeTableElement** subclasses is equivalent to setting the **accessMode** property of the **Table** control using the constant values listed in the following table without having to set the **row**, **column**, or **sheet** property of the **Table** control.

JadeTableElement Subclass	Equivalent to the accessMode Value for the Table class…
JadeTableCell	Table.AccessMode_Cell
JadeTableColumn	Table.AccessMode_Column
JadeTableRow	Table.AccessMode_Row
JadeTableSheet	Table.AccessMode_Sheet

As the overhead required to create an object for each cell, column, row, and sheet of the table would be too great, only one object of each type is created. These objects are proxy objects that hold a reference to the cell, column, row, or sheet that was last accessed by using the methods listed below. Accessing a cell, column, row, or sheet sets a corresponding property in the **Table** class, allowing subsequent access to that table element, as follows.

- **accessCell** method sets the **accessedCell** property to the returned cell
- **accessColumn** method sets **accessedColumn** property to the returned column
- **accessRow** method sets **accessedRow** property to the returned row
- **accessSheet** method sets **accessedSheet** property to the returned sheet

The following code fragments show examples of accessing the last table elements that were accessed.

```
table1.accessCell(2, 3).inputType := Table.InputType_TextBox;
table1.accessedCell.foreColor     := Red;

table1.accessSheet(2).accessCell(1, 4).text := "Company";
table1.accessedCell.alignment := Table.Alignment_Right_Middle;
```

Storing a reference to a returned cell causes problems unless you take a copy of that cell, as shown in the following example in which both **cell1** and **cell2** refer to the same object, which is referencing **cell(3, 4)**.

```
cell1 := table1.accessCell(2, 3);
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

In the following example, **cell1** has been cloned and still refers to **cell(2, 3)**.

```
cell1 := table1.accessCell(2, 3).cloneSelf(true);
// the cloned cell must be deleted by your logic
cell2 := table1.accessCell(3, 4);
cell1.text := "abc";
```

For details about the properties defined in the **JadeTableElement** class, see "JadeTableElement Properties", in the following subsection. For details about the table control and the constants, properties, methods, and events that it provides, see "Table Class", in Chapter 2.

Inherits From: Object

Inherited By: JadeTableCell, JadeTableColumn, JadeTableRow, JadeTableSheet

JadeTableElement Properties

The properties defined in the **JadeTableElement** class are summarized in the following table.

Property	Description
alignment	Contains the placement of text in a table element
backColor	Contains the background color of a table element
cellControl	Controls the input and display within a table element by defining a user-specified control that is placed over the table element when that element becomes current
comboList	Contains the list entries displayed in a combo box in the table element
decimals	Specifies that a cell can accept decimal input for table elements with a numeric text or signed numeric input type
editMask	Sets the mask used for edit mask input for a cell, row, column, or sheet
enabled	Specifies whether the cell, column, row, or sheet of the table can respond to user-generated events
fontBold	Specifies whether the font style is bold
fontItalic	Specifies whether the font style is italic
fontName	Contains the font name used for text displayed in a table element
fontSize	Contains the size of the font used for text displayed in a table element
fontStrikethru	Specifies whether the font style is strikethrough
fontUnderline	Specifies whether the font style is underlined
foreColor	Contains the foreground color used to display text in a table element
gridBottom	Specifies whether a grid line is drawn along the bottom edge of cells in a table element
gridRight	Specifies whether a grid line is drawn along the right-hand edge of cells in a table element
inputType	Contains the type of input (if any) that is accepted by a table element
itemObject	Contains an object for each table element
marginBottom	Contains the amount by which content is offset from the bottom edge of cells in a table element
marginLeft	Contains the amount by which content is offset from the left-hand edge of cells in a table element
marginRight	Contains the amount by which content is offset from the right-hand edge of cells in a table element

Property	Description
marginTop	Contains the amount by which content is offset from the top edge of cells in a table element
maxLength	Contains the amount of text that can be entered into a table element
partialTextIndication	Specifies whether to indicate that not all text can be displayed
selected	Contains the selected status of the table element
wordWrap	Specifies whether the text of a cell is displayed using word wrap when the width of the cell is less than the length of the text

alignment

Type: Integer

Availability: Read or write at any time

The **alignment** property of the **JadeTableElement** class sets the alignment of the text in a table element. For a description of this property, see the **Table** control **alignment** property.

The code fragment in the following example shows the use of the **alignment** property.

```
table.accessColumn(2).alignment := Table.Alignment_Center_Top;
```

backColor

Type: Integer

Availability: Read or write at any time

The **backColor** property of the **JadeTableElement** class determines the background color of a table element. For a description of this property, see the **Window** class **backColor** property.

The following example shows the use of the **backColor** property.

```
table1.accessRow(2).backColor := Red;
```

cellControl

Type: Control

Availability: Read or write at run time only

The **cellControl** property of the **JadeTableElement** class allows control over the input and display within the table by defining a user-supplied control that is placed over the cell when that cell becomes current.

For a description of this property, see the **Table** control **cellControl** property. See also the **inputType** property and the **Table** control **cellInputReady** event method.

The code fragment in the following example shows the use of the **cellControl** property.

```
table1.accessCell(2, 4).cellControl := myTextBox;
```

comboList

Type: String

Availability: Read or write at run time only (for a cell with **inputType** set to **3** only)

The **comboList** property of the **JadeTableElement** class contains the list entries displayed in a combo box in the table element.

This property applies only to a cell that has an effective **inputType** property set to **InputType_ComboBox** (3).

For a description of this property, see the **Table** control **comboList** property.

The code fragment in the following example shows the use of the **comboList** property.

```
table1.accessColumn(2).comboList := "one" & Tab & "two" & Tab & "three";
```

decimals

Type: Integer

Availability: Read or write at any time

The **decimals** property of the **JadeTableElement** class is used in conjunction with the **Table** control **inputType** property to indicate whether the text in an input cell is a **Decimal** primitive type numeric. For a description of this property, see the **Table** control **comboList** property.

The code fragment in the following example shows the use of the **decimals** property.

```
table1.accessColumn(2).decimals := 2;
```

editMask

Type: String

Availability: Read or write at any time

The **editMask** property of the **JadeTableElement** class sets the mask used for edit mask input for a table cell, row, column, or sheet when the value of the **Table** class **inputType** property is set to **InputType_EditMask** (7).

For more details, see the **Table** class **editMask** property.

enabled

Type: Boolean

Availability: Read or write at any time

The **enabled** property of the **JadeTableElement** class specifies whether the status of a cell, a row, a column, or all the cells of a sheet is enabled (that is, can respond to user-generated events).

By default, the value of the **enabled** property for **JadeTableCell**, **JadeTableColumn**, **JadeTableRow**, and **JadeTableSheet** subclasses is **true**.

If the effective enabled status of a cell is disabled (that is, any of the cell, row, column, or sheet **enabled** properties are **false**), the cell will not respond to mouse clicks, will not allow cell input, and will be skipped by any keyboard actions such as arrow, Page, Home, and End keys.

This property allows table cells, rows, columns, or sheets to be enabled or disabled at run time.

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **JadeTableElement** class specifies whether the font style of text in a table element is bold. For a description of this property, see the **Control** class **fontBold** property.

The code fragment in the following example shows the use of the **fontBold** property.

```
table1.accessCell(2, 3).fontBold := true;
```

fontItalic

Type: Boolean

Availability: Read or write at any time

The **fontItalic** property of the **JadeTableElement** class specifies whether the font style of text in a table element is italic. For a description of this property, see the **Control** class **fontItalic** property.

The code fragment in the following example shows the use of the **fontItalic** property.

```
table1.accessCell(2, 3).fontItalic := true;
```

fontName

Type: String[31]

Availability: Read or write at any time

The **fontName** property of the **JadeTableElement** class contains the font used to display text in a table element. For a description of this property, see the **Control** class **fontName** property.

The code fragment in the following example shows the use of the **fontName** property.

```
table1.accessCell(2, 3).fontName := "Arial";
```

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property of the **JadeTableElement** class contains the size of the font to be used for text displayed in a table element. For a description of this property, see the **Control** class **fontSize** property.

The code fragment in the following example shows the use of the **fontSize** property.

```
table1.accessCell(2, 3).fontSize := 9;
```

fontStrikethru

Type: Boolean

Availability: Read or write at any time

The **fontStrikethru** property of the **JadeTableElement** class specifies whether the font style for text displayed in a table element is strikethrough.

For a description of this property, see the **Control** class **fontStrikethru** property.

The code fragment in the following example shows the use of the **fontStrikethru** property.

```
table1.accessCell(2, 3).fontStrikethru := true;
```

fontUnderline

Type: Boolean

Availability: Read or write at any time

The **fontUnderline** property of the **JadeTableElement** class specifies whether the font style for text displayed in a table element text is underlined. For a description of this property, see the **Control** class **fontUnderline** property.

The code fragment in the following example shows the use of the **fontUnderline** property.

```
table1.accessCell(2, 3).fontUnderline := true;
```

foreColor

Type: Integer

Availability: Read or write at any time

The **foreColor** property of the **JadeTableElement** class contains the foreground color used to display text in a table element. For a description of this property, see the **Control** class **foreColor** property.

The code fragment in the following example shows the use of the **foreColor** property.

```
table1.accessRow(2).foreColor := Red;
```

gridBottom

Type: Boolean

Availability: Read or write at any time

The **gridBottom** property of the **JadeTableElement** class specifies whether a grid line is drawn along the bottom edge of cells in a table element. If the **gridLines** property for the table or the **JadeTableSheet** object is **false**, the property is ignored.

The code fragment in the following example shows the use of the **gridBottom** property to suppress the printing of horizontal grid lines for the sheet.

```
table1.accessSheet(1).gridBottom := false;
```

The value of the **gridBottom** property for a cell, which is **true** by default, can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The code fragment in the following example shows a change at the **JadeTableCell** level overriding a change at the **JadeTableSheet** level.

```
table1.accessCell(3,3).gridBottom := true;
table1.accessSheet(1).gridBottom := false;
```

Note When the value of the **gridLines** property is **true**, a pixel is always used for the grid to the right and bottom of each cell, regardless of whether it is drawn or not. Grid lines are never drawn for fixed cells drawn as three-dimensional (3D) buttons, so the value of **gridBottom** has no effect on those cells.

gridRight

Type: Boolean

Availability: Read or write at any time

The **gridRight** property of the **JadeTableElement** class specifies whether a grid line is drawn along the right edge of cells in a table element. If the **gridLines** property for the table or the **JadeTableSheet** object is **false**, the property is ignored.

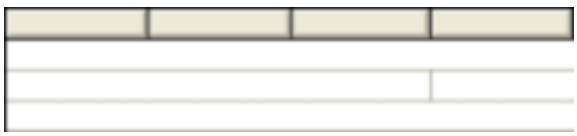
The code fragment in the following example shows the use of the **gridRight** property to suppress the printing of vertical grid lines for the sheet.

```
table1.accessSheet(1).gridRight := false;
```

The value of the **gridRight** property for a cell, which is **true** by default, can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The code fragment in the following example shows a change at the **JadeTableCell** level overriding a change at the **JadeTableSheet** level.

```
table1.accessCell(3,3).gridRight := true;
table1.accessSheet(1).gridRight := false;
```



Note When the value of the **gridLines** property is **true**, a pixel is always used for the grid to the right and bottom of each cell, regardless of whether it is drawn or not. Grid lines are never drawn for fixed cells drawn as three-dimensional (3D) buttons, so the value of **gridRight** has no effect on those cells.

inputType

Type: Integer

Availability: Read or write at run time only

The **inputType** property of the **JadeTableElement** class contains the type of input (if any) that is accepted by a table element.

For a description of this property, see the **Table** control **inputType** property.

The code fragment in the following example shows the use of the **inputType** property.

```
table1.accessColumn(2).inputType := Table.InputType_Numeric;
```

itemObject

Type: Object array

Availability: Read or write at run time only

The **itemObject** property of the **JadeTableElement** class enables you to store an object with each element of a table. This allows an object to be stored with each sheet, row, column, and cell of the table.

For a description of this property, see the **Table** control **itemObject** property.

Note As the object reference that is stored is of the **Object** class, you may then need to cast it to the required class so that it can be used. (For details about type casting, see "Type Casts", in Chapter 1 of the *JADE Developer's Reference*.)

The following example shows the use of the **itemObject** property.

```
vars
    atest : Atest;
begin
    beginTransaction;
    atest := table1.accessRow(table1.row).itemObject.Atest;
    delete atest;
    commitTransaction;
end;
```

marginBottom

Type: Integer

Availability: Read or write at run time only

The **marginBottom** property of the **JadeTableElement** class contains the amount by which the cell content is offset from the bottom edge for all cells in a table element. Effectively, the margins create a rectangle inside the cell into which the image, text, or cell control is placed.

The value of the **marginBottom** property for a cell is initially *unset* but can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The value is in the range **0** through **255**, with the value **255** having the special meaning of returning the property to its *unset* state. If no value has been set, the default value is zero (**0**).

If the **autoSize** property is set for the table, changing the value of the **marginBottom** property affects the height of the row unless the **rowHeight** property has been specifically set by logic.

The code fragment in the following example sets the bottom margin of **3** pixels for all cells in the sheet.

```
table1.accessSheet(1).marginBottom := 3;
```

Note When merging cells, the **marginBottom** value of the first cell in the range is used.

marginLeft

Type: Integer

Availability: Read or write at run time only

The **marginLeft** property of the **JadeTableElement** class contains the amount by which the cell content is offset from the left-hand edge for all cells in a table element. Effectively, the margins create a rectangle inside the cell into which the image, text, or cell control is placed.

The value of the **marginLeft** property for a cell is initially *unset* but can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The value is in the range **0** through **255**, with the value **255** having the special meaning of returning the property to its *unset* state. If no value has been set, the default value is zero (**0**).

If the **autoSize** property is set for the table, changing the value of the **marginLeft** property affects the width of the column unless the **columnWidth** property has been specifically set by logic.

The code fragment in the following example sets the left-hand margin of **3** pixels for all cells in the sheet.

```
table1.accessSheet(1).marginLeft := 3;
```

Note When merging cells, the **marginLeft** value of the first cell in the range is used.

marginRight

Type: Integer

Availability: Read or write at run time only

The **marginRight** property of the **JadeTableElement** class contains the amount by which the cell content is offset from the right-hand edge for all cells in a table element. Effectively, the margins create a rectangle inside the cell into which the image, text, or cell control is placed.

The value of the **marginRight** property for a cell is initially *unset* but can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The value is in the range **0** through **255**, with the value **255** having the special meaning of returning the property to its *unset* state. If no value has been set, the default value is zero (**0**).

If the **autoSize** property is set for the table, changing the value of the **marginRight** property affects the width of the column unless the **columnWidth** property has been specifically set by logic.

The code fragment in the following example sets the right-hand margin of **3** pixels for all cells in the sheet.

```
table1.accessSheet(1).marginRight := 3;
```

Note When merging cells, the **marginLeft** value of the first cell in the range is used.

marginTop

Type: Integer

Availability: Read or write at run time only

The **marginTop** property of the **JadeTableElement** class contains the amount by which the cell content is offset from the top edge for all cells in a table element. Effectively, the margins create a rectangle inside the cell into which the image, text, or cell control is placed.

The value of the **marginTop** property for a cell is initially *unset* but can be changed through code at increasingly more-specific levels: **JadeTableSheet**, **JadeTableRow**, **JadeTableColumn**, and **JadeTableCell**. Where there are conflicting changes, the most-specific change determines the resulting value of the property.

The value is in the range **0** through **255**, with the value **255** having the special meaning of returning the property to its *unset* state. If no value has been set, the default value is zero (**0**).

If the **autoSize** property is set for the table, changing the value of the **marginTop** property affects the height of the row unless the **rowHeight** property has been specifically set by logic.

The code fragment in the following example sets the top margin of **3** pixels for all cells in the sheet.

```
table1.accessSheet(1).marginTop := 3;
```

Note When merging cells, the **marginTop** value of the first cell in the range is used.

maxLength

Type: Integer

Availability: Read or write at any time

The **maxLength** property of the **JadeTableElement** class contains the maximum amount of text that can be entered into a table element that has the **inputType** property set to the **Table** class constant value of **InputType_TextBox** (2), **InputType_TextNumeric** (4), or **InputType_SignedNumeric** (6). For a description of this property, see the **Table** control **maxLength** property.

The code fragment in the following example shows the use of the **maxLength** property.

```
table1.accessColumn(2).maxLength := 6;
```

partialTextIndication

Type: Boolean

Availability: Read or write at run time only

The **partialTextIndication** property of the **JadeTableElement** class specifies whether an indication is displayed when there is insufficient room to show all of the text in a cell. For a description of this property, see the **Table** control **partialTextIndication** property. For details about word wrapping when displaying text in a table cell, see the **Table** class **wordWrap** property.

The code fragment in the following example shows the use of the **partialTextIndication** property.

```
table1.accessSheet(1).partialTextIndication := true;
```

selected

Type: Boolean

Availability: Read or write at run time

The **selected** property of the **JadeTableElement** class accesses the **selected** status of a **Table** element, as follows.

- Retrieving the selected status
  - **JadeTableCell** class **selected** property returns whether the referenced cell is selected.
  - **JadeTableColumn** class **selected** property returns whether all non-fixed cells of the column are selected.
  - **JadeTableRow** class **selected** property returns whether all non-fixed cells of the row are selected.
  - **JadeTableSheet** class **selected** property returns whether all non-fixed cells of the whole sheet are selected.
- Setting the selected status
  - **JadeTableCell** class **selected** property sets whether the referenced cell is selected.
  - **JadeTableColumn** class **selected** property sets the selected status of all non-fixed cells in the column.
  - **JadeTableRow** class **selected** property sets the selected status of all non-fixed cells in the row.
  - **JadeTableSheet** class **selected** property sets the selected status of all non-fixed cells in the whole sheet.

For a description of this property, see the **Table** control **selected** property.

The code fragment in the following example shows the use of the **selected** property.

```
table1.accessCell(2, 3).selected := true; // set selected status of a cell
table1.accessColumn(4).selected := true;  // set selected status of a column
table1.accessRow(6).selected    := true;  // set selected status of a row
if table1.accessSheet(1).selected then    // are all cells selected?
```

wordWrap

Type: Boolean

Availability: Read or write at run time only

The **wordWrap** property of the **JadeTableElement** class specifies whether the text of a cell is displayed using word wrap when the width of the cell is less than the length of the text.

For a description of this property, see the **Table** control **wordWrap** property.

JadeTableRow Class

The **JadeTableRow** class provides access to an internally created row object that you can use to directly reference the properties and methods of the table row. This object is created on the first call to the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method for each **Table** control.

Using instances of the **JadeTableRow** class is equivalent to setting the **accessMode** property of the **Table** control to the **Table.AccessMode_Row** value without having to set the **row** or **sheet** properties.

One **JadeTableRow** object only is created for each **Table** control, as there would be too much overhead in creating an object for each row. This object is a proxy object that holds the reference to the row that was last accessed by the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method.

Accessing a row by using the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method sets the **accessedRow** property in the **Table** class, allowing subsequent access to that table row.

Storing a reference to a returned row causes problems unless you take a copy of that row, as there is one row object only. (Your logic must delete the cloned row.)

For details about the properties and methods defined in the **JadeTableRow** class, see "JadeTableRow Properties" and "JadeTableRow Methods", in the following subsections. For details about the table control and the constants, properties, methods, and events that it provides, see "Table Class", in Chapter 2.

Inherits From: JadeTableElement

Inherited By: (None)

JadeTableRow Properties

The properties defined in the **JadeTableRow** class are summarized in the following table.

Property	Description
height	Contains the size of the row
row	Contains the row number of the accessed row
sheet	Contains the sheet number of the accessed row
visible	Specifies whether the current row is displayed or hidden, or the visibility status

height

Type: Integer

Availability: Read or write at run time only

The **height** property of the **JadeTableRow** class enables the size of the row referenced by this object to be accessed.

For a description of this property, see the **Table** control **rowHeight** property.

The code fragment in the following example shows the use of the **height** property.

```
myTextBox.height := table1.accessRow(table.row).height;
```

row

Type: Integer

Availability: Read or write at any time

The **row** property and the **sheet** property of the **JadeTableRow** class define the row referenced by this object. These properties are set when the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method is called.

You can also set this property manually, allowing your logic to dynamically modify the row that is being referenced, as shown in the following example.

```
tableRow := table1.accessRow(table1.rows);
counter  := table1.rows;
while counter >= 1 do
    tableRow.row     := counter;
    tableRow.visible := true;
    counter          := counter - 1;
endwhile;
```

sheet

Type: Integer

Availability: Read or write at any time

The **row** property and the **sheet** property of the **JadeTableRow** class define the row referenced by this object. These properties are set when the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method is called.

You can also set this property manually, allowing your logic to dynamically modify the sheet that is being referenced.

visible

Type: Boolean

Availability: Read or write at any time

The **visible** property of the **JadeTableRow** specifies whether the row referenced by this object is displayed or hidden, or the visibility status to be obtained.

For a description of this property, see the **Table** control **rowVisible** property.

The code fragment in the following example shows the use of the visible property.

```
if table1.accessRow(indx).visible then
    table1.accessedRow.backColor := Red;
endif;
```

JadeTableRow Methods

The methods defined in the **JadeTableRow** class are summarized in the following table.

Method	Description
delete	Deletes the entire row from the sheet
findObject	Returns the column of the cell holding the specified object
findString	Searches the cells in a row of a table for the specified string
restoreAutoSize	Recalculates row heights, ignoring a height set by logic or by any user resize of that row

delete

Signature `delete() updating;`

The **delete** method of the **JadeTableRow** class deletes the row referenced by this object from the **Table** control.

A new **JadeTableRow** object is created on the next call of the **Table** class **accessRow** method or the **JadeTableSheet** class **accessRow** method.

findObject

Signature
```
findObject(object: Object;
           column: Integer io): Boolean;
```

The **findObject** method of the **JadeTableRow** class searches the **itemObject** property value of every cell in the row referenced by this object for the value specified in the **object** parameter, starting from the column specified in the **column** parameter value.

If the **column** parameter contains a zero (**0**) value indicating that it is not specified, the column is treated as the first column; that is, the **column** value is set to **1**.

If the specified object is found, this method returns **true** and the column of the cell that contains the specified object. If the object is not found, this method returns **false**.

The code fragment in the following example shows the use of the **findObject** method.

```
bool := table1.accessRow(table1.row).findObject(obj, column);
```

findString

Signature
```
findString(str:           String;
           row:           Integer io;
           caseSensitive: Boolean;
           exact:         Boolean): Boolean;
```

The **findString** method of the **JadeTableRow** class searches the cells in a row of a **Table** control for the string specified in the **str** parameter. The search starts at the cell column specified by the **column** parameter or at the first column if the parameter value is less than **1**.

The case-sensitivity of the search is determined by the value of the **caseSensitive** parameter.

If the **exact** parameter value is **false**, the search matches any cell text with the specified prefix. If the **exact** parameter value is **true**, the search matches only cells whose text is an exact match.

If the search does not find a cell containing the string with the required **caseSensitive** and **exact** parameter values, this method returns **false**. If the search finds a cell containing the string with the required case-sensitive and exact options, this method returns **true** and the column of the located cell is returned; for example:

```
col := 0;
while table1.accessRow(2).findString("city", col, false, false) do
    ....
    col := col + 1; // repeat the search starting in the next column
endwhile;
```

restoreAutoSize

Signature `restoreAutoSize();`

The **restoreAutoSize** method of the **JadeTableRow** class results in the row height being re-evaluated, ignoring a height set by logic or by any user resize of that row.

JadeTableSheet Class

The **JadeTableSheet** class provides access to an internally created sheet object that you can use to directly reference the properties and methods of the table sheet. This object is created on the first call to the **Table** class **accessSheet** method for each **Table** control. Using instances of the **JadeTableSheet** class is equivalent to setting the **accessMode** property of the **Table** control to the **Table.AccessMode_Sheet** value without having to set the **Table** control **sheet** property.

One **JadeTableSheet** object only is created, which is a proxy object that holds the last reference to the sheet that was last accessed.

Accessing a sheet by using the **Table** class **accessSheet** method sets the **accessedSheet** property in the **Table** class, allowing subsequent access to that table sheet.

The following code fragment shows an example of these methods.

```
table1.accessSheet(1).caption     := "Company";
table1.accessedSheet.columns      := 5;
table1.accessedSheet.fixedColumns := 0;
table1.accessSheet(2).caption     := "Group";
table1.accessedSheet.columns      := 3;
```

Storing a reference to a returned sheet causes problems unless you take a copy of that sheet, as there is one sheet object only. (Your logic must delete the cloned sheet.)

For details about the properties and methods defined in the **JadeTableSheet** class, see "JadeTableSheet Properties" and "JadeTableSheet Methods", in the following subsections. For details about the table control and the constants, properties, methods, and events that it provides, see "Table Class", in Chapter 2.

Inherits From: JadeTableElement

Inherited By: (None)

JadeTableSheet Properties

The properties defined in the **JadeTableSheet** class are summarized in the following table.

Property	Description
alternatingRowBackColor	Specifies an alternate row background color
alternatingRowBackColorCount	Specifies the number of table rows at which the alternating background color of each visible non-fixed row and non-fixed cell is displayed
caption	Contains the caption for the sheet
column	Contains the current column on the sheet
columns	Contains the number of columns on the sheet
currentRowImage	Contains the image to display in the first fixed cell of the current row
displaySorting	Specifies whether a column used for sorting displays a sorting indicator in the fixed cell
extendedColumn	Specifies that when a table is auto-sized, the specified column is enlarged to use the remaining space

Property	Description
fixed3D	Specifies whether a three-dimensional (3D) button image is painted on the cells in the fixed area
fixedColumns	Contains the number of fixed columns in the sheet
fixedRows	Contains the number of fixed rows in the sheet
gridColor	Contains the color of grid lines
gridLines	Specifies whether lines are drawn between the rows and columns of the sheet
leftColumn	Contains the column that is displayed at the left edge of the non-fixed area of the sheet
myTable	Contains a reference to the table that owns the sheet
pixelHorzScrollIncrement	Specifies the number of pixels that are scrolled horizontally when the scrolling mode of the sheet is pixels
pixelVertScrollIncrement	Specifies the number of pixels that are scrolled vertically when the scrolling mode of the sheet is pixels
row	Contains the current row on the sheet
rows	Contains the number of rows on the sheet
scrollBars	Specifies whether the sheet has horizontal or vertical scroll bars when required
scrollHorzPos	Contains the number of pixels that the current left column is scrolled
scrollMode	Specifies how the table scrolls when using the mouse
scrollVertPos	Contains the number of pixels that the current top row is scrolled
sheet	Contains the sheet number of the sheet being accessed
showCurrentRowImage	Contains the image to display in the first fixed cell of the current row
showPartialTextBubbleHelp	Specifies whether the full text is displayed in a cell in which the text is not fully visible when the user moves the cursor over the cell
tabInitialPosition	Specifies how the table **row** and **column** properties are set when tabbing into a table
tabOffEnds	Specifies the result of tabbing out of the last visible, enabled cell of the sheet
topRow	Contains the row that is displayed at the top edge of the non-fixed area of the sheet
visible	Specifies whether the sheet is displayed or hidden, or the visibility status
widthPercentStyle	Specifies whether fixed columns are included or excluded when calculating the percentage widths of columns in a sheet

alternatingRowBackColor

Type: Integer

Availability: Read or write run time only

The **alternatingRowBackColor** property of the **JadeTableSheet** class specifies an alternate row background color. By default, alternating table rows have a background color of **Azure**. When you set this property to a value other than **Azure**, the specified value is used as the default background color of each alternate non-fixed row.

If the value of the **alternatingRowBackColorCount** property is **2**, the first, third, and so on non-fixed row default background color is the **backColor** property value of the sheet. The second, fourth, and so on non-fixed row default background color is the **alternatingRowBackColor** property value when it is not the default value (otherwise the **backColor** property value of the sheet is used).

If the value of the **backColor** property of a cell, row, or column is specifically set and it is not **#800000000** (that is, transparent), the default value of the cell is ignored and the specific value of the **backColor** property is used.

Note When a cell is drawn, the **backColor** property value is overridden by any specified **backColor** value set for that cell, its row, or its column.

Note that when the table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;
```

alternatingRowBackColorCount

Type: Integer

Availability: Read or write at run time only

The **alternatingRowBackColorCount** property of the **JadeTableSheet** class specifies the number of table rows at which the alternating background color of each visible non-fixed row and non-fixed cell is displayed.

If the value of the **alternatingRowBackColorCount** property is:

- Less than or equal to zero (**0**), the background color of each non-fixed cell defaults to the value of the **backColor** property of the sheet, or of the table itself if the value of the sheet is not specifically set. The **alternatingRowBackColor** property value is ignored.
- Greater than zero (**0**), for each visible **alternatingRowBackColorCount** non-fixed row and non-fixed cell, the background color defaults to the value of the **alternatingRowBackColor** property.

For example, if the count is **2**, the first, third, fifth, and so on non-fixed rows and the non-fixed cells in that row default to the value of the **backColor** property of the sheet, while the second, fourth, sixth, and so on non-fixed rows and the non-fixed cells in that row default to the value of the **alternatingRowBackColor** property.

Note When the table is scrolled, the colors do not move with a row. The color scheme is applied to the rows, starting with the first visible non-fixed row; for example:

```
table1.accessSheet(1).alternatingRowBackColorCount := 2;
table1.accessSheet(1).alternatingRowBackColor := Azure;
```

caption

Type: String

Availability: Read or write at run time only

The **caption** property of the **JadeTableSheet** class contains the text for the caption of the sheet referenced by this object. This caption is displayed in the tabs area of the current table.

For a description of this property, see the **Table** control **sheetCaption** property.

The code fragment in the following example shows the use of the **caption** property.

```
table1.accessSheet(2).caption := "Company";
```

column

Type: Integer

Availability: Read or write at run time only

The **column** property of the **JadeTableSheet** class contains the current column on the sheet referenced by this object.

For a description of this property, see the **Table** control **column** property.

The code fragment in the following example shows the use of the **column** property.

```
table1.accessSheet(2).column := 2;
```

columns

Type: Integer

Availability: Read or write at run time only

The **columns** property of the **JadeTableSheet** class contains the number of columns on the sheet referenced by this object.

For a description of this property, see the **Table** control **columns** property.

The code fragment in the following example shows the use of the **columns** property.

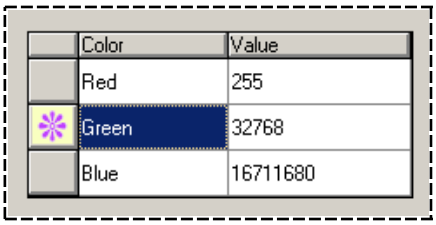
```
table1.accessSheet(2).columns := 5;
```

currentRowImage

Type: Binary

Availability: Read or write at run time only

The **currentRowImage** property of the **JadeTableSheet** class contains a binary image to be displayed in the fixed cell at the start of the current row, as shown in the following image.



The image is displayed only when the **showCurrentRowImage** property of the **JadeTableSheet** class is set to **true** and the table has a fixed column. A default image is used if the value of the **currentRowImage** property is null (the default).

The image is displayed as if the **stretch** property of the table were set to **Stretch_None_Picture_First** (2) and the cell text is displayed as if the value of the **partialTextIndication** property is **true**.

Note Row heights and the width of the first column could be affected by this image. The spacing between the image and the text is 3 through 15 pixels, depending on the extra space available.

The code fragment in the following example shows the use of the **currentRowImage** property.

```
table1.accessSheet(1).currentRowImage := app.loadPicture("c:\select.png");
table1.accessSheet(1).showCurrentRowImage := true;
```

displaySorting

Type: Integer

Availability: Read or write at run time only

The **displaySorting** property of the **JadeTableSheet** class displays a sorting indicator after the text in the fixed row cells of a table to indicate an ascending or descending sort order for the column. The image for the sorting indicator is an up or down arrow.

The property is ignored if the sheet does not have a fixed row or the cells in the fixed row already contain a picture.

The values for **displaySorting** and their effects are summarized in the following table.

Table Class Constant	Value	Description
DisplaySorting_None	0	Default value, which indicates no sorting indicator is displayed.
DisplaySorting_First	1	First sort column displays a sorting indicator in the cell in the first fixed row, depending on the **sortAsc** property for the column.
DisplaySorting_AllColumns	2	Displays a sorting indicator for all sort columns.

Table Class Constant	Value	Description
DisplaySorting_Numbers	3	Displays a sorting indicator for all sort columns where the image includes an integer that indicates the sort preference in the range 1 through 6.

When display sorting is specified:

- The spacing of the image from the text varies, depending on the space available, and it is in the range 3 through 15 pixels.
- The arrow is displayed after any cell text and the image takes precedence over the text.
- The picture is displayed as if the **stretch** property is set to **Stretch_None_Picture_First** (2) and the cell text is displayed as if the **partialTextIndication** property is set to **true**.

  Note that the display is based on the value of the **sortAsc** property for the cell, regardless of how the table is actually sorted. If the cell has no text, the arrow is centered; otherwise the cell and arrow are displayed according to the effective **alignment** property of the cell.

Setting the value of the **displaySorting** property affects the size of a column that has the **autoSize** property set, to allow for the size of the sorting indicator. If you change the width of a column, the size of the image does not change and any text in the cell is truncated, if necessary.

The code fragment in the following example shows the use of the **displaySorting** property.

```
table1.sortColumn[1] := 1;
table1.sortColumn[2] := 3;
table1.sortAsc[2] := false;
table1.accessSheet(1).displaySorting := Table.DisplaySorting_AllColumns;
```

extendedColumn

Type: Integer

Availability: Read or write at run time only

The **extendedColumn** property of the **JadeTableSheet** class specifies when the **Table** class **autoSize** property is set to **true**, after the table columns are auto-sized to fit the minimum size for their contents and the total width of the visible columns is less than the value of the **clientWidth** property, the specified column is enlarged to use the remaining space. The value of the **JadeTableColumn** class **maxColumnWidth** property is ignored.

The value of the **extendedColumn** property is ignored unless all of the following are **true**.

1. The **autoSize** property is set to **AutoSize_BothColumnMinimum** or **AutoSize_ColumnMinimum**.
2. The **extendedColumn** property is set to a valid visible column number.
3. The **columnWidth** property has not been set manually by the user or by logic.
4. The total width of all visible columns is less than the value of the **clientWidth** property of the table.

The default value is zero (**0**).

fixed3D

Type: Boolean

Availability: Read or write at any time

The **fixed3D** property of the **JadeTableSheet** class specifies whether a three-dimensional (3D) button image is painted on the cells in the fixed area of the sheet referenced by this object.

For a description of this property, see the **Table** control **fixed3D** property.

The code fragment in the following example shows the use of the **fixed3D** property.

```
table1.accessSheet(2).fixed3D := false;
```

fixedColumns

Type: Integer

Availability: Read or write at any time

The **fixedColumns** property of the **JadeTableSheet** class contains the number of fixed columns in the sheet referenced by this object.

For a description of this property, see the **Table** control **fixedColumns** property.

The code fragment in the following example shows the use of the **fixedColumns** property.

```
table1.accessSheet(2).fixedColumns := 2;
```

fixedRows

Type: Integer

Availability: Read or write at any time

The **fixedRows** property of the **JadeTableSheet** class contains the number of fixed rows in the sheet referenced by this object.

For a description of this property, see the **Table** control **fixedRows** property.

The code fragment in the following example shows the use of the **fixedRows** property.

```
table1.accessSheet(2).fixedRows := 1;
```

gridColor

Type: Integer

Availability: Read or write at any time

The **gridColor** property of the **JadeTableSheet** class contains the color of grid lines in the sheet referenced by this object.

For a description of this property, see the **Table** control **gridColor** property.

The code fragment in the following example shows the use of the **gridColor** property.

```
table1.accessSheet(2).gridColor := Red;
```

gridLines

Type: Boolean

Availability: Read or write at any time

The **gridLines** property of the **JadeTableSheet** class specifies whether lines are drawn between the rows and columns of the sheet referenced by this object.

For a description of this property, see the **Table** control **gridLines** property.

The code fragment in the following example shows the use of the **gridLines** property.

```
table1.accessSheet(2).gridLines := false;
```

leftColumn

Type: Integer

Availability: Read or write at run time

The **leftColumn** property of the **JadeTableSheet** class contains the column that is displayed at the left edge of the non-fixed area of the sheet referenced by this object.

For a description of this property, see the **Table** control **leftColumn** property.

The code fragment in the following example shows the use of the **leftColumn** property.

```
table1.accessSheet(2).leftColumn := 2;
```

myTable

Type: Table

Availability: Read or write at run time only

The **myTable** property of the **JadeTableSheet** class contains a reference to the **Table** control that owns the sheet referenced by this object.

pixelHorzScrollIncrement

Type: Integer

Availability: Read or write at any time

The **pixelHorzScrollIncrement** property of the **JadeTableSheet** class specifies the number of pixels that are scrolled horizontally when the scrolling mode of the sheet is pixels (that is, the value of the **JadeTableSheet** class **scrollMode** property is **ScrollMode_HorzPixel_VertCell** (1) or **ScrollMode_Both_Pixel** (3).

The default value is **1**, and the value can be in the range **1** through **32767**. Values outside of this range are treated as **1**.

The property enables the amount of horizontal scrolling to be increased. For example, by setting the horizontal pixel increment to 10 pixels at a time, the scrolling would scroll ten pixels.

The increment value is used only when the user clicks on a horizontal scroll bar arrow or scrolls using the mouse wheel and the scrolling mode is pixels for that scroll bar.

Note When scrolling with the mouse wheel, the scrolling amount is multiplied by an increment set by the user. (This increment is usually **3**.)

The **pixelHorzScrollIncrement** property value is ignored unless the value of the **scrollMode** property of the sheet is **ScrollMode_Both_Pixel** or **ScrollMode_HorzPixel_VertCell**.

The code fragment in the following example shows the use of the **pixelHorzScrollIncrement** property.

```
table1.accessSheet(1).pixelHorzScrollIncrement := 15;
```

pixelVertScrollIncrement

Type: Integer

Availability: Read or write at any time

The **pixelVertScrollIncrement** property of the **JadeTableSheet** class specifies the number of pixels that are scrolled vertically when the scrolling mode of the sheet is pixels (that is, the value of the **JadeTableSheet** class **scrollMode** property is **ScrollMode_VertPixel_VertCell** or **ScrollMode_Both_Pixel** (3).

The default value is **1**, and the value can be in the range **1** through **32767**. Values outside of this range are treated as **1**.

The **pixelVertScrollIncrement** property enables the amount of vertical scrolling to be increased. For example, by setting the vertical pixel increment to the height of the displayed text in the table, the scrolling would scroll a line at a time.

The increment value is used only when the user clicks on a vertical scroll bar arrow or scrolls using the mouse wheel and the scrolling mode is pixels for that scroll bar.

Note When scrolling with the mouse wheel, the scrolling amount is multiplied by an increment set by the user. (This increment is usually **3**.)

The **pixelVertScrollIncrement** value is ignored unless the value of the **scrollMode** property of the sheet is **ScrollMode_Both_Pixel** or **ScrollMode_VertPixel_HorzCell**.

The code fragment in the following example shows the use of the **pixelVertScrollIncrement** property.

```
table1.accessSheet(1).pixelVertScrollIncrement := 14;
```

row

Type: Integer

Availability: Read or write at run time only

The **row** property of the **JadeTableSheet** class contains the current row on the sheet referenced by this object.

For a description of this property, see the **Table** control **row** property.

The code fragment in the following example shows the use of the **row** property.

```
table1.accessSheet(2).row := 3;
```

rows

Type: Integer

Availability: Read or write at run time only

The **rows** property of the **JadeTableSheet** class contains the number of rows on the sheet referenced by this object.

For a description of this property, see the **Table** control **rows** property.

The code fragment in the following example shows the use of the **rows** property.

```
table1.accessSheet(2).rows := 9;
```

scrollBars

Type: Integer

Availability: Read or write at any time

The **scrollBars** property of the **JadeTableSheet** class determines whether the sheet referenced by this object has horizontal or vertical scroll bars.

For a description of this property, see the **Table** control **scrollBars** property.

The code fragment in the following example shows the use of the **scrollBars** property.

```
table1.accessSheet(2).scrollBars := ScrollBars_None;
```

scrollHorzPos

Type: Integer

Availability: Read or write at any time

The **scrollHorzPos** property of the **JadeTableSheet** class is set to the number of pixels that the current left column (identified by the **leftColumn** property of the **Table** control) is scrolled. The **scrollHorzPos** property is reset to zero (**0**) when the value of the **leftColumn** property is set by logic. To establish a scrolled position by logic, set the value of the **leftColumn** property and then that of the **scrollHorzPos** property.

The value of the **scrollHorzPos** property must be less than the width of the **leftColumn** column. (The **leftColumn** property value applies to the first cell to the right of the fixed columns, regardless of how much of that cell is on view.)

The code fragment in the following example shows the use of the **scrollHorzPos** property.

```
// Enable scrolling by pixel for a sheet
table1.accessSheet(2).scrollMode := Table.ScrollMode_Both_Pixel;
// Select a column as the 'left' column of the sheet
table1.sheet := 2;
table1.leftColumn := 5;
// Scroll the selected column
table1.accessSheet(2).scrollHorzPos := 10;
```

scrollMode

Type: Integer

Availability: Read or write at any time

The **scrollMode** property of the **JadeTableSheet** class determines how the table scrolls when using the mouse, determined by the **Table** control class constants listed in the following table.

Table Class Constant	Description
ScrollMode_Cell (0)	The default, indicating the table scrolls vertically and horizontally by cell
ScrollMode_HorzPixel_VertCell (1)	Allows horizontal scrolling by pixel and vertical scrolling by cell
ScrollMode_VertPixel_HorzCell (2)	Allows vertical scrolling by pixel and horizontal scrolling by cell
ScrollMode_Both_Pixel (3)	Allows vertical and horizontal scrolling by pixel

Note Using the arrow or page keys to move around the table always scrolls by cell, regardless of the value of the **scrollMode** property. Pixel scrolling occurs only when the table is scrolled using the mouse.

The code fragment in the following example shows the use of the **scrollMode** property.

```
table1.accessSheet(2).scrollMode := Table.ScrollMode_Both_Pixel;
```

Note Scrolling by pixel is much slower than scrolling by cell.

A scroll event is generated for every scroll position generated, which could significantly increase thin client traffic if the scrolled event is defined. Other than the **scrolled** event, thin client traffic is not affected by the value of the **scrollMode** property.

When a cell control is displayed on a partially hidden cell, the cell control is only partly shown. To achieve this, when scrolling by pixel is enabled, an extra window layer is inserted between the table and the cell control. This is transparent, and is mentioned only in case your code is performing some very unusual logic that may be affected.

When using the **displayCollection** method and vertical pixel scrolling, the scroll bar thumb size and position may vary unusually when scrolling if there are variable height rows, because the scroll bar has to estimate the number of pixels for all of the collection entries.

scrollVertPos

Type: Integer

Availability: Read or write at any time

The **scrollVertPos** property of the **JadeTableSheet** class is set to the number of pixels that the current top row (identified by the **topRow** property of the **Table** control) is scrolled. The **scrollVertPos** property is reset to zero (**0**) when the value of the **topRow** property is set by logic. To establish a scrolled position by logic, set the value of the **topRow** property and then that of the **scrollVertPos** property.

The value of the **scrollVertPos** property must be less than the height of the **topRow** column. (The **topRow** property value applies to the first cell below the fixed rows, regardless of how much of that cell is on view.)

The code fragment in the following example shows the use of the **scrollVertPos** property.

```
// Enable scrolling by pixel for this sheet
table1.accessSheet(2).scrollMode := Table.ScrollMode_Both_Pixel;
```

```
// Select the top row
table1.sheet := 2;
table1.topRow := 3;
// Scroll the selected row
table1.accessSheet(2).scrollVertPos := 5;
```

sheet

Type: Integer

Availability: Read or write at any time

The **sheet** property of the **JadeTableSheet** class contains the last sheet requested by the **accessSheet** method of the **Table** class. This value is used to determine the required sheet when a property or method of the **JadeTableSheet** object is referenced. (See also the **Table** control **sheet** property.)

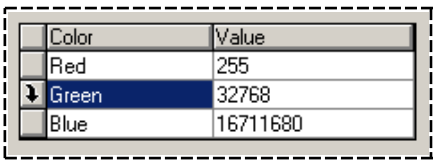
Your JADE logic can change the value of this property dynamically, allowing control over the sheet that is accessed by this object.

showCurrentRowImage

Type: Boolean

Availability: Read or write at run time only

The **showCurrentRowImage** property of the **JadeTableSheet** class specifies whether a binary image is displayed in the fixed cell at the start of the currently selected row, as shown in the following image.



If the value of the **showCurrentRowImage** property is **false** (the default) or the sheet has no fixed columns, there is no impact. If the value is **true**, the first fixed cell displays an image indicating that it is the current row (unless that cell already has a picture assigned) for the current non-fixed row. The image is displayed after any cell text and takes precedence over the text. A default image is displayed unless a different image is set for the **currentRowImage** property.

The image is displayed after any cell text as if the **stretch** property of the **Table** control were set to **Stretch_None_Picture_First** (2) and the cell text is displayed as if the value of the **partialTextIndication** property is **true**. The effective **alignment** property of the cell is still used to draw the image.

Note Row heights and the width of the first column could be affected by this image. The spacing between the image and the text is 3 through 15 pixels, depending on the extra space available.

The code fragment in the following example shows the use of the **showCurrentRowImage** property.

```
table1.accessSheet(1).showCurrentRowImage := true;
```

showPartialTextBubbleHelp

Type: Boolean

Availability: Read or write at run time only

The **showPartialTextBubbleHelp** property of the **JadeTableSheet** class specifies whether a bubble help window displaying the full text is shown when the user moves the cursor over a table cell for which the text is not fully visible.

When the value of this property is **true** (the default value), the bubble help is displayed when the cursor is over a cell in which the text is not fully visible, with the following exceptions.

- The value of the **Window** class **bubbleHelp** property is not null (**bubbleHelp** text takes precedence)
- The cell is disabled
- The cell displays a hyperlink
- The cell is displaying a cell control
- A mouse button is down

The bubble help is hidden when any of the following conditions applies.

- After a short delay (three seconds plus the text length increment)
- If the user moves the cursor out of the cell boundaries
- If the user clicks the mouse
- If the user presses a key

Note If the user clicks the mouse while over the bubble help, the table cell is clicked as normal.

The code fragment in the following example shows the use of the **showPartialTextBubbleHelp** property.

```
table.accessSheet(1).showPartialTextBubbleHelp := false;
```

tabInitialPosition

Type: Integer

Availability: Read or write at run time only

The **tabInitialPosition** property of the **JadeTableSheet** class can change the cell that becomes current when you tab into a **Table** control; that is, the values of the **row** and **column** properties can change. The values for **tabInitialPosition** and their effects are summarized in the following table.

Table Class Constant	Integer Value	Effect on the Current Cell
TabInitialPosition_None	0	No change. This is the default action.
TabInitialPosition_First	1	Current cell becomes first visible, enabled cell in non-fixed area of the table.
TabInitialPosition_Last	2	Current cell becomes last visible, enabled cell in non-fixed area of the table.
TabInitialPosition_First_Last	3	For a forward tab, current cell is first visible, enabled cell in non-fixed table area. For a back tab, current cell is last visible, enabled cell in non-fixed table area.

If the row or the column changes, the **queryRowColChg** event is called. If that event is successful, the **rowColumnChg** event is called.

tabOffEnds

Type: Boolean

Availability: Read or write at run time only

Pressing the Tab key specified for the table moves focus from one visible, enabled cell to the next (the **autoTab** property of a cell control can result in the same shift in focus). The **tabOffEnds** property of the **JadeTableSheet** class determines what happens when a tab action occurs from the last visible, enabled cell in the sheet.

If the value of the **tabOffEnds** property is **true**, which is the default value, focus moves to the next control in the tab order after the table; that is, focus moves out of the table. If the value of the **tabOffEnds** property is **false**, focus moves to the next non-fixed cell that is visible and enabled, the search for such a cell resuming at the start of the sheet.

For a back tab, if the value of the **tabOffEnds** property is **true** and the current cell is the first visible, enabled cell of the non-fixed area of the table, focus moves to the prior control in the tab order before the table; that is, focus moves out of the table.

topRow

Type: Integer

Availability: Read or write at run time

The **topRow** property of the **JadeTableSheet** class contains the row that is displayed at the top edge of the non-fixed area of the sheet referenced by this object.

For a description of this property, see the **Table** control **topRow** property.

The code fragment in the following example shows the use of the **topRow** property.

```
table1.accessSheet(2).topRow := 2;
```

visible

Type: Boolean

Availability: Read or write at run time only

The **visible** property of the **JadeTableSheet** class enables the visibility of the sheet referenced by this object to be accessed. For a description of this property, see the **Table** control **sheetVisible** property.

The code fragment in the following example shows the use of the **visible** property.

```
table1.accessSheet(2).visible := false;
```

widthPercentStyle

Type: Integer

Availability: Read or write at run time only

The **widthPercentStyle**property of the **JadeTableSheet** class controls how any columns percentages set on the table sheet using the **JadeTableColumn** class **widthPercent** property are interpreted.

The property values can be one of the new **Table** class constants listed in the following table.

Table Class Constant	Value	Description
WidthPercent_Style_ClientWidth	0	The default value, which specifies that if the value of the **JadeTableColumn** class **widthPercent** property is greater than zero (**0**), the width of the column is calculated using the formula (Table.clientWidth * JadeTableColumn.widthPercent) / 100
WidthPercent_Style_NoSetWidths	1	Specifies that if the value of the **JadeTableColumn** class **widthPercent** property is greater than zero (0), the width of the column is calculated using the formula ((Table.clientWidth - <set widths>) * JadeTableColumn.widthPercent) / 100

In this table, the **<set widths>** value is the sum of all column widths that have been specifically set by user logic or by the user resizing the column, which means that if the **widthPercent** property values of the other columns add up to 100 percent, those columns fully use the remaining horizontal space in the table.

If the value of **<set widths>** is greater than the value of the **Table** class **clientWidth** property, the width is calculated as if the value of the **widthPercentStyle** property is zero (**0**).

The code fragment in the following example shows the use of the **widthPercentStyle** property.

```
table1.accessSheet(table1.sheet).widthPercentStyle := WidthPercent_Style_
NoSetWidths;
```

JadeTableSheet Methods

The methods defined in the **JadeTableSheet** class are summarized in the following table.

Method	Description
accessCell	Returns a reference to the requested cell
accessColumn	Returns a reference to the requested column
accessRow	Returns a reference to the requested row
addItem	Adds a new row to the sheet
addItemAt	Adds a new row to the sheet at a specified position
clear	Clears the contents of the sheet
delete	Deletes the entire sheet if it is not the only sheet on the table
findColumnObject	Searches the column **itemObjects** for an object, if it exists
findObject	Returns the column and row containing the specified object, if it exists in the current sheet
findRowObject	Searches the row **itemObjects** for an object, if it exists
findString	Searches the cells in a sheet of a table for the specified string
getCellFromPosition	Returns the cell at the specified position, and the row and column of that cell
getCollection	Returns the collection attached to the associated sheet of the table
insertColumn	Enables a single column to be inserted into the sheet

Method	Description
moveColumn	Moves a column of the sheet
moveRow	Moves a row of the sheet
positionCollection	Positions the collection attached to the **Table** control to an object in that collection and to a row within the table
refreshEntries	Refreshes the displayed list of entries on the current sheet of the table
removeItem	Removes a row from the sheet at run time
resort	Resorts the contents of the sheet
restoreAutoSize	Recalculates row heights and column widths, ignoring changes made by logic or user resize
selectedCount	Returns the number of cells with the selected status set
selectedNext	Returns the next selected cell
setCollectionObject	Sets the object in the collection attached to the current sheet of the table

accessCell

Signature `accessCell(row: Integer; column: Integer): JadeTableCell updating;`

The **accessCell** method of the **JadeTableSheet** class returns a reference to the **JadeTableCell** object for the requested **row** and **column** on that sheet.

This method also sets the **Table** class **accessedCell** property to the returned cell, allowing subsequent reuse of that object.

Storing a reference to a returned cell causes problems unless you take a copy of that column, as there is one **JadeTableCell** object for each **Table** control only.

Note Your logic must delete cloned cells.

The code fragment in the following example shows the use of the **accessCell** method.

```
table1.accessSheet(2).accessCell(4, 5).text := "test";
```

See also the **JadeTableSheet** class **accessColumn** and **accessRow** methods and the **Table** class **accessCell**, **accessColumn**, **accessRow**, and **accessSheet** methods.

accessColumn

Signature `accessColumn(column: Integer): JadeTableColumn updating;`

The **accessColumn** method of the **JadeTableSheet** class returns a reference to the **JadeTableColumn** object for the requested **column** on that sheet. This method also sets the **Table** class **accessedColumn** property to the returned column, allowing subsequent reuse of that object.

Storing a reference to a returned column causes problems unless you take a copy of that column, as there is one **JadeTableColumn** object for each **Table** control only.

Note Your logic must delete cloned columns.

The code fragment in the following example shows the use of the **accessColumn** method.

```
while index <=10 do
    col := table1.accessColumn(index);
    if index = 6 then
        col.sortOrder       := 1;
        col.sortType        := SortType_Time;
        table1.sortAsc[1]   := true;
        table1.sortCased[1] := true;
    endif;
endwhile;
```

See also the **JadeTableSheet** class **accessCell** and **accessRow** methods and the **Table** class **accessCell**, **accessColumn**, **accessRow**, and **accessSheet** methods.

accessRow

Signature `accessRow(row: Integer): JadeTableRow updating;`

The **accessRow** method of the **JadeTableSheet** class returns a reference to the **JadeTableRow** object for the requested **row** on that sheet. This method also sets the corresponding **Table** class **accessRow** property to the returned row, allowing subsequent reuse of that object.

Storing a reference to a returned row causes problems unless you take a copy of that row, as there is one **JadeTableRow** object for each **Table** control only.

Note Your logic must delete cloned rows.

See also the **JadeTableSheet** class **accessCell** and **accessColumn** methods and the **Table** class **accessCell**, **accessColumn**, **accessRow**, and **accessSheet** methods.

addItem

Signature `addItem(str: String): Integer;`

The **addItem** method of the **JadeTableSheet** class adds a new row to the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **addItem** method.

The following example shows the use of the **addItem** method.

```
// add a new row that has two columns for company name and address line 1
table1.accessSheet(2).addItem(coy.name & Tab & coy.address1);
```

addItemAt

Signature
```
addItemAt(str:   String;
          index: Integer);
```

The **addItemAt** method of the **JadeTableSheet** class adds a new row to the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **addItemAt** method.

The code fragment in the following example shows the use of the **addItemAt** method.

```
table1.accessSheet(2).addItemAt(coy.name & Tab & coy.address1, 2);
```

clear

Signature `clear();`

The **clear** method of the **JadeTableSheet** class clears the contents of the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **clear** method.

The code fragment in the following example shows the use of the **addItem** method.

```
table1.accessSheet(2).clear;
```

delete

Signature `delete() updating;`

The **delete** method of the **JadeTableSheet** class deletes the entire sheet referenced by this object if the **Table** control contains two or more sheets.

A new **JadeTableSheet** object is created during the next call to the **accessSheet** method of the **Table** class.

If the current sheet is the only sheet in the table, this method does nothing.

findColumnObject

Signature
```
findColumnObject(object: Object;
                 column: Integer io): Boolean;
```

The **findColumnObject** method of the **JadeTableSheet** class searches the item object values for each column (set by the **JadeTableElement** class **itemObject** property) on the sheet for the value specified in the **object** parameter, starting from the column specified in the **column** parameter.

A zero (**0**) value in the **column** parameter is treated as **1**; that is, if this parameter is not specified, the search starts at the first column of the current sheet.

If the specified object is found, this method returns **true** and the value of the column that contains the specified object. If the specified object is not found, this method returns **false** and a zero (**0**) column value.

The code fragment in the following example shows the use of the **findColumnObject** method.

```
int  := 0;
bool := table1.accessSheet(2).findColumnObject(obj, int);
```

findObject

Signature
```
findObject(object: Object;
           column: Integer io;
           row:    Integer io): Boolean;
```

The **findObject** method of the **JadeTableSheet** class searches every cell of the sheet for the value specified in the **object** parameter, starting from the cell specified by the **column** and **row** parameters.

The column of each row is searched, and zero (**0**) values in the **column** and **row** parameters are treated as **1**; that is, if these parameters are not specified, the search starts at the first column of the first row of the current sheet.

If the specified object is found, this method returns **true** and the values of the column and row that contain the specified object. If the specified object is not found, this method returns **false** and a zero (**0**) column and row value.

The code fragment in the following example shows the use of the **findObject** method.

```
row := 0;
col := 0;
if table1.accessSheet(2).findObject(obj, row, col) then
    delete obj;
    table1.accessSheet.removeItem(row);
endif;
```

findRowObject

Signature
```
findRowObject(object: Object;
              row:    Integer io): Boolean;
```

The **findRowObject** method of the **JadeTableSheet** class searches the item object values for each row (set by the **JadeTableElement** class **itemObject** property) on the sheet for the value specified in the **object** parameter, starting from the row specified in the **row** parameter.

A zero (**0**) value in the **row** parameter is treated as **1**; that is, if this parameter is not specified, the search starts at the first row of the current sheet.

If the specified object is found, this method returns **true** and the value of the row that contains the specified object. If the specified object is not found, this method returns **false** and a zero (**0**) row value. The code fragment in the following example shows the use of the **findRowObject** method.

```
row := 0;
if table1.accessSheet(2).findRowObject(obj, row) then
    delete obj;
    table1.accessSheet.removeItem(row);
endif;
```

findString

Signature
```
findString(str:           String;
           row:           Integer io;
           column:        Integer io;
           caseSensitive: Boolean;
           exact:         Boolean): Boolean;
```

The **findString** method of the **JadeTableSheet** class searches the cells in a sheet of a **Table** control for the string specified in the **str** parameter. The search starts at the cell row and column specified by the respective **row** and **column** parameter values or at the first row and column if the parameter values are less than **1**.

The case-sensitivity of the search is determined by the value of the **caseSensitive** parameter.

If the **exact** parameter value is **false**, the search matches any cell text with the specified prefix. If the **exact** parameter value is **true**, the search matches only cells whose text is an exact match.

If the search does not find a cell containing the string with the required **caseSensitive** and **exact** parameter values, this method returns **false**. If the search finds a cell containing the string with the required case-sensitive and exact options, this method returns **true** and the row and column of the located cell are returned; for example:

```
col := 0;
row := 0;
while table1.accessSheet(1).findString("city", row, col, false, false) do
    ....
    col := col + 1; // repeat the search starting in the next column
endwhile;
```

getCollection

Signature `getCollection(): Collection;`

The **getCollection** method of the **JadeTableSheet** class returns a reference to the collection attached to the associated sheet of the **Table** control by the **listCollection** or **displayCollection** method; for example:

```
coll := table1.accessSheet(1).getCollection;
```

If no collection is attached to the sheet, null is returned.

getCellFromPosition

Signature
```
getCellFromPosition(x:      Real;
                    y:      Real;
                    row:    Integer output;
                    column: Integer output): Boolean;
```

The **getCellFromPosition** method of the **JadeTableSheet** class returns a reference to the cell at the position specified by the horizontal and vertical **x** and **y** parameters of the sheet referenced by this object (in units specified by the value of the **Table** class **scaleMode** property).

This method returns **true** if the **row** and **column** parameter values of the cell have been returned, or **false** if the **x** and **y** parameters do not correspond to a cell position on the current sheet.

insertColumn

Signature `insertColumn(at: Integer);`

The **insertColumn** method of the **JadeTableSheet** class enables a single column to be inserted into the **Table** sheet referenced by this object.

The column is empty and assumes the default column width.

The existing columns are shifted to the right of the column specified in the **at** parameter and remain untouched.

For a description of this method, see the **Table** control **insertColumn** method.

The code fragment in the following example shows the use of the **insertColumn** method.

```
table1.accessSheet(2).insertColumn(6);
```

moveColumn

Signature
```
moveColumn(src: Integer;
           dst: Integer);
```

The **moveColumn** method of the **JadeTableSheet** class can be used to move a column of the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **moveColumn** method.

The following example of the **moveColumn** method moves column **4** to column **2**. Column **2** becomes column **3**, and column **3** becomes column **4**.

```
table1.accessSheet(2).moveColumn(4, 2);
```

The following example of the **moveColumn** method moves column **2** to column **4**. Column **3** becomes column **2**, and column **4** becomes column **3**.

```
table1.accessSheet(2).moveColumn(2, 4);
```

moveRow

Signature `moveRow(src: Integer; dst: Integer);`

The **moveRow** method of the **JadeTableSheet** class can be used to move a row of the **Table** sheet referenced by this object.

The following example of the **moveRow** method moves row **4** to row **2**. Row **2** becomes row **3**, and row **3** becomes row **4**.

```
table1.accessSheet(2).moveRow(4, 2);
```

The following example of the **moveRow** method moves row **2** to row **4**. Row **3** becomes row **2**, and row **4** becomes row **3**.

```
table1.accessSheet(2).moveRow(2, 4);
```

The current row is adjusted if that row is affected.

positionCollection

Signature `positionCollection(obj: Object; row: Integer) updating;`

The **positionCollection** method of the **JadeTableSheet** class positions the collection attached to the **Table** control to an object in that collection and to a row within the table. Use the **obj** parameter to specify the object to be positioned and the **row** parameter to specify the visible row in which to position that object.

You can use this method to scroll through an existing collection display by specifying the new position of an object within the current display. For example, the following code fragment scrolls the current collection view so that the second item is positioned in the top row (if the value of the **fixedRows** property is **0**).

```
table1.accessSheet(2).positionCollection(table1.itemObject[2], 1);
```

When using the **positionCollection** method:

- The specified row may not be the resulting displayed row if the required table cannot display sufficient entries to fill the **Table** control.
- The **row** property is set to the row of the object.
- If the specified object is not a visible member of the collection in the table, the display starts from the first visible collection entry.
- If the specified row is:
  - Less than **fixedRows** + 1, **fixedRows** + 1 is assumed.
  - Greater than the number of rows in a table, the number of visible rows is assumed.

refreshEntries

Signature `refreshEntries(obj: Object) updating;`

The **refreshEntries** method of the **JadeTableSheet** class refreshes the list of entries on the current sheet of the table when a collection is attached to the table. For a description of this method, see the **Table** control **refreshEntries** method.

removeItem

Signature `removeItem(index: Integer);`

The **removeItem** method of the **JadeTableSheet** class removes a row from the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **removeItem** method.

The code fragment in the following example shows the use of the **removeItem** method.

```
table1.accessSheet(2).removeItem(2);
```

resort

Signature `resort();`

The **resort** method of the **JadeTableSheet** class resorts the contents of the **Table** sheet referenced by this object. For a description of this method, see the **Table** control **resort** method.

The code fragments in the following examples show the use of the **resort** method.

```
table1.accessSheet(1).accessColumn(table1.column).sortOrder := 1;

table1.accessedSheet.resort;
```

restoreAutoSize

Signature `restoreAutoSize();`

The **restoreAutoSize** method of the **JadeTableSheet** class results in all row and column heights and widths being recalculated, ignoring any column widths or row heights set by logic or by user resize.

selectedCount

Signature `selectedCount(): Integer;`

The **selectedCount** method of the **JadeTableSheet** class returns the number of selected cells in the **Table** sheet referenced by this object.

For a description of this method, see the **Table** control **selectedCount** method.

The code fragment in the following example shows the use of the **selectedCount** method.

```
if table1.accessSheet(2).selectedCount > 0 then
```

selectedNext

Signature
```
selectedNext(r: Integer io;
             c: Integer io): Boolean;
```

The **selectedNext** method of the **JadeTableSheet** class returns the next selected cell following the row and column specified in the **r** and **c** parameters for the **Table** sheet referenced by this object.

For a description of this method, see the **Table** control **selectedNext** method.

The following example steps through all of the selected cells of the current sheet of a table.

```
vars
    row      : Integer;
```

```
    col      : Integer;
    tblSheet : JadeTableSheet;
begin
    tblSheet := table1.accessSheet(2);
    while tblSheet.selectedNext(row, col) do
        ...
    endwhile;
end;
```

setCollectionObject

Signature `setCollectionObject(obj: Object) updating;`

The **setCollectionObject** method of the **JadeTableSheet** class refreshes the list of entries on the current sheet of the table when a collection is attached to the current sheet. This ensures that the object referenced is in the displayed list of collection entries for the table sheet; for example:

```
table1.accessSheet(1).setCollectionObject(obj);
```

For a description of this method, see the **Table** control **setCollectionObject** method.

JadeTcpIpProxy Class

The transient **JadeTcpIpProxy** class implements TCP/IP network proxy support that enables you to open a TCP/IP network connection through a proxy server.

Note Asynchronous connection operations are executed on another thread. If this asynchronous worker thread needs to access JADE objects (for example, **TcpIpConnection** and **JadeTcpIpProxy** objects), these objects need to be shared transient or persistent objects.

If you cannot establish a direct TCP/IP connection because of physical network layouts or restrictions (for example, the use of a firewall), you may have to establish a connection through a proxy server by using the functionality provided by the **JadeTcpIpProxy** class.

You can use proxies as part of a firewall solution, as they sit between the client application and the server application, and may not permit the client to connect directly to the server. The client is required to connect to the proxy and asks the proxy to connect to the server on behalf of the client. The proxy may also require authentication from the client before it allows the connection to the server.

There are a number of different types of proxies, the two major types being HyperText Transfer Protocol (HTTP) and SOCKS. From the perspective of the client, the difference between the types of proxy is the protocol (that is, the type and format of messages) used between the client and the proxy. The other issue for the client is determining the type of proxy and where it is running.

The **TcpIpConnection** class **networkProxy** property contains a reference to a **JadeTcpIpProxy** object. If this reference contains a non-null value, the **JadeTcpIpProxy** class **connect** method is executed instead of the **TcpIpConnection** class **open** or **openAsynch** method for each attempt to connect to the proxy.

If the **networkProxy** property value is **null**, the **TcpIpConnection** class **open** or **openAsynch** method is executed.

For details about the **JadeTcpIpProxy** class constants and the properties and method defined in the **JadeTcpIpProxy** class, see "JadeTcpIpProxy Class Constants", "JadeTcpIpProxy Properties", and "JadeTcpIpProxy Method", in the following subsections.

For details about reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples" and "Considerations when Implementing JadeTcpIpProxy Class Features", in the following subsections. (See also "Firewall for the JADE Internet Environment", in Chapter 2 of the *JADE Installation and Configuration Guide*.)

Inherits From: Object

Inherited By: (None)

Proxy Communication Code Examples

In the following example of a method that defines values for **JadeTcpIpProxy** class properties, note the following points that are referred to in comments within the method.

1. Setting the **browserType** property controls how the proxy object behaves. To indicate that the proxy object should not go looking for any configuration information and that all required details are available as property values on the proxy object, set the browser type to **BrowserType_None** (0).

2. For the location and type of the proxy server in this example, the proxy server is running on the **proxyhost.testing.com**, and it is listening for connections on port **8088**. In addition, the proxy server is an HTTP-based server so the **proxyType** property is set to **ProxyType_Http** (1).

3. The proxy server requires authentication. If the proxy server supports the Windows Challenge/Response (NTLM) authentication protocol on a Windows PC logged into the domain, the proxy object uses the PC log-in details. If these details fail or if NTLM is not supported, the values in the **userName** and **password** properties are used for authentication.

```
vars
    tcpip : TcpIpConnection;
    proxy : JadeTcpIpProxy;
begin
    // Create and setup the proxy object
    create proxy transient;
    // Set the properties we need on the proxy
    // [1] We want total control
    proxy.browserType := JadeTcpIpProxy.BrowserType_None;
    // [2] We know the location and type of the proxy server.
    proxy.host := "proxyhost.testing.com";
    proxy.port := 8088;
    proxy.proxyType := JadeTcpIpProxy.ProxyType_Http;
    // We know it is an HTTP proxy
    // [3] Authentication details are required
    proxy.userName := "Dr. Who";
    proxy.password := "tardis";
    // Set up the TCP/IP-based connection
    create tcpip transient;
    // Normal TCP/IP connection details
    tcpip.name := "server.internet.com";
    tcpip.port := 5432;
    // Add a reference to the proxy object from the TCP/IP object so
    // that the connection is attempted through a proxy server
    tcpip.networkProxy := proxy;
    // Now perform standard TCP/IP logic
    tcpip.open;
    . . .               // do some processing here
```

In the following example of a method that shows JADE locating and using proxy values for the appropriate type of browser, note the following points that are referred to in comments within the method.

1. Setting the **browserType** property controls how the proxy object behaves. Set the browser type to **BrowserType_InternetExplorer** (1) if the proxy server details have been configured into Internet Explorer or to **BrowserType_Netscape** (2) if they have been configured into a Mozilla-style Web browser.

2. As the proxy object needs to know the location and type of the proxy server in this example, we assume that all of the necessary details can be obtained automatically, as follows.

   - If the browser type is **BrowserType_InternetExplorer** (1), the registry is checked.
   - If the browser type is **BrowserType_Netscape** (2), JADE checks the **MozillaPrefs** parameter in the [JadeClient] section of the JADE initialization file for the name of a valid Mozilla-style Web browser user preferences file (which is usually called **prefs.js**).

3. The proxy server requires authentication. In the following example, we assume that the proxy server does not require authentication or that the proxy object can obtain the necessary information from the operating system and pass this behind the scenes to the proxy server without involving us.

```
vars
    tcpip : TcpIpConnection;
    proxy : JadeTcpIpProxy;
```

```
begin
    // Create and set up the proxy object
    create proxy transient;
    // Set the properties we need on the proxy
    // [1] We are running on a Windows platform
    proxy.browserType := JadeTcpIpProxy.BrowserType_InternetExplorer;
    // [2] Assume that the location and type of proxy server can
    // be discovered from the registry.
    // [3] Assume that the proxy server does not require authentication
    // or that we are authenticated as part of log on to our PC.
    // Setup the TCP/IP-based connection.
    create tcpip transient;
    // Normal TCP/IP connection details
    tcpip.name := "server.internet.com";
    tcpip.port := 5432;
    // Add a reference to the proxy object from the TCP/IP object so
    // that the connection is attempted through a proxy server.
    tcpip.networkProxy := proxy;
    // Now perform standard TCP/IP logic
    tcpip.open;
    . . .            // do some processing here
```

Considerations when Implementing JadeTcpIpProxy Class Features

When implementing features of the **JadeTcpIpProxy** class, consider the following issues.

- For HTTP proxy servers, NTLM and Basic authentication modes only are supported (that is, message digest is not supported).

  The degree of impact depends on the type of proxy server. Microsoft Proxy Server supports message digest authentication.

- HTTP-based proxy servers that implement redirection are not supported. You are more likely to use redirection if you are a large site or you have multiple proxies that are geographically distributed.

- The Internet Explorer (version 5.0 or higher) ability to automatically discover the location of proxy server configuration information is not implemented in any form.

- The Internet Explorer ability to check a destination server address to see if it is a "local address" that should optionally be excluded from using a proxy server is not supported. This may affect you if you are connecting locally to a server but you require a proxy to access sites on the Internet.

- The Web browser feature that excludes specific hosts, domains, or IP address ranges is not supported. This may affect you if you are connecting locally to a server but you require a proxy to access sites on the Internet.

- There is no ability to select between multiple available proxy servers based on the higher-level protocol that is being used or implemented by the **TcpIpConnection** object. The proxy server to use for File Transfer Protocol (FTP) transfers can be different from the proxy server for HTTP, HTTPS, or Gopher.

- You cannot control the authentication method that the proxy object uses if the proxy server supports multiple authentication methods. The proxy server attempts authentication using the methods listed by the proxy server in the specified order, if it understands that authentication method.

- If the user name and password combination fails, the entire connection process fails. This requires the **userName** and **password** properties to be updated before the connection is retried.

If both NTLM and Basic authentication are supported and the user is not allowed to connect through an NTLM-based authentication, JADE still tries NTLM authentication each time before it retries with Basic authentication, which could lead to a possible account lockout.

- The **SOCKS_SERVER** and **/etc/socks.conf** files are not supported.

JadeTcpIpProxy Class Constants

The constants provided by the **JadeTcpIpProxy** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
BrowserType_Auto	3	ProxyType_Auto	0
BrowserType_InternetExplorer	1	ProxyType_Direct	5
BrowserType_Netscape	2	ProxyType_Http	1
BrowserType_None	0	ProxyType_Https	4
ProxyType_Socks4	2		

JadeTcpIpProxy Properties

The properties defined in the **JadeTcpIpProxy** class are summarized in the following table.

Property	Contains the…
browserType	Browser type whose proxy host configuration settings are used
domain	Domain name to log in to the host
host	Name or IP address of the host
password	Password that is to complete the log in to the host
port	Port number used to connect to the host
proxyType	Communications protocol used to connect to the proxy host
userName	User name that logs in to the host

browserType

Type: Integer

The **browserType** property of the **JadeTcpIpProxy** class contains the type of Web browser whose configuration settings are used for the proxy connection and controls how the proxy object behaves.

If this property contains a non-zero value, JADE attempts to read proxy host configuration settings from that browser.

The **browserType** property can be set to one of the values listed in the following table.

JadeTcpIpProxy Class Constant	Integer Value
BrowserType_None	0 (the default)
BrowserType_InternetExplorer	1

JadeTcpIpProxy Class Constant	Integer Value
BrowserType_Netscape	2
BrowserType_Auto	3

Use the **BrowserType_InternetExplorer** (1) value if your proxy server details have been configured into Internet Explorer or the **BrowserType_Netscape** (2) value if your proxy server details have been configured into a Mozilla-style Web browser.

If the browser type is **BrowserType_InternetExplorer** (1), the registry is checked. If the browser type is **BrowserType_Netscape** (2), JADE checks the **MozillaPrefs** parameter in the [JadeClient] section of the JADE initialization file for the name of a valid Mozilla-style Web browser user preferences file (which is usually called **prefs.js**). If you want your application to have its proxy settings set externally from your JADE code, use the **BrowserType_Auto** (3) value.

BrowserType_InternetExplorer (1) is used for client nodes.

The default value of **BrowserType_None** (0) indicates that the **JadeTcpIpProxy** object does not have to look for configuration information and that all required details are available as properties on the proxy object.

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

domain

Type: String[255]

The **domain** property of the **JadeTcpIpProxy** class contains the name that identifies the secure proxy server controller through which a connection is made to the host server. Set this property if you require authentication to connect through a Windows secure proxy server to the host (for example, **mydomain**).

If your Windows proxy server controller requires authentication, the **userName** property can contain both the domain name and user name, separated by a backslash character (for example, **mydomain\myloginname**).

host

Type: String[255]

The **host** property of the **JadeTcpIpProxy** class contains the network name or Internet Protocol (IP) address of the proxy server controller through which a connection is made to the host server.

If your application is behind a firewall and your network administrator requires connections to the Internet to be done through a proxy server, this property and the **port** property identify the proxy server controller through which connections are made to the host server. For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

password

Type: String[255]

The **password** property of the **JadeTcpIpProxy** class contains the text that is used for log-in authentication on the proxy server in conjunction with **userName** property, if required, to enable you to communicate through the proxy server to the host server. Set this property if you require authentication to connect to the proxy server.

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

port

Type: Integer[4]

The **port** property of the **JadeTcpIpProxy** class contains the port number of the proxy server controller through which a connection is made to the host server. If your application is behind a firewall and your network administrator requires connections to the Internet to be done through a proxy server, this property and the **host** property identify the proxy server controller through which connections are made to the host server.

Note If the value of the **proxyType** property is set to **ProxyType_Auto** (0) and the value of the **port** property is zero (**0**), the default port number is used for each attempted protocol. The default port number is **80** for HyperText Transfer Protocol (http), **3128** for HyperText Transfer Protocol secure (https), and **1080** for SOCKS V4.

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

proxyType

Type: Integer[4]

The **proxyType** property of the **JadeTcpIpProxy** class contains the proxy server communications protocol through which client nodes connect to the host server.

Note Only the HyperText Transfer Protocol (HTTP) proxy type and the connect part of the SOCKS V4 protocol is implemented (that is, SOCKS V4 binding is not implemented).

The constants provided by the **JadeTcpIpProxy** class for the **proxyType** property are listed in the following table.

Class Constant	Integer Value	Comment
ProxyType_Auto	0	The default value.
ProxyType_Direct	5	Protocol allowing a reference from a **TcpIpConnection** object to a **JadeTcpIpProxy** object to be defined, but the network connection will not attempt to connect via a proxy server. The behavior is equivalent to having the **TcpIpConnection** class **networkProxy** property set to **null**.
ProxyType_Http	1	Protocol allowing redirection based on domain that is currently supported.
ProxyType_Https	4	Attempts to connect to the destination host via a proxy that supports the HTTP **CONNECT** protocol.
ProxyType_Socks4	2	Connect part only is implemented in this release.

When the **proxyType** property is set to **ProxyType_Auto** (the default), you can retrieve network proxy settings automatically, by using the parameters in the [JadeClient] section of the JADE initialization file listed in the following table. (For details, see "JADE Object Manager Client Module Section [JadeClient]", in the *JADE Initialization File Reference*.)

Parameter	Specifies…
ProxyAutoconfigUrl	The name of URL used for automatic configuration of the client node, used only when the **ProxySettingsLocation** parameter is set to **ini**.
ProxySettingsLocation	Where proxy configuration details are defined. The settings location can be one of the following values. ■ Undefined (defaults to **registry**) ■ **auto** (retrieves network proxy settings automatically) ■ **direct** or **none** (direct network connection that does not attempt to connect via a proxy server and is equivalent to having the **TcpIpConnection** class **networkProxy** property set to **null**) ■ **environment** (looks for the **http_proxy** environment variable, to obtain proxy host and port numbers) ■ **ini** (read further settings from the **ProxyAutoconfigUrl**, **ProxyHost**, **ProxyPort**, and **ProxyType** parameters) ■ **mozilla** or **netscape** (read values from the file specified in the **MozillaPrefs** parameter in the [JadeClient] section of the JADE initialization file) ■ **registry** (read settings from the Windows registry)
ProxyHost	The host name or IP address number of the proxy server controller through which a connection is made to the host server, used only when the **ProxySettingsLocation** parameter is set to **ini**.
ProxyPort	The valid port number of the proxy server controller through which a connection is made to the host server, used only when the **ProxySettingsLocation** parameter is set to **ini**.
ProxyType	The proxy server communications protocol through which client nodes connect to the host server, used only when the **ProxySettingsLocation** parameter is set to **ini**.

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

userName

Type: String[255]

The **userName** property of the **JadeTcpIpProxy** class contains the user name that is used for log-in authentication on the proxy server in conjunction with the **password** property, if required, to enable you to communicate through the proxy server to the host server (for example, **myloginname**).

If your proxy server controller requires authentication, the **userName** property can contain both the domain name (optionally specified by using the **domain** property) and user name, separated by a backslash character (for example, **mydomain\myloginname**).

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

JadeTcpIpProxy Method

The method defined in the **JadeTcpIpProxy** class is summarized in the following table.

Method	Description
connect	Establishes a connection to the target host through the specified network proxy

connect

Signature `connect(tcpipConnection: TcpIpConnection);`

The **connect** method defined in the **JadeTcpIpProxy** class connects to the target host through the specified proxy server. When a **TcpIpConnection** class **open** or **openAsynch** method is called, JADE checks to see if the **TcpIpConnection** class **networkProxy** property contains a reference to a **JadeTcpIpProxy** object.

If a connection to the host server through a proxy server is required (that is, the **networkProxy** property contains a **JadeTcpIpProxy** reference), the **TcpIpConnection** object is passed to the **connect** method of the **JadeTcpIpProxy** object.

The **connect** method establishes a connection to a remote application through a proxy server and returns when the connection is established. An exception is raised if an object reference mismatch is detected between the proxy server and the proxy specified in the **tcpipConnection** parameter or if the attempt to establish a connection fails.

You can reimplement the **connect** method if you have special proxy requirements. The reimplementing method must use the **TcpIpConnection** object specified in the **tcpipConnection** parameter to perform the necessary TCP/IP communications with the proxy server.

For examples of reimplementing **JadeTcpIpProxy** class functionality, see "Proxy Communication Code Examples", earlier in this chapter. See also "Considerations when Implementing JadeTcpIpProxy Class Features", earlier in this chapter.

JadeTestCase Class

The **JadeTestCase** class provides unit testing functionality for user-written test subclasses.

For details about the methods defined in the **JadeTestCase** class, see "JadeTestCase Methods", in the following subsection.

For details about using the JADE unit testing framework to provide unit testing services for your own applications, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeTestCase Methods

The methods defined in the **JadeTestCase** class are summarized in the following table.

Method	Description
assert	Invoked by a user test method if the method fails.
assertEquals	Asserts that an actual test result is equal to the expected result. If this is not the case, the test fails. A failure message is generated.
assertEqualsMsg	Asserts that an actual test result is equal to the expected result. If this is not the case, the test fails. The failure message passed to the method is used.
assertFalse	Asserts that a condition is **false**. If this is not the case, the test fails. A failure message is generated.
assertFalseMsg	Asserts that a condition is **false**. If this is not the case, the test fails. The failure message passed to the method is used.
assertNotNull	Asserts that an object exists. If this is not the case, the test fails. A failure message is generated.
assertNotNullMsg	Asserts that an object exists. If this is not the case, the test fails. The failure message passed to the method is used.
assertNull	Asserts that an object does not exist. If this is not the case, the test fails. A failure message is generated.
assertNullMsg	Asserts that an object does not exist. If this is not the case, the test fails. The failure message passed to the method is used.
assertTrue	Asserts that a condition is **true**. If this is not the case, the test fails. A failure message is generated.
assertTrueMsg	Asserts that a condition is **true**. If this is not the case, the test fails. The failure message passed to the method is used.
expectedException	Registers expected exceptions before a test method is executed.
info	Outputs the specified message but does not cause the test to fail.

assert

Signature `assert(message: String);`

The **assert** method of the **JadeTestCase** class is invoked by a user test method if the method fails. It can be invoked directly by the user test method, or indirectly by one of the specific *assert* methods in the **JadeTestCase** class (for example, the **assertEquals** method).

A message describing the failure is passed to the method as the value of the **message** parameter.

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertEquals

Signature
```
assertEquals(expected: Any;
             actual:   Any);
```

The **assertEquals** method of the **JadeTestCase** class is invoked by a user test method to compare the result of the test, represented by the value of the **actual** parameter, with the expected result represented by the value of the **expected** parameter.

If the test fails, indicated by the values of the **expected** and **actual** parameters being different, a message is generated in the following format.

```
assertEquals - expected m but actual = n
```

The following code example shows the use of the **assertEquals** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertEquals(2, calculator.getResult());
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertEqualsMsg

Signature
```
assertEqualsMsg(message:  String;
                expected: Any;
                actual:   Any);
```

The **assertEqualsMsg** method of the **JadeTestCase** class is invoked by a user test method to compare the result of the test, represented by the value of the **actual** parameter, with the expected result represented by the value of the **expected** parameter.

If the test fails, indicated by the values of the **expected** and **actual** parameters being different, a message is generated in the following format.

```
assertEqualsMsg - expected m but actual = n
```

The following code example shows the use of the **assertEqualsMsg** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertEqualsMsg("Addition error", 2, calculator.getResult());
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertFalse

Signature `assertFalse(condition: Boolean);`

The **assertFalse** method of the **JadeTestCase** class is invoked by a user test method to evaluate the result of the test, represented by the value of the **condition** parameter.

If the test fails, indicated by the **condition** parameter evaluating to **true**, the following message is generated to describe the failure.

```
assertFalse
```

The following code example shows the use of the **assertFalse** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertFalse(calculator.getResult() <> 2);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertFalseMsg

Signature
```
assertFalseMsg(message:   String;
               condition: Boolean);
```

The **assertFalseMsg** method of the **JadeTestCase** class is invoked by a user test method to evaluate the result of the test, represented by the value of the **condition** parameter. A message is provided in the **message** parameter for the case when the test fails, indicated by the **condition** parameter evaluating to **true**.

The following code example shows the use of the **assertFalseMsg** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertFalseMsg("Addition error", calculator.getResult() <> 2);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertNotNull

Signature `assertNotNull(object: Object);`

The **assertNotNull** method of the **JadeTestCase** class is invoked by a user test method to confirm that the object specified by the **object** parameter exists (that is, it is not a **null** reference).

If the test fails, indicated by the **object** parameter being a **null** reference, the following message is generated to describe the failure.

```
assertNotNull - object is null
```

The following code example shows the use of the **assertNotNull** method in a user test method.

```
calculatorSetup() unitTest;
begin
    create calculator transient;
    assertNotNull(calculator);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertNotNullMsg

Signature
```
assertNotNullMsg(message: String;
                 object:  Object);
```

The **assertNotNullMsg** method of the **JadeTestCase** class is invoked by a user test method to confirm that the object specified by the **object** parameter exists (that is, it is not a **null** reference).

A message is provided in the **message** parameter for the case when the test fails, indicated by the **object** parameter being a **null** reference.

The following code example shows the use of the **assertNotNullMsg** method in a user test method.

```
calculatorSetup() unitTest;
begin
    create calculator transient;
    assertNotNullMsg("Calculator missing", calculator);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertNull

Signature `assertNull(object: Object);`

The **assertNull** method of the **JadeTestCase** class is invoked by a user test method to confirm that the object specified by the **object** parameter is a **null** reference.

If the test fails, indicated by the **object** parameter not being a **null** reference, the following message is generated to describe the failure.

```
assertNull - object <class name and oid>
```

The following code example shows the use of the **assertNotNull** method in a user test method.

```
calculatorTeardown() unitTest;
begin
    delete calculator;
    assertNull(calculator);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertNullMsg

Signature `assertNullMsg(message: String;`
`object: Object);`

The **assertNullMsg** method of the **JadeTestCase** class is invoked by a user test method to confirm that the object specified by the **object** parameter is a **null** reference.

A message is provided in the **message** parameter for the case when the test fails, indicated by the **object** parameter being a **null** reference.

The following code example shows the use of the **assertNullMsg** method in a user test method.

```
calculatorTeardown() unitTest;
begin
    delete calculator;
    assertNullMsg("Calculator still present", calculator);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertTrue

Signature `assertTrue(condition: Boolean);`

The **assertTrue** method of the **JadeTestCase** class is invoked by a user test method to evaluate the result of the test, represented by the value of the **condition** parameter. If the test fails, indicated by the **condition** parameter evaluating to **false**, the following message is generated to describe the failure.

```
assertTrue
```

The following code example shows the use of the **assertTrue** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertTrue(calculator.getResult() = 2);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

assertTrueMsg

Signature `assertTrueMsg(message: String; condition: Boolean);`

The **assertTrueMsg** method of the **JadeTestCase** class is invoked by a user test method to evaluate the result of the test, represented by the value of the **condition** parameter.

A message is provided in the **message** parameter for the case when the test fails, indicated by the **condition** parameter evaluating to **false**.

The following code example shows the use of the **assertTrueMsg** method in a user test method.

```
add() unitTest;
begin
    calculator.add(1);
    calculator.add(1);
    assertTrueMsg("Addition error", calculator.getResult() = 2);
end;
```

A failure in the test method results in the **testFailure** method being executed by an object implementing the **JadeTestListenerIF** interface. For details, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

expectedException

Signature `expectedException(params: ParamListType);`

The **expectedException** method of the **JadeTestCase** class is invoked by a user test method to register exceptions that are expected during execution of the test method.

A test method that encounters an exception is considered to have failed, unless the exception is registered as an *expected* exception.

The **params** parameter specifies expected exceptions by the class of the exception or the **errorCode** property.

In the following example, a **1035** (String too long) exception, a **5011** (Record truncated) exception, and any exception of type **FileException** are expected.

```
someTest() unitTest;
begin
    expectedException(1035, FileException, 5011);
    // test instructions omitted
end;
```

You can also register expected exceptions by calling the **expectedException** method a number of times, as shown in the following code fragment.

```
    expectedException(1035);
    expectedException(FileException);
    expectedException(5011);
```

info

Signature `info(message: String);`

The **info** method of the **JadeTestCase** class outputs the message contained in the **message** parameter but does not cause the test to fail. This method enables you to log user information (for example, progress or test descriptions) with error output.

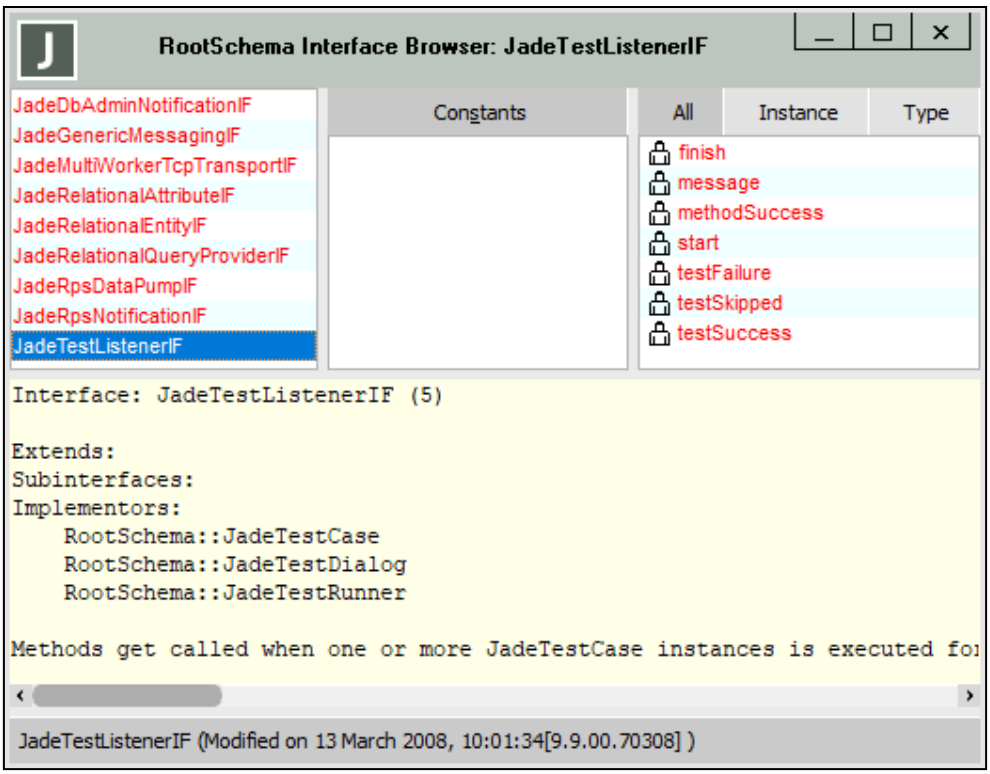
Tip Use the **info** method to provide feedback when debugging a test method.

JadeTestListenerIF Interface

The **JadeTestListenerIF** interface, defined in the **RootSchema**, provides the definition of the event callback methods that you can implement in your user schema classes to display or report on the progress of unit tests run for one or more **JadeTestCase** instances.

For details about using the JADE unit testing framework, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

You can view the **JadeTestListenerIF** interface and its methods only in the Interface Browser of a user schema that has an implementation mapping to this **RootSchema** interface, as shown in the following image.



For details about implementing the **JadeTestListenerIF** interface for a class selected in the Class Browser of a user schema, see "Implementing an Interface", in Chapter 14, "Adding and Maintaining Interfaces", of the *JADE Development Environment User's Guide*.

Notes Automatically generated stub methods in classes that implement the interface contain no body logic.

It is your responsibility to provide the source that meets your application requirements for each stub method.

For details about the **JadeTestListenerIF** interface methods, see "JadeTestListenerIF Interface Method Callback Signatures", in the following subsection.

JadeTestListenerIF Interface Callback Method Signatures

The signatures of callback methods provided the **JadeTestListenerIF** interface are summarized in the following table.

Method	When the callback method is invoked …
finish	After the last test method for the last **JadeTestCase** subclass completes
message	Before the first test method for a **JadeTestCase** subclass starts or after the last test method completes
methodSuccess	If a test method completes successfully without an exception or an assertion failure
start	Before the first test method for the first **JadeTestCase** subclass starts
testFailure	If a test method results in an exception or an assertion failure
testSkipped	If a test method has the **unitTestIgnore** method option and is skipped
testSuccess	For each individual assertion that passes, in each test method run

For details about the method options that are available for methods in a unit test class (for example, the **unitTestIgnore** method option), see "Writing Unit Tests", in Chapter 17 of the *JADE Developer's Reference*. For details about method options, see "Method Options", in Chapter 1 of the *JADE Developer's Reference*.

finish

Signature

```
finish(elapsedTime:    Time;
       testsFailed:    Integer;
       testsSkipped:   Integer;
       testsSucceeded: Integer);
```

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **finish** event occurs just once, at the conclusion of the unit test run. The **finish** callback method is then invoked for the test listener object, if it exists.

The parameters for the **finish** method are listed in the following table.

Parameter	Contains the…
elapsedTime	Time to run all the test methods
testsFailed	Number of test methods that failed
testsSkipped	Number of test methods that were skipped
testsSucceeded	Number of test methods that succeeded

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

message

Signature `message(messageText: String);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **JadeTestListenerIF** class **message** event occurs before the first **JadeTestCase** unit test method is executed and after the last **JadeTestCase** unit test method is executed, for each schema that has tests being run. The **message** callback method is then invoked for the test listener object, if it exists.

The **start** and **end** messages specify the schema or schemas being tested.

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

methodSuccess

Signature `methodSuccess(testMethodName: String);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **methodSuccess** event occurs for each test method from a **JadeTestCase** class that completes successfully without an assertion failing or an exception occurring. The **methodSuccess** callback method is then invoked for the test listener object, if it exists.

The **testMethodName** parameter is the fully qualified name of the successful method in the following format.

```
schema-name::class-name::method-name
```

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

start

Signature `start(numberOfTestMethods: Integer);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **start** event occurs just once, at the beginning of the unit test run. The **start** callback method is then invoked for the test listener object, if it exists.

The **numberOfTestMethods** parameter contains the number of test methods that have the **unitTest** or the **unitTestIgnore** method option.

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

testFailure

Signature `testFailure(testMethodName: String;`
`callStack: String;`
`failureReason: String);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **testFailure** event occurs for each test method from a **JadeTestCase** class that fails because an assertion fails or an exception is raised. The **testFailure** callback method is then invoked for the test listener object, if it exists.

The parameters for the **testFailure** method are listed in the following table.

Parameter	Contains the…
testMethodName	Name of the test method that failed in the *schema-name*::*class-name*::*method-name* fully qualified format .
callStack	Call stack of all user test methods at the point of the assertion failure or the captured exception in the form *schema-name*::*class-name*::*method-name*, followed by the position within the method source in parentheses. The top method in the stack is the one in which the assertion failed or the exception was captured.
failureReason	Reason the test method failed.

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

testSkipped

Signature `testSkipped(testMethodName: String);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed, apart from those methods that have **unitTestIgnore** method parameter, which are skipped.

A typical reason for skipping a test is that the functionality to be tested has not yet been completed.

The **testSkipped** event occurs for each test method from a **JadeTestCase** class that is skipped. The **testSkipped** callback method is then invoked for the test listener object, if it exists.

The **testMethodName** parameter is the fully qualified name of the skipped method in the following format.

```
schema-name::class-name::method-name
```

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

testSuccess

Signature `testSuccess(testMethodName: String);`

When unit tests are run, all of the test methods for each class in a collection of **JadeTestCase** subclasses are executed.

The **testSuccess** event occurs for each individual assertion that succeeds, in each test method run. The **testSuccess** callback method is then invoked for the test listener object, if it exists.

The **testMethodName** parameter is the fully qualified name of the successful method in the following format.

```
schema-name::class-name::method-name
```

For details about running unit tests, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*.

JadeTestRunner Class

The **JadeTestRunner** class enables you to run unit tests that have been defined in subclasses of the **JadeTestCase** class. The class is involved regardless of whether the unit tests are run from the JADE development environment, from an application, or by using a batch process.

For details about using the JADE unit testing framework, see "Using the JADE Testing Framework", in Chapter 17 of the *JADE Developer's Reference*. For details about the methods defined in the **JadeTestRunner** class, see "JadeTestRunner Methods", in the following subsection.

Inherits From: Object

Inherited By: (None)

JadeTestRunner Class Methods

The methods provided by the **JadeTestRunner** class are listed in the following table.

Method	Description
runTests	Executes the specified test methods
setDebugOnAssert	Specifies that the test runner invokes the Process class debug method if an assert fails
setDebugOnException	Specifies that the test runner invokes the **Process** class **debug** method if any exception occurs
setDebugOnUnexpectedException	Specifies that the unit test is paused and the call stack is displayed when the unit test encounters an unexpected exception.
setLogCallStack	Specifies that the test runner reports the call stack when a test method assertion fails or an exception is raised
setTestListener	Specifies the object that will listen for the results of test methods

runTests

Signature `runTests(tests: ObjectArray): Integer;`

The **runTests** method of the **JadeTestRunner** class executes the test methods of all test classes in the collection specified in the **tests** parameter.

The **tests** parameter is a collection of unit test classes and individual unit test methods. For a unit test class included in the collection, all unit test methods are executed.

The test classes must all be subclasses of the **JadeTestCase** class.

The following example shows the use of the **runTests** method to run all unit test methods for the **TestConvertor** class and two unit test methods for the **TestCalculator** class.

```
vars
    tests : ObjectArray;
    jtr : JadeTestRunner;
begin
    create tests transient;
```

```
        tests.add(TestConvertor);
        tests.add(TestCalculator::add);
        tests.add(TestCalculator::divide);
        create jtr transient;
        jtr.runTests(tests);
    epilog
        delete tests;
        delete jtr;
    end;
```

setDebugOnAssert

Signature `setDebugOnAssert(value: Boolean);`

The **setDebugOnAssert** method of the **JadeTestRunner** class specifies that the test runner invokes the **Process** class **debug** method if an assert fails.

setDebugOnException

Signature `setDebugOnException(value: Boolean);`

The **setDebugOnException** method of the **JadeTestRunner** class specifies that the test runner invokes the **Process** class **debug** method if an exception occurs. (See also the **JadeTestCase** class **expectedException** method.)

setDebugOnUnexpectedException

Signature `setDebugOnUnexpectedException(value: Boolean);`

The **setDebugOnUnexpectedException** method of the **JadeTestRunner** class specifies that the test runner pauses and invokes the **Process** class **debug** method to display the call stack when the unit test encounters an unexpected exception so that you can debug exceptions other than any that the test has registered with the **JadeTestCase** class **expectedException** method.

setLogCallStack

Signature `setLogCallStack(value: Boolean);`

The **setLogCallStack** method of the **JadeTestRunner** class specifies that the test runner reports the call stack when a test method assertion fails or an exception is raised.

setTestListener

Signature `setTestListener(listener: JadeTestListenerIF);`

The **setTestListener** method of the **JadeTestRunner** class identifies the object specified by the **listener** parameter that will *listen* to information about the progress and results of test methods.

The **listener** object must implement the methods of the **JadeTestListenerIF** interface, which receive information about the success or failure of test methods as they are run.

There can be, at most, one listener object for a test run. If a listener object is not specified, information about the test run is output to the Jade Interpreter Output Viewer.

The following example shows the use of the **setTestListener** method to specify the test listener object.

```
vars
    tests : ObjectArray;
    file : ListenerFile;
 // ListenerFile is a subclass of File that implements JadeTestListenerIF
    jtr : JadeTestRunner;
begin
    create file transient;
    file.mode := File.Mode_Append;
    file.fileName := "C:\UnitTests\results.txt";
    create tests transient;
    tests.add(TestCalculator);
    create jtr transient;
    jtr.setTestListener(file);
    jtr.runTests(tests);
epilog
    delete tests;
    delete file;
    delete jtr;
end;
```

JadeTransactionTrace Class

The **JadeTransactionTrace** class enables you to identify objects that are updated, created, and deleted within a transaction. For details about transaction tracing, see "Tracing Transactions", in Chapter 19 of the *JADE Developer's Reference*.

For details about the constants, properties, and methods defined in the **JadeTransactionTrace** class, see "JadeTransactionTrace Class Constants", "JadeTransactionTrace Properties", and "JadeTransactionTrace Methods", in the following subsections.

Inherits From: Object

JadeTransactionTrace Class Constants

The constants provided by the **JadeTransactionTrace** class are listed in the following table.

Constant	Value	Description
CallbackMethod	"callbackMethod"	Identify callback methods returned by the **getTransactionTraceCallbacks** method
CallbackReceiver	"callbackReceiver"	Identify callback receivers returned by the **getTransactionTraceCallbacks** method
OperationAutoAdd	10	An object has been automatically added to a collection
OperationAutoProp	8	A property of an object has been set automatically
OperationAutoRemove	12	An object has been automatically removed from a collection
OperationCollAdd	9	An object has been manually added to a collection
OperationCollRemove	11	An object has been manually removed from a collection
OperationCreate	4	An object has been created
OperationDelete	6	An object has been deleted
OperationSetProp	7	A property of an object has been set manually
OperationUpdate	3	An object has been updated
TraceAborted	11	The traced transaction has been aborted
TraceCommitted	10	The traced transaction has been committed
TraceCommitting	9	The traced transaction is about to be committed
TraceStarted	1	Transaction tracing is currently active
TraceStopped	2	Transaction tracing has been stopped
TraceUndefined	0	The tracing is unknown

JadeTransactionTrace Properties

The properties defined in the **JadeTransactionTrace** class are summarized in the following table.

Property	Contains …
myProcess	A reference to the current process instance
startTime	Start time of the current transaction
status	Status of the current transaction
stopTime	End time of the current transaction
tranId	Transaction id of the current transaction

myProcess

Type: Process

The **myProcess** property of the **JadeTransactionTrace** class contains a reference to the current process.

startTime

Type: TimeStamp

The **startTime** property of the **JadeTransactionTrace** class contains the time at which the transaction started.

If a transaction has not started since transaction tracing was initiated, the **startTime** property is null.

status

Type: Integer

The **status** property of the **JadeTransactionTrace** class indicates the current tracing and transaction status.

The **status** property can have one of the class constant values listed in the following table.

Value	Class Constant	Description
0	TraceUndefined	The tracing status is unknown
1	TraceStarted	Transaction tracing is currently active
2	TraceStopped	Transaction tracing has been stopped
9	TraceCommitting	The traced transaction is about to be committed
10	TraceCommitted	The traced transaction has been committed
11	TraceAborted	The traced transaction has been aborted

stopTime

Type: TimeStamp

The **stopTime** property of the **JadeTransactionTrace** class contains the time at which the transaction was stopped, if transaction tracing is stopped within a transaction by using the **stopTransactionTrace** method of the **Process** class. Otherwise, it contains the time at which the most recent transaction was committed or aborted.

tranId

Type: Integer

The **tranId** property of the **JadeTransactionTrace** class contains an **Integer64** value representing the transaction id of the traced transaction.

The property value is the same as that returned by the **getTransactionId** or **getTransactionId64** or method of the **Process** class.

JadeTransactionTrace Methods

The methods defined in the **JadeTransactionTrace** class are summarized in the following table.

Method	Description
clear	Clears all transaction information
getEntry	Returns a specific item in the list
getEntryCount	Returns the number of entries in the list

clear

Signature `clear() updating;`

The **clear** method of the **JadeTransactionTrace** class clears information in the receiver.

getEntry

Signature
```
getEntry(index:     Integer input;
         object:    Object output;
         operation: Integer output;
         prop:      Property output;
         value:     Any output);
```

The **getEntry** method of the **JadeTransactionTrace** class returns information from a specified entry in the list held by the receiver.

The **getEntry** method parameters, representing **JadeTransactionTrace** property values, are listed in the following table.

Parameter	Description
index	Identifies the entry to be accessed. The first entry in the list has an index of one (**1**).
object	References an object that was created, updated, or deleted by the transaction.

Parameter	Description
operation	Specifies the action that was carried out on the object. The possible values are listed in the following table.

Value	Class Constant	Description
3	OperationUpdate	The object was updated
4	OperationCreate	The object was created
6	OperationDelete	The object was deleted
8	OperationAutoProp	Indicates an automatic set property
9	OperationCollAdd	Indicates a manual collection add
10	OperationAutoAdd	Indicates an automatic collection add
11	OperationCollRemove	Indicates a manual collection remove
12	OperationAutoRemove	Indicates an automatic collection remove

prop	References a property that was modified by the transaction. The **object** parameter contains the owner of the property.
value	Contains the value of the **prop** parameter immediately after the update.

There can be more than one entry for the same object (for example, if an object is updated and then deleted in a transaction). However, objects that are updated more than once in a transaction usually appear once only in the list.

The list includes collections and other objects that are updated automatically.

getEntryCount

Signature `getEntryCount(): Integer;`

The **getEntryCount** method of the **JadeTransactionTrace** class returns the number of entries in the list held by the receiving **JadeTransactionTrace** instance.

JadeUserCollClass Class

The **JadeUserCollClass** class

For details about the methods defined in the **JadeUserCollClass** class and usage of this class, see "JadeUserCollClass Methods" and "Using JadeUserCollClass Collections", respectively, in the following subsections.

Inherits From: CollClass

Inherited By: (None)

JadeUserCollClass Methods

The methods defined in the **JadeUserCollClass** class are summarized in the following table.

Method	Description
addExternalKey	Adds an external key definition to a user class at run time
addMemberKey	Adds a member key definition to a user class at run time
clearKeys	Clears existing key definitions
endKeys	Indicates the end of a single or multiple key definition
setLength	Sets or changes the element length for an array
setMembership	Sets or changes the membership of a user class

For examples of the use of **JadeUserCollClass** class methods, see "Using JadeUserCollClass Collections", later in this chapter.

addExternalKey

Signature

```
addExternalKey(keyType:          PrimType;
               keyLength:        Integer;
               scaleFactor:      Integer;
               descending:       Boolean;
               caseInsensitive:  Boolean;
               sortOrder:        Integer) updating;
```

The **addExternalKey** method of the **JadeUserCollClass** class adds an external key specification to a user collection that is a subclass of **ExtKeyDictionary**.

Use the **keyType** to specify the primitive type for the key. (For more details, see "Pseudo Types" and "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.)

For **String**, **StringUtf8**, **Binary**, and **Decimal** keys, you must specify the **keyLength** parameter. This parameter is ignored for keys of other primitive types. For **Decimal** keys, you must also specify the **scaleFactor** parameter.

Set the **descending** parameter to **true** if you want keys sorted in descending order and the **caseInsensitive** parameter to **true** if case-sensitivity is not required.

For **String** and **StringUtf8** keys, the **sortOrder** parameter specifies the locale identifier for the locale used to order entries in the collection. This parameter is ignored for keys of other primitive types. A value of zero (**0**) indicates the binary sort order.

If you require multiple keys, call the **addExternalKey** method to define each key in sequence. To signify that all keys have been defined, call the **endKeys** method.

The following preconditions apply when adding keys to a dynamic dictionary.

- The collection is empty
- The member type has been specified by using the **setMembership** method
- The dictionary contains external key definitions only
- The total concatenated key size does not exceed the current key size limit (512 character units)

The appropriate system exception is raised if any of these preconditions are violated.

addMemberKey

Signature

```
addMemberKey(propertyName:    String;
             descending:      Boolean;
             caseInsensitive: Boolean;
             sortOrder:       Integer) updating;
```

The **addMemberKey** method of the **JadeUserCollClass** class adds a member key specification to a user collection that is a subclass of **MemberKeyDictionary**.

If you require multiple keys, call the **addMemberKey** method to define each key in sequence. To signify that all keys have been defined, call the **endKeys** method.

Specify a key path by passing a key-path expression in the **propertyName** parameter; for example, **"shipment.supplier.name"**. Set the **descending** parameter to **true** if you want keys sorted in descending order and the **caseInsensitive** parameter to **true** if case-sensitivity is not required.

For **String** and **StringUtf8** keys, the **sortOrder** parameter specifies the locale identifier for the locale used to order entries in the collection. This parameter is ignored for keys of other primitive types. A value of zero (**0**) indicates the binary sort order.

The following preconditions apply when adding keys to a dynamic dictionary.

- The collection is empty
- The member type has been specified by using the **setMembership** method
- The dictionary contains member key definition only
- The **propertyName** parameter represents a valid property for the member type
- The **propertyName** parameter is not an exclusive collection
- The total concatenated key size does not exceed the current key size limit (512 key units)

The appropriate system exception is raised if any of these preconditions are violated.

For an example of the use of the **addMemberKey** method, see "Using JadeUserCollClass Collections", later in this chapter.

clearKeys

Signature `clearKeys() updating;`

The **clearKeys** method of the **JadeUserCollClass** class clears existing dictionary key definitions so that the user collection can be reused.

Before the **clearKeys** method is called, the collection must be empty; that is, it cannot contain data. If this precondition is violated, the appropriate system exception is raised.

endKeys

Signature `endKeys(duplicatesAllowed: Boolean) updating;`

The **endKeys** method of the **JadeUserCollClass** class indicates the end of a single or multiple key specification.

Use the **duplicatesAllowed** parameter to specify whether the dictionary allows or disallows duplicate key entries.

At least one key must have been defined (by using the **addExternalKey** or **addMemberKey** method). If this precondition is violated, the appropriate system exception is raised.

For an example of the use of the **endKeys** method, see "Using JadeUserCollClass Collections", later in this chapter.

setLength

Signature
```
setLength(length:      Integer;
          scaleFactor: Byte) updating;
```

The **setLength** method of the **JadeUserCollClass** class sets or changes the element length for a user collection that is a subclass of **Array**.

The **length** parameter has a maximum value of:

- 16,000 for arrays with membership **String** or **Binary**
- 8,000 for arrays with membership **StringUtf8**
- 23 for arrays with membership **Decimal**

The scaleFactor parameter applies to arrays with membership **Decimal** only.

setMembership

Signature `setMembership(type: Class) updating;`

The **setMembership** method of the **JadeUserCollClass** class sets the membership (that is, the base type for members) of a user collection.

Before the **setMembership** method is called, the collection must be empty; that is, it cannot contain data. If this precondition is violated, the appropriate system exception is raised. This method implicitly calls the **clearKeys** method.

For an example of the use of this method, see "Using JadeUserCollClass Collections", later in this chapter.

Using JadeUserCollClass Collections

In the following example that shows the use of the **JadeUserCollClass** class, a user collection is defined as a subclass of **MemberKeyDictionary**. This collection class is used as the type for an exclusive runtime dynamic property that is added to the class of the root object.

```
vars
    dict : JadeUserCollClass;
    cluster : JadeDynamicPropertyCluster;
begin
    // Define the user collection as a member key dictionary
    beginTransaction;
    dict := currentSchema.addUserCollectionSubclass(MemberKeyDictionary,
                                                    "CustomersByName",
                                                    "dbfilename");
    dict.setMembership(Customer);
    dict.addMemberKey("lastName", false, true, 0);
    dict.endKeys(true);
    commitTransaction;

    // Make a runtime dynamic property using the user collection
    beginTransaction;
    cluster := Root.addDynamicPropertyCluster("RootCluster");
    cluster.addExclusiveDynamicProperty("allCustomersByName", dict);
    commitTransaction;
end;
```

JadeWebService Class

The **JadeWebService** class maintains all Web service information.

Note Methods declared on the **JadeWebService** class and its subclasses that are marked as Web service methods cannot have a return type of **Any** and cannot have parameters of type **Any**. (For details about specifying Web service methods, see "webService Option" under "Method Options", in Chapter 1 of the *JADE Developer's Reference*.)

JADE timestamp values use the local time zone. External Web service consumers often expect Coordinated Universal Time (UTC) values and external Web service providers often return UTC values. You may need to convert between UTC and local timestamp values, by using the **localToUTCTime** and **utcToLocalTime** methods of the **TimeStamp** primitive type.

For details about the constants and methods defined in the **JadeWebService** class, see "JadeWebService Class Constants" and "JadeWebService Methods", in the following subsections.

Inherits From: Object

Inherited By: JadeWebServiceConsumer, JadeWebServiceProvider, JadeWebServiceSoapHeader

JadeWebService Class Constants

The constants defined in the **JadeWebService** class are listed in the following table.

JadeWebService Class Constant	Integer or String Value
RIA_MODULE_API	1
RIA_MODULE_TYPES	2
RIA_MODULE_UTIL	0
SOAP_12_RPC	"http://www.w3.org/2003/05/soap-rpc"

JadeWebService Methods

The methods defined in the **JadeWebService** class are summarized in the following table.

Method	Description
isNilItem	Returns **true** if the specified element in the receiver has a **nil** attribute value of **true**
setAnyPropType	Specifies the type of an imported primitive of type Any in an external Web service application
setError	Specifies the Web service provider error code, item, and text of the SOAP error

isNilItem

Signature
```
isNilItem(obj:          Object;
          propertyName: String;
          indx:         Integer): Boolean;
```

The **isNilItem** method of the **JadeWebService** class returns **true** if the item defined in the method parameters was specified as having a **nil** state in the XML message that is being processed.

The **isNilItem** method parameters are listed in the following table.

Parameters	Description
obj	The JADE object containing the property to be examined or an array containing the index to be examined
propertyName	The JADE name of the property in the object to be examined or null (**""**) when examining the object itself
indx	The array index of the entry to be examined when an array is specified by the other parameters

In a **JadeWebServiceConsumer** method, call this method to determine whether an item received in response to a Web service request is specified as having a **nil** state, as shown in the following code fragment.

```
if myWebServiceConsumer.isNilItem(employee, 'spouse', null) then
    // employee has no spouse
endif;
if myWebServiceConsumer.isNilItem(addressArray, null, 3) then
    // addressArray[3] is empty
endif;
```

In a **JadeWebServiceProvider** method, call this method to determine whether an item received in the Web service request is specified as having a **nil** value, as shown in the following code fragment.

```
if isNilItem(employee, 'spouse', null) then
    // employee has no spouse
endif;
if isNilItem(addressArray, null, 3) then
    // addressArray[3] is empty
endif;
```

For both Web service consumer and Web service provider calls, the item received will have a JADE value of **null**.

Note The **nil** state of an element applies only to element values and not to attribute values. (An element whose **nil** value is **true** may not have any element content, but it may still carry attributes.)

setAnyPropType

Signature
```
setAnyPropType(prop:     Object;
               primType: Object) updating;
```

The **setAnyPropType** method of the **JadeWebService** class is used to handle primitive types that are defined as being of type **Any** in an external Web service application.

When the WSDL for the Web service is imported into JADE, the properties in the Web service classes have the corresponding types corresponding to the primitive entities; that is, strings are of type **String**, integers of type Integer, and so on. However, JADE does not permit properties to be defined as being of type **Any**, so the default type is **String** for those properties.

You can use the **setAnyPropType** to specify that the property specified by the **prop** parameter is actually of the type specified by the **primType** parameter, thereby enabling JADE to generate the appropriate Web service request.

setError

Signature

```
setError(errorCode: Integer;
         errorItem: String;
         errorText: String);
```

The **setError** method of the **JadeWebService** class specifies the error code, item, and text of the SOAP error. This generates a SOAP fault to be returned to the client.

If you do not want to return a SOAP fault, you must handle it differently (for example, by sending a message back as part of your response).

JadeWebServiceConsumer Class

The JADE Web service consumer enables you to access external Web services (including JADE Web services) from within your JADE application. A Web service consumer cannot be called asynchronously from a JADE application. When a WSDL file is imported into JADE, a subclass of the **JadeWebServiceConsumer** class is created for each service that is defined in the WSDL.

Note The JADE Web services framework does not have special code for cookie handling. This is left to the underlying Microsoft Windows Internet (WinINet) or Microsoft Windows HTTP Services (WinHTTP) library to manage.

When a JADE Web services consumer is first invoked, the WINHTTP (default) or WININET library is loaded and this is shared by all consumers within the node. As cookies are managed by the library, the same cookies are sent with every consumer Web service request from the node. For consumers to have unique cookies, they must run in separate nodes.

For details about the constants, properties, and methods defined in the **JadeWebServiceConsumer** class, see "JadeWebServiceConsumer Class Constants", "JadeWebServiceConsumer Properties", and "JadeWebServiceConsumer Methods", in the following subsections. See also "JADE Web Service Consumer", in Chapter 11 of the *JADE Developer's Reference* and "Generating a Web Service Consumer Unit Test Class and Stub Methods", in Chapter 17 of the *JADE Developer's Reference*.

Inherits From: JadeWebService

Inherited By: (None)

JadeWebServiceConsumer Class Constants

The constants defined in the **JadeWebServiceConsumer** class are listed in the following table.

JadeWebService Class Constant	Integer or String Value
INTERNET_OPEN_TYPE_DIRECT	1
INTERNET_OPEN_TYPE_PRECONFIG	0
INTERNET_OPEN_TYPE_PRE_NOAUTO	4
INTERNET_OPEN_TYPE_PROXY	3
Jdo_Delimiter	'::'
Jdo_in_use	2
Jdo_PropertyNamePrefix_w_	'w_'
Jdo_PropertyNamePrefix_wsp_	'wsp_'

JadeWebServiceConsumer Properties

The properties defined in the **JadeWebServiceConsumer** class are summarized in the following table.

Property	Description
characterConversionException	Specifies whether a Web service response on an ANSI system contains non-ANSI characters

Property	Description
handleCharConversionException	Specifies whether an exception is raised on an ANSI system if a Web service response contains non-ANSI characters
logStatistics	Value that specifies whether Web service request statistics are logged
password	Web service consumer user authentication password, if required
proxyHostName	Host name of the proxy server for the Web service consumer, if required
proxyPassword	Web service consumer user authentication password for proxy servers, if required
proxyUsername	Web service consumer user authentication identifier for proxy servers, if required
soapHeaders	List of SOAP headers sent by the Web service consumer
soapRequest	Outgoing SOAP message sent to the Web service provider
soapResponse	SOAP message response received from the Web service provider
timeout	Number of milliseconds after which the Web service times out if a response has not been received
unknownHeaders	Contains an array of any SOAP headers that were part of the response but could not be processed
userName	Web service consumer user authentication identifier, if required
workerApp	Name of a worker application configured to process a Web service request asynchronously

characterConversionException

Type: Boolean

The **characterConversionException** property of the **JadeWebServiceConsumer** class is set to **true** by the framework if a Web service response on an ANSI system contains non-ANSI characters (that is, characters with a code greater than 127) and the option not to raise a character conversion exception has been set. For details about whether a character conversion exception is raised, see the **handleCharConversionException** property.

If the **characterConversionException** property is **true**, the value of the **soapResponse** property is the unconverted response string containing the non-ANSI characters.

handleCharConversionException

Type: Boolean

The **handleCharConversionException** property of the **JadeWebServiceConsumer** class specifies whether an exception is raised by an ANSI JADE system if a response from a Web service contains non-ANSI characters (that is, characters with a code greater than 127).

Note A character conversion exception occurs only on a Unicode system if the Web service response contains invalid UTF8 characters.

If the value of the property is **false**, the default value, a character conversion exception is raised and the **extendedErrorText** property of the exception object is set to the unconverted response string containing the non-ANSI characters. If the value of the property is **true**, instead of raising an exception, the **characterConversionException** property is set to **true** and the value of the **soapResponse** property is set to the unconverted response string containing the non-ANSI characters.

logStatistics

Type: Boolean

The **logStatistics** property of the **JadeWebServiceConsumer** class specifies whether statistics are logged for a Web service request. The default value is **false**.

password

Type: String

The **password** property of the **JadeWebServiceConsumer** class contains a password, if user authentication is required by a site.

This property is used for authentication in conjunction with the **userName** property, if required, and the JADE Web services framework sends this information when requesting a connection.

The default value is null (**""**).

proxyHostName

Type: String

If your site uses proxy servers and these servers require authentication, the **proxyHostName** property of the **JadeWebServiceConsumer** class contains the host name of the proxy server to which the JADE Web services framework connects.

proxyPassword

Type: String

If your site uses proxy servers and these servers require authentication, the **proxyPassword** property of the **JadeWebServiceConsumer** class contains the proxy password that the JADE Web services framework sends when requesting a connection.

This property is used for proxy server authentication in conjunction with the **proxyUsername** property. The default value is null (**""**).

proxyUsername

Type: String

If your site uses proxy servers and these servers require authentication, the **proxyUsername** property of the **JadeWebServiceConsumer** class contains the proxy user identifier that the JADE Web services framework sends when requesting a connection.

This property is used for proxy server authentication in conjunction with the **proxyPassword** property. The default value is null (**""**).

soapHeaders

Type: ObjectArray

The **soapHeaders** property of the **JadeWebServiceConsumer** class contains a reference to an object array of SOAP headers that were sent to the Web service provider by the Web service consumer.

soapRequest

Type: String

The **soapRequest** property of the **JadeWebServiceConsumer** class contains the outgoing SOAP message that is sent to the Web service provider.

soapResponse

Type: String

The **soapResponse** property of the **JadeWebServiceConsumer** class contains the SOAP message that was sent to the Web service consumer from the Web service provider.

timeout

Type: Integer

The **timeout** property of the **JadeWebServiceConsumer** class contains the number of milliseconds after which a Web service consumer session times out if no SOAP message is received from the Web service provider.

The timeout value remains active until you reset the value in your application for that transient instance of the Web service consumer object. For details about setting and getting timeout values for connect, send, and receive messages, see the **getTimeouts** and **setTimeouts** methods.

If you do not set this property, the request times out after two minutes (that is, **120,000** milliseconds).

When you specify the number of milliseconds after which control is regained if the remote server fails to respond and the specified time is exceeded, a **JadeSOAPException** (exception 11052) is raised and the body of the message states:

```
HTTP Error 12002 HTTP Send Request Failed
```

Error 12002 is a WinINET or WinHTTP error that indicates that the request has timed out.

You can control the length of time that the JADE Web service consumer waits for the response by using the **JadeWebServiceConsumer** class **setTimeouts** method.

unknownHeaders

Type: JadeWebServiceUnknownHdrArray

The **unknownHeaders** property of the **JadeWebServiceConsumer** class contains an array of any SOAP headers that were part of the response but could not be processed.

userName

Type: String

The **userName** property of the **JadeWebServiceConsumer** class contains the name of a valid user id, if user authentication is required by a site.

This property is used for authentication in conjunction with the **password** property, if required, so that the JADE Web services framework sends this information when requesting a connection. The default value is null (**""**).

workerApp

Type: String[100]

The **workerApp** property of the **JadeWebServiceConsumer** class contains the name of a worker application that can process a Web service request asynchronously. The name must be the name of an application defined in the same schema or a superschema of the one containing the **JadeWebServiceConsumer** class.

The **initialize** and **finalize** methods of the application must execute the **asyncInitialize** and **asyncFinalize** methods of the **Application** class, respectively. Additionally, the WSDL for the Web service must specify that the Web service is to be executed asynchronously.

JadeWebServiceConsumer Methods

The methods defined in the **JadeWebServiceConsumer** class are summarized in the following table.

Method	Description
addHttpHeader	Adds, changes, or removes HTTP headers from a Web service consumer request
getEndpointURL	Returns the name of the end-point URL to which the Web service consumer request is sent
getHttpHeader	Returns the value of a specified user-defined HTTP header
getHttpHeaderClient	Returns the value of a specified client HTTP header sent with a Web service request
getHttpHeaderServer	Returns the value of a specified server HTTP header sent with a Web service response
getLastStatistics	Returns statistics relating to the last Web service consumer SOAP message
getTimeouts	Returns the timeout values in milliseconds for connect, send, and receive messages, respectively
invoke	Sends the message to your Web service provider using your own communication handlers or dynamically connects to a Web service (that is, without using or importing a WSDL file)
invokeAsync	Sends the message asynchronously to your Web service provider using your own communication handlers or dynamically connects to a Web service (that is, without using or importing a WSDL file)
invokeAsyncWithVerb	Sends the message and the specified verb asynchronously to your Web service provider using your own communication handlers or dynamically connects to a Web service (that is, without using or importing a WSDL file)

Method	Description
invokeWithVerb	Sends the message and the specified verb to your Web service provider using your own communication handlers or dynamically connects to a Web service (that is, without using or importing a WSDL file)
processReply	Processes the result of a Web service request (that is, a SOAP message) and sets up transient objects for further processing by your application
reset	Deletes all transient objects created by the Web service consumer when making a Web service request
sendRequest	Sets up the SOAP message for a Web service request and sends the message to the Web service provider
setEndpointURL	Dynamically changes the URL to which the Web service request is sent
setTimeouts	Sets the timeout values for connect, send, and receive messages, respectively

addHttpHeader

Signature `addHttpHeader(key: String; value: String);`

The **addHttpHeader** method of the **JadeWebServiceConsumer** class enables you to add, change, or remove HTTP headers from a Web service consumer request.

The value of the **key** parameter is the HTTP header to create and the value of the **value** parameter is the value to assign to that key.

In the following code fragment, the **Authorization: Basic c29hcHRlc3Q6cGFzc3dvcmQ=\r\n** header is added to a request message sent to a Web service provider.

```
webService.addHttpHeader("Authorization",
                         "Basic c29hcHRlc3Q6cGFzc3dvcmQ=\r\n");
```

The following example shows the removal of an HTTP header with a **key** of **Authorization**.

```
webService.addHttpHeader("Authorization", "");
```

Note The framework creates and passes the HTTP headers to the underlying libraries (**wininet.dll** and **winhttp.dll**) that you specified and does not attempt to validate their accuracy. It is your responsibility to ensure that the HTTP headers are valid.

getEndpointURL

Signature `getEndpointURL(): String;`

The **getEndpointURL** method of the **JadeWebServiceConsumer** class returns the name of the end-point URL to which a Web service request is sent.

The default value of the URL end-point is obtained from the WSDL file.

getHttpHeader

Signature `getHttpHeader(key: String): String;`

The **getHttpHeader** method of the **JadeWebServiceConsumer** class returns the value of a user-defined HTTP header specified by the value of the **key** parameter.

If the HTTP header does not exist, a null string is returned.

getHttpHeaderClient

Signature `getHttpHeaderClient(key: String): String;`

The **getHttpHeaderClient** method of the **JadeWebServiceConsumer** class returns the value of a client HTTP header specified by the value of the **key** parameter. Client HTTP headers are sent with a Web service request.

If the HTTP header does not exist, a null string is returned; for example, a Web service request has the following HTTP headers.

```
Accept: text/plain
Accept: text/html
Accept: text/xml
Content-Type: text/xml; charset=utf-8
Host: cnwcrs1a
Pragma: no-cache
Proxy-Connection: Keep-Alive
SOAPAction: "urn:JadeWebServices/CalculatorService/add"
User-Agent: Jade/9.9.00
```

The following code fragment shows the **getHttpHeaderClient** method used with the example Web service request.

```
// Output from the next instruction is "Keep-Alive"
write webService.getHttpHeaderClient("Proxy-Connection");
```

getHttpHeaderServer

Signature `getHttpHeaderServer(key: String): String;`

The **getHttpHeaderServer** method of the **JadeWebServiceConsumer** class returns the value of a server HTTP header specified by the value of the **key** parameter. Server HTTP headers are sent in a response message from the server.

If the HTTP header does not exist, a null string is returned; for example, a Web service response has the following HTTP headers.

```
HTTP/1.1 200 OK
Content-Length: 1034
Content-Type: text/xml; charset=utf-8
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Thu, 15 Apr 2010 03:03:04 GMT
```

The following code fragment shows the **getHttpHeaderServer** method used with the example Web service response.

```
// Output from the next instruction is "ASP.NET"
write webService.getHttpHeaderServer("X-Powered-By");
```

getLastStatistics

Signature `getLastStatistics(headerOnly: Boolean): String;`

The **getLastStatistics** method of the **JadeWebServiceConsumer** class returns a string containing statistics data of the last Web service request.

Note Web service statistics are logged only when the **logStatistics** property is set to **true**.

If the **headerOnly** parameter is set to **true**, the data listed in the following table is returned (in XML format).

Value	Description
name	Web service name
operation	Web service operation that was invoked
url	Uniform Resource Locator (URL) of the Web service
dateTime	Timestamp of when the request was sent
responseTime	Time from the time the request is sent to the time a response is received, including the time the Web service spends processing user logic
processingTime	Total time the Web service consumer takes to formulate the request plus the time the Web service consumer takes to process the reply; that is, the total time taken by the Web service consumer, excluding the time in the Web service itself
errorCode	Error code if an error was returned by the Web service
requestSize	Size of the SOAP message request
responseSize	Size of the SOAP message response

In addition to the above data that is returned when the **headerOnly** parameter is set to **true**, the data listed in the following table is also returned (in XML format) when the **headerOnly** parameter is set to **false**.

Value	Description
requestHeaders	HTTP request headers
soapRequest	The SOAP request message
responseHeaders	HTTP response headers
soapResponse	The SOAP response message

The following example of the data returned from the **getLastStatistics** method invokes an operation called **getQuote** from the Web service at **http://www.webservicex.netstockquote.asmx**. In this example, all details are returned because the **headerOnly** parameter is set to **false**.

```
<?xml version="1.0" encoding="utf-8"?>
<WebServiceStatistics>
    <name>StockQuote</name>
```

```
    <operation>getQuote</operation>
    <url>http://www.webservicex.netstockquote.asmx</url>
    <dateTime>30 March 2004, 11:00:54</dateTime>
    <responseTime>846</responseTime>
    <processingTime>3</processingTime>
    <errorCode>0</errorCode>
    <requestSize>349</requestSize>
    <responseSize>988</responseSize>
    <requestHeaders>Accept: text/plain
Accept: text/html
Accept: text/xml
Content-Type: text/xml; charset=utf-8
Host: www.webservicex.net
Pragma: no-cache
Proxy-Connection: Keep-Alive
SOAPAction: http://www.webserviceX.NET/GetQuote
User-Agent: Jade/2016
    </requestHeaders>
    <soapRequest>&lt;?xml version=&quot;1.0&quot;
    encoding=&quot;utf-8&quot;?&gt;
&lt;soap:Envelope
xmlns:xsi=&quot;http://www.w3.org/2001/XMLSchema-instance&quot;
xmlns:xsd=&quot;http://www.w3.org/2001/XMLSchema&quot;
xmlns:soap=&quot;http://schemas.xmlsoap.org/soap/envelope/&quot;&gt;
    &lt;soap:Body&gt;
        &lt;GetQuote xmlns=&quot;http://www.webserviceX.NET/&quot;&gt;
                &lt;symbol&gt;IBM&lt;/symbol&gt;
        &lt;/GetQuote&gt;
    &lt;/soap:Body&gt;
&lt;/soap:Envelope&gt;
    </soapRequest>
    <responseHeaders>HTTP/1.0 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 29 Mar 2004 23:00:53 GMT
X-Powered-By: ASP.NET
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 988
X-Cache: MISS from localhost.localdomain
Proxy-Connection: close
    </responseHeaders>
    <soapResponse>&lt;?xml version=&quot;1.0&quot;
    encoding=&quot;utf-8&quot;?&gt;&lt;soap:Envelope
xmlns:soap=&quot;http://schemas.xmlsoap.org/soap/envelope/&quot;
xmlns:xsi=&quot;http://www.w3.org/2001/XMLSchema-instance&quot;
xmlns:xsd=&quot;http://www.w3.org/2001/XMLSchema&quot;&gt;&lt;soap:Body&gt;
&lt;GetQuoteResponse
xmlns=&quot;http://www.webserviceX.NET/&quot;&gt;&lt;GetQuoteResult&gt;&amp;
lt;StockQuotes&amp;gt;&amp;lt;Stock&amp;gt;&amp;lt;Symbol&amp;gt;IBM&amp;lt;
/Symbol&amp;gt;&amp;lt;Last&amp;gt;92.68&amp;lt;/Last&amp;gt;&amp;lt;
Date&amp;gt;3/29/2004&amp;lt;/Date&amp;gt;&amp;lt;Time&amp;gt;4:01pm&amp;lt;
/Time&amp;gt;&amp;lt;Change&amp;gt;-0.09&amp;lt;/Change&amp;gt;&amp;
lt;Open&amp;gt;92.99&amp;lt;/Open&amp;gt;&amp;lt;High&amp;gt;93.61&amp;lt;
/High&amp;gt;&amp;lt;Low&amp;gt;92.18&amp;lt;/Low&amp;gt;&amp;lt;Volume&amp;
gt;4876300&amp;lt;/Volume&amp;gt;&amp;lt;MktCap&amp;gt;157.5B&amp;lt;
```

```
/MktCap&amp;gt;&amp;lt;PreviousClose&amp;gt;92.77&amp;lt;/PreviousClose&amp;
gt;&amp;lt;PercentageChange&amp;gt;-0.10%&amp;lt;/PercentageChange&amp;
gt;&amp;lt;AnnRange&amp;gt;78.12 -100.43&amp;lt;/AnnRange&amp;gt;&amp;lt;
Earns&amp;gt;4.34&amp;lt;/Earns&amp;gt;&amp;lt;P-E&amp;gt;21.38&amp;lt;
/P-E&amp;gt;&amp;lt;Name&amp;gt;INTL BUSMACHINE&amp;lt;/Name&amp;gt;&amp;lt;
/Stock&amp;gt;&amp;lt;/StockQuotes&amp;gt;&lt;/GetQuoteResult&gt;&lt;
/GetQuoteResponse&gt;&lt;/soap:Body&gt;&lt;/soap:Envelope&gt;
    </soapResponse>
</WebServiceStatistics>
```

getTimeouts

Signature

```
getTimeouts(connectTimeout: Integer output;
            sendTimeout:    Integer output;
            receiveTimeout: Integer output);
```

The **getTimeouts** method of the **JadeWebServiceConsumer** class returns the timeout values in milliseconds for connect, send, and receive messages, respectively.

The **connectTimeout**, **sendTimeout**, and **receiveTimeout** parameters are populated with the number of milliseconds after which a Web service consumer session times out if no SOAP connect, send, or receive message is received from the Web service provider.

See also the **timeout** property (which you can use to set all three message types to the same value) and the **setTimeouts** method.

invoke

Signature

```
invoke(inputMessage: String): String updating;
```

The **invoke** method of the **JadeWebServiceConsumer** class sends the SOAP-formatted message specified in the **inputMessage** parameter to the Web service and returns the response from the Web service provider (a SOAP message).

Re-implement the **invoke** method in the following situations.

- If you do not want to use the JADE Web service communications framework, which currently supports only HTTP (for example, if you prefer to use SMTP rather than the HTTP protocol), so that you can use your own communication handlers.
- If you want to dynamically connect to a Web service (that is, without using or importing a WSDL file).

  For example, if you have a **JadeWebServiceConsumer** subclass called **DoItMyself**, the method in the following example illustrates calling the Amazon Web service dynamically.

```
vars
    doItMyself : DoItMyself;
    inputMsg   : String;
    outputMsg  : String;
begin
    create doItMyself transient;
    doItMyself.setEndpointURL('http://soap.amazon.com/onca/soap2');
    inputMsg := ""; //soap request here
    outputMsg := doItMyself.invoke(inputMsg);
    // outputMsg will now contain the response from the Web service
    // provider or a SOAP fault raised by the JADE Web services
```

```
    // framework (for example, if the connection failed)
epilog
    delete doItMyself;
end;
```

Note In this case, you are dealing directly with SOAP messages.

This method uses the **POST** verb to send a message to the Web service via the associated **JadeHTTPConnection**. Use the **JadeWebServiceConsumer** class **invokeWithVerb** method to send a message to the Web service using a verb other than **POST**.

invokeAsync

Signature `invokeAsync(inputMessage: String): JadeMethodContext updating;`

The **invokeAsync** method of the **JadeWebServiceConsumer** class sends the SOAP-formatted message specified in the **inputMessage** parameter to the Web service and returns an instance of the **JadeMethodContext** class that handles the asynchronous execution of the Web service request and waits for the response from the Web service provider (a SOAP message).

The **workerApp** property must be set to the name of an asynchronous worker application, which must execute the **asyncInitialize** and **asyncFinalize** methods of the **Application** class in the **initialize** and **finalize** methods of the application, respectively.

The **JadeMethodContext** instance can be used as input to the **waitForMethods** method on the **Process** class, to enable your code to wait for the completion of the asynchronous execution of the Web service and obtain the results.

Reimplement the **invokeAsync** method in the following situations.

- If you do not want to use the JADE Web service communications framework, which currently supports only HTTP (for example, if you prefer to use SMTP rather than the HTTP protocol), so that you can use your own communication handlers.
- If you want to dynamically connect to a Web service (that is, without using or importing a WSDL file). If you have a **JadeWebServiceConsumer** subclass called **DoItMyself**, the method in the following example illustrates calling the Amazon Web service dynamically.

```
vars
    doItMyself : DoItMyself;
    inputMsg   : String;
    context    : JadeMethodContext;
    outputMsg  : String;
begin
    create doItMyself transient;
    doItMyself.workerApp := "AsyncWorkerApp";
    doItMyself.setEndpointURL('http://soap.amazon.com/onca/soap2');
    inputMsg := ""; //soap request here
    context  := doItMyself.invokeAsync(inputMsg);
    // context now contains the JadeMethodContext that will send
    // the Web service request in a worker application and receive
    // the response from the Web service provider
    process.waitForMethods(context);
    // wait for asynchronous Web service message to complete
    // could do other processing while waiting for the completion
    outputMsg := context.getReturnValue.String;
```

```
    // outputMsg now contains the response from the Web service
    write outputMsg;
epilog
    delete doItMyself;
end;
```

Note In this case, you are dealing directly with SOAP messages.

This method uses the **POST** verb to send a message to the Web service via the associated **JadeHTTPConnection**. Use the **JadeWebServiceConsumer** class **invokeAsyncWithVerb** method to send a message to the Web service using a verb other than **POST**.

invokeAsyncWithVerb

Signature `invokeAsyncWithVerb(inputMessage: String; verbIn: String): JadeMethodContext updating;`

The **invokeAsyncWithVerb** method of the **JadeWebServiceConsumer** class sends the SOAP-formatted message specified in the **inputMessage** parameter, using the verb specified in the **verbIn** parameter, to the Web service and returns an instance of the **JadeMethodContext** class that handles the asynchronous execution of the Web service request and waits for the response from the Web service provider (a SOAP message).

This method does the same as the **JadeWebServiceConsumer** class **invokeAsync** method, except that the verb specified in the **verbIn** parameter is used (for example, **"GET"** or **"PUT"**) instead of **"POST"**. Calling the **invokeAsync** method is the same as calling the **invokeAsyncWithVerb** method with **"POST"** specified in the **verbIn** parameter.

The **workerApp** property must be set to the name of an asynchronous worker application, which must execute the **asyncInitialize** and **asyncFinalize** methods of the **Application** class in the **initialize** and **finalize** methods of the application, respectively.

The **JadeMethodContext** instance can be used as input to the **waitForMethods** method on the **Process** class, to enable your code to wait for the completion of the asynchronous execution of the Web service and obtain the results.

Reimplement the **invokeAsyncWithVerb** method in the following situations when you also need to specify the verb that is used.

- If you do not want to use the JADE Web service communications framework, which currently supports only HTTP (for example, if you prefer to use SMTP rather than the HTTP protocol), so that you can use your own communication handlers.
- If you want to dynamically connect to a Web service (that is, without using or importing a WSDL file). If you have a **JadeWebServiceConsumer** subclass called **DoItMyself**, the method in the following example illustrates calling the Amazon Web service dynamically.

```
vars
    doItMyself : DoItMyself;
    inputMsg   : String;
    context    : JadeMethodContext;
    outputMsg  : String;
begin
    create doItMyself transient;
    doItMyself.workerApp := "AsyncWorkerApp";
    doItMyself.setEndpointURL('http://soap.amazon.com/onca/soap2');
    inputMsg := ""; //soap request here
```

```
    context  := doItMyself.invokeAsyncWithVerb(inputMsg, verb);
    // context now contains the JadeMethodContext that will send
    // the Web service request in a worker application and receive
    // the response from the Web service provider
    process.waitForMethods(context);
    // wait for asynchronous Web service message to complete
    // could do other processing while waiting for the completion
    outputMsg := context.getReturnValue.String;
    // outputMsg now contains the response from the Web service
    write outputMsg;
epilog
    delete doItMyself;
end;
```

Note In this case, you are dealing directly with SOAP messages.

You can use the **invokeAsyncWithVerb** method, for example, to allow users to access a REST service using the HTTP **GET** verb.

invokeWithVerb

Signature `invokeWithVerb(inputMessage: String; verbIn: String): String updating;`

The **invokeWithVerb** method of the **JadeWebServiceConsumer** class sends the SOAP-formatted message specified in the **inputMessage** parameter, using the verb specified in the **verbIn** parameter, to the Web service and returns the response from the Web service provider (a SOAP message).

This method does the same as the **JadeWebServiceConsumer** class **invoke** method, except that the verb specified in the **verbIn** parameter is used (for example, **"GET"** or **"PUT"**) instead of **"POST"**. Calling the **invoke** method is the same as calling the **invokeWithVerb** method with **"POST"** specified in the **verbIn** parameter.

Re-implement the **invokeWithVerb** method in the following situations when you also need to specify the verb that is used.

- If you do not want to use the JADE Web service communications framework, which currently supports only HTTP (for example, if you prefer to use SMTP rather than the HTTP protocol), so that you can use your own communication handlers.
- If you want to dynamically connect to a Web service (that is, without using or importing a WSDL file).

  For example, if you have a **JadeWebServiceConsumer** subclass called **DoItMyself**, the method in the following example illustrates calling the Amazon Web service dynamically.

```
vars
    doItMyself : DoItMyself;
    inputMsg   : String;
    outputMsg  : String;
begin
    create doItMyself transient;
    doItMyself.setEndpointURL('http://soap.amazon.com/onca/soap2');
    inputMsg := ""; //soap request here
    outputMsg := doItMyself.invokeWithVerb(inputMsg, verb);
    // outputMsg will now contain the response from the Web service
    // provider or a SOAP fault raised by the JADE Web services
    // framework (for example, if the connection failed)
```

```
epilog
    delete doItMyself;
end;
```

Note In this case, you are dealing directly with SOAP messages.

You can use the **invokeWithVerb** method, for example, to allow users to access a REST service using the HTTP **GET** verb.

processReply

Signature `processReply(): Any protected, updating;`

Reimplement the **processReply** method in your user-defined **JadeWebServiceConsumer** subclasses if you want to take the result of a Web service request (that is, a SOAP message), process the message, and set up transient objects for further processing by your application.

Note If the JADE implementation of this method is not called (by using the **inheritMethod** instruction), it is your responsibility to do any processing that is necessary. For details, see the **JadeWebServiceConsumer** class **sendRequest** method.

reset

Signature `reset() updating;`

The **reset** method of the **JadeWebServiceConsumer** class removes all transient objects that were created by the Web services framework when making a Web service request.

This method deletes all transient objects created by the Web service consumer but retains the Web service consumer. By default, these transient objects are deleted only when the Web service consumer is deleted.

The following code fragment shows the use of the rest method.

```
websvc.reset;
```

sendRequest

Signature `sendRequest(methodName: String): Any updating;`

Re-implement the **sendRequest** method of the **JadeWebServiceConsumer** class if you want to set up the SOAP message for a Web service request and send the message to the Web service provider.

Use the **methodName** parameter to specify the Web service method to invoke.

The **sendRequest** method returns the result of the invocation.

setEndpointURL

Signature `setEndpointURL(endpoint: String) updating;`

The **setEndpointURL** method of the **JadeWebServiceConsumer** class cannot be reimplemented in your user-defined Web service consumer subclasses.

To change the end-point URL (for example, if you want to dynamically change the end-point of the URL for your Web service consumer instance at run time), call the **setEndpointURL** method, passing the appropriate end-point URL as a parameter.

setTimeouts

Signature
```
setTimeouts(connectTimeout: Integer;
            sendTimeout:    Integer;
            receiveTimeout: Integer);
```

The **setTimeouts** method of the **JadeWebServiceConsumer** class sets the timeout values for connect, send, and receive messages, respectively.

By default, Web service consumer messages time out after 2 minutes (120,000 milliseconds).

Use the **connectTimeout**, **sendTimeout**, and **receiveTimeout** parameters to specify the respective number of milliseconds after which a Web service consumer session times out if no SOAP connect, send, or receive message is received from the Web service provider.

When specify the number of milliseconds after which control is regained if the remote server fails to respond and the specified time is exceeded, a **JadeSOAPException** (exception 11052) is raised and the body of the message states:

```
HTTP Error 12002 HTTP Send Request Failed
```

Error 12002 is a WinINET or WinHTTP error that indicates that the request has timed out.

See also the **timeout** property (which you can use to set all three message types to the same value) and the **getTimeouts** method.

JadeWebServiceProvider Class

The **JadeWebServiceProvider** class maintains all Internet service provider information.

For details about the properties and methods defined in the **JadeWebServiceProvider** class, see "JadeWebServiceProvider Properties" and "JadeWebServiceProvider Methods", in the following subsections.

Inherits From: JadeWebService

Inherited By: (None)

JadeWebServiceProvider Properties

The properties defined in the **JadeWebServiceProvider** class are summarized in the following table.

Property	Description
deleteTransientReturnType	Specifies whether the transient object return type from a Web service method is deleted when processing is complete
incomingMessage	Contains the incoming SOAP message string from the Web service consumer
rawXML	Specifies whether the Web service framework does any further XML processing of the data that is returned from a Web service method
unknownHeaders	Contains an array of any SOAP headers that were part of the request but could not be processed

deleteTransientReturnType

Type: Boolean

The **deleteTransientReturnType** property of the **JadeWebServiceProvider** class specifies whether the transient object return type from a Web service method is deleted when processing is complete. The object is deleted only if it is a transient object.

If the return type is a collection, all transient members of this collection are also deleted.

This property is set to **true** by default. If you do not want the framework to delete transient return types, you must set this property to **false** in your code.

incomingMessage

Type: String

The **incomingMessage** property of the **JadeWebServiceProvider** class contains the incoming SOAP message string sent by the Web service consumer.

rawXML

Type: Boolean

The **rawXML** property of the **JadeWebServiceProvider** class specifies whether the Web services framework does any further XML processing of the data that is returned from a Web service method. This property is set to **false**, by default.

When you do not want any further processing of the data returned from the Web service performed, set this property to **true**. The **<body>** element of the SOAP message then contains the returned value.

unknownHeaders

Type: JadeWebServiceUnknownHdrArray

The **unknownHeaders** property of the **JadeWebServiceProvider** class contains an array of any SOAP headers that were part of the request but could not be processed.

JadeWebServiceProvider Methods

The methods defined in the **JadeWebServiceProvider** class are summarized in the following table.

Method	Descriptions
createVirtualDirectoryFile	Passes files created by a JADE application to the **jadehttp** library
deleteVirtualDirectoryFile	Deletes specified files from the virtual directory used by the **jadehttp** library
getLastStatistics	Returns an XML-formatted string containing information about the current request
getServerVariable	Returns HyperText Transfer Protocol (HTTP) header information for your Web service request
initialize	Sets up the appropriate options for the specified Web service application when the default HTTP implementation is not used
isVDFilePresent	Returns **true** if the specified file is present in the virtual directory used by the **jadehttp** library
processMessage	Calls the relevant Web service method and returns the result of the processing as a SOAP message
processRequest	Processes Web service requests received from a Web service consumer
processRequestPostHeaders	Processes Web service requests received from a Web service consumer after the SOAP headers for the request have been processed
reply	Executed when a request is received from a Web service consumer

createVirtualDirectoryFile

Signature

```
createVirtualDirectoryFile(filename: String;
                           contents: Binary;
                           retain:   Boolean): Integer;
```

The **createVirtualDirectoryFile** method of the **JadeWebServiceProvider** class enables you to pass files created by a JADE application to the **jadehttp** library. The **jadehttp** library creates the specified file in the directory specified by the **VirtualDirectory** parameter in the **jadehttp.ini** file.

The **createVirtualDirectoryFile** method parameters are listed in the following table.

Parameter	Description
fileName	Name of the file to be created in the virtual directory

Parameter	Description
contents	Binary holding the file contents
retain	Creates read-only files when set to **true** or standard files when set to **false**

The **jadehttp** library creates the specified file in the directory (the virtual directory visible to Web browsers) in which the library is running. This method returns zero (**0**) if the method successfully formats a request to the **jadehttp** library or it returns the non-zero operating system error code indicating the failure to create the file.

You can specify whether files created in the virtual directory are deleted automatically and how this happens, by setting the **PurgeDirectoryRule** parameter in the [*application-name*] section of the **jadehttp.ini** file or the **PurgeDirectoryRule** configuration directive in the JADE **mod_jadehttp** file. If this parameter or directive is not set, files of type .**jpg**, .**png**, or .**gif** that are more than 12 hours old are removed. For details, see "Internal Housekeeping of the Virtual Directory", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Note This method must be called during the processing cycle of the message.

deleteVirtualDirectoryFile

Signature
```
deleteVirtualDirectoryFile(filename:         String;
                           deleteIfReadOnly: Boolean): Integer;
```

The **deleteVirtualDirectoryFile** method of the **JadeWebServiceProvider** class enables you to delete files that are in the directory specified by the **VirtualDirectory** parameter in the **jadehttp.ini** file.

The **deleteVirtualDirectoryFile** method parameters are listed in the following table.

Parameter	Description
filename	Name of the file to be deleted from the virtual directory
deleteIfReadOnly	Deletes files marked as read-only when set to **true**

This method returns zero (**0**) if the file deletion is successful or a non-zero error code if it fails.

You can specify whether files created in the virtual directory are deleted automatically and how this happens, by setting the **PurgeDirectoryRule** parameter in the [*application-name*] section of the **jadehttp.ini** file or the **PurgeDirectoryRule** configuration directive in the JADE **mod_jadehttp** file. If this parameter or directive is not set, files of type .**jpg**, .**png**, or .**gif** that are more than 12 hours old are removed. For details, see "Internal Housekeeping of the Virtual Directory", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Note This method must be called during the processing cycle of the message.

getLastStatistics

Signature
```
getLastStatistics(headerOnly: Boolean): String;
```

The **getLastStatistics** method of the **JadeWebServiceProvider** class returns an XML-formatted string that represents the information listed in the following table for the current request.

Statistic	Description
<queuedTime>	Time spent in the queue

Statistic	Description
<requestTime>	Time taken to process the request
<webServiceTime>	Time spent in the Web service method
<responseTime>	Time taken to generate the SOAP message and send the response
<requestSize>	Size of the request message
<responseSize>	Size of the response message

In addition, the information listed in the following table is returned when you set the **headerOnly** parameter to **false**.

Statistic	Description
<requestHeaders>	The HTTP headers that were received in the request
<soapRequest>	The SOAP message that was received
<soapResponse>	The SOAP message that was sent

To obtain all statistics for the request, you must call this method in your reimplemented **JadeWebServiceProvider** class **reply** method or in the destructor of the Web service.

Setting the value of the **headerOnly** parameter to **true** returns a string similar to the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<WebServiceStatistics>
    <queuedTime>5030</queuedTime>
    <requestTime>5</requestTime>
    <webServiceTime>4999</webServiceTime>
    <responseTime>4</responseTime>
    <requestSize>423</requestSize>
    <responseSize>387</responseSize>
</WebServiceStatistics>
```

Setting the value of the **headerOnly** parameter to **false** returns a string similar to the following example.

```
<?xml version="1.0" encoding="utf-8"?>
<WebServiceStatistics>
    <queuedTime>5016</queuedTime>
    <requestTime>5</requestTime>
    <webServiceTime>5000</webServiceTime>
    <responseTime>3</responseTime>
    <requestSize>423</requestSize>
    <responseSize>387</responseSize>
    <requestHeaders><![CDATA[Cache-Control: no-cache
Connection: Keep-Alive
Content-Length: 423
Content-Type: text/xml; charset=utf-8
Accept: text/plain, text/html, text/xml
Host: localhost
User-Agent: Jade/2016
SOAPAction: "urn:JadeWebServices/Mine/helloWorld"
]]></requestHeaders>
    <soapRequest><![CDATA[<?xml version="1.0" encoding="utf-8"?>
```

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="urn:JadeWebServices/Mine/" xmlns:s1="urn:JadeWebServices/Mine/">
    <soap:Body>
        <s1:helloWorld>
                <s1:user/>
                <s1:date>1900-01-01</s1:date>
        </s1:helloWorld>
    </soap:Body>
</soap:Envelope>
]]></soapRequest>
    <soapResponse><![CDATA[<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <helloWorldResponse xmlns="urn:JadeWebServices/Mine/">
            <helloWorldResult>true</helloWorldResult>
        </helloWorldResponse>
    </soap:Body>
</soap:Envelope>
]]></soapResponse>
</WebServiceStatistics>
```

getServerVariable

Signature `getServerVariable(var: String): String;`

The **getServerVariable** method of the **JadeWebServiceProvider** class returns the specified HTTP header information for your Web service request from the Internet Information Server (IIS).

As the **var** parameter is IIS-dependent and is therefore subject to change, refer to the **ServerVariables** function in your Internet Information Services (IIS) documentation for details.

The code fragment in the following example returns the IP address of the current Web service as determined by IIS.

```
JadeWebServiceProvider.getServerVariable('REMOTE_ADDR');
```

Common server environment variables, documented in the IIS documentation under the **ServerVariables** function, include those listed in the following table.

Variable	Returns…
HTTP_ACCEPT_LANGUAGE	A string describing the language to use for displaying content
HTTP_USER_AGENT	A string describing the browser that sent the request
HTTPS	**ON** if the request came in through a secure channel (SSL) or it returns **OFF** if the request is for a non-secure channel
REMOTE_ADDR	IP address of the remote host making the request
SERVER_NAME	Host name, DNS alias, or IP address of the server as it would appear in self-referencing URLs

Variable	Returns…
SERVER_PORT	Port number to which the request was sent
URL	Base portion of the URL

An exception is raised if this method is invoked from a server method.

The Web service provider requires the name of the method to invoke. In order to obtain this, the **getServerVariable** method is called. When the name that is retrieved is longer than 100 characters, the name is truncated to 100 characters. In addition, if the first character of the name is uppercase, it is changed to lowercase.

This name is used to determine the method to invoke when using non-wrapped document literal format messages. When the name does not meet the JADE method-naming requirements, the method invocation is likely to fail and a SOAP fault will be returned to the Web service consumer.

You can implement your own **getServerVariable** method (equivalent to this method in the **JadeWebServiceProvider** class) if you are using a **JadeInternetTCPIPConnection** instance to communicate with the **jadehttp** library (that is, **jadehttp.dll**) when your application does not use **WebSession** functionality.

The following method returns the value of the Internet Server Application Programming Interface (ISAPI) variable (specified by the **var** parameter) associated with an Internet message that is received.

```
getServerVariable(var: String): String;
// The request for the ISAPI variable var is built in the bin variable
// The JadeInternetTCPIPConnection instance must exist and be connected
constants
    NULL:       Character = #00.Character;
vars
    bin:        Binary;
    connection: JadeInternetTCPIPConnection;
begin
    if connection <> null and connection.state = Connection.Connected then
        if IsUnicodeSystem then
           bin := ("GSV" & NULL & var.trimBlanks &
                   NULL).Binary._unicodeToAnsi; // unpublished Binary method
        else
           bin := ("GSV" & NULL & var.trimBlanks & NULL).Binary;
        endif;
        connection.writeBinary(bin);
        bin := connection.readBinary(0);
    endif;
    if IsUnicodeSystem then
        return bin.ansiToUnicode.trimBlanks;
    else
        return bin.String.trimBlanks;
    endif;
end;
```

Caution You can call this method only during the processing of a received Internet message and before the reply is sent. Accessing the method at any other time causes the process to wait indefinitely for the connection read or causes the message exchange process with the **jadehttp** library to be out of step.

initialize

Signature `initialize(appName: String): Boolean updating;`

When the default HTTP implementation is not used, no Web service application is run. However, you must still define a Web service application and set up the appropriate options.

In your application code, create an instance of your **JadeWebServiceProvider** subclass and then set up that instance with your options, by calling the **JadeWebServiceProvider** class **initialize** method, specifying the name of your Web-enabled application in the **appName** parameter. The Web service options are then set up.

If the application specified in the **appName** parameter does not exist or it is not a Web service application, this method returns **false**.

For more details and an example of this use of this method, see "Using Communications Protocols Other than HTTP in your Web Service", in Chapter 11 of the *JADE Developer's Reference*.

isVDFilePresent

Signature `isVDFilePresent(fileName: String): Boolean;`

The **isVDFilePresent** method of the **JadeWebServiceProvider** class determines whether the file specified in the **fileName** parameter is present in the directory specified by the **VirtualDirectory** parameter in the **JadeHttp.ini** file.

The method returns **true** if the specified file exists or it returns **false** if it does not exist.

Note This method must be called during the processing cycle of the message.

processMessage

Signature `processMessage(message: String): String updating;`

The **JadeWebServiceProvider** class **processMessage** method takes the value of the **message** parameter as input (which is assumed to be a SOAP message), calls the relevant Web service method, passing it the necessary parameters, and returns the result of the processing as a SOAP message.

If the incoming message is not a SOAP message, an exception is raised. Similarly, if the method name or the parameters are not valid, an exception is raised. It is your responsibility to trap this exception and take whatever action is necessary. Use the **Exception** class **createSOAPMessage** method to transform this error into a SOAP message.

For more details and an example of this use of this method, see "Using Communications Protocols Other than HTTP in your Web Service", in Chapter 11 of the *JADE Developer's Reference*.

processRequest

Signature `processRequest() protected, updating;`

When a request is received from the Web service consumer, a transient instance of the class corresponding to this request is created and the **processRequest** method of the **JadeWebServiceProvider** class is called. This method identifies the Web service method in your user-defined **JadeWebServiceProvider** subclass to be invoked for the request but does not invoke the method.

Reimplement this method in your user-defined **JadeWebServiceProvider** subclasses if you want to process Web requests and send a reply back to the Web service consumer after all processing is complete.

Note If the JADE implementation of this method is not called (by using the **inheritMethod** instruction), it is your responsibility to do any processing that is necessary and to send a reply back to the consumer. For details, see the **JadeWebServiceProvider** class **processRequestPostHeaders** and **reply** methods.

processRequestPostHeaders

Signature `processRequestPostHeaders();`

The **JadeWebServiceProvider** class **processRequestPostHeaders** method enables you to process a request from the Web service consumer after the **processRequest** method has processed the SOAP headers for the request but before it has processed the body of the request.

If you reimplement the **processRequest** method by first calling the **inheritMethod** instruction, the SOAP headers and the body of the request have both been processed.

reply

Signature `reply(): String protected, updating;`

The **reply** method of the **JadeWebServiceProvider** class invokes the Web service method in your user-defined **JadeWebServiceProvider** subclass and builds the response string that is returned to the Web service consumer.

Reimplement in your user-defined Web service provider subclasses.

The following example manipulates the string before sending a reply back to the Web service consumer.

```
reply(): String updating, protected;
vars
    response : String;
begin
    response := inheritMethod();
    // invokes your Web service method and builds a response string
    // manipulate the response
    return response;
end;
```

Note If the JADE implementation of this method is not called (by using the **inheritMethod** instruction), it is your responsibility to send a response back to the Web service consumer.

JadeWebServiceSoapHeader Class

The **JadeWebServiceSoapHeader** class defines the behavior of SOAP headers in Web service provider applications. For details about the properties defined in the **JadeWebServiceSoapHeader** class, see "JadeWebServiceSoapHeader Properties", in the following subsection.

Inherits From: JadeWebService

Inherited By: (None)

JadeWebServiceSoapHeader Properties

The properties defined in the **JadeWebServiceSoapHeader** class are summarized in the following table.

Property	Descriptions
actor	Contains the URL of the SOAP header recipient
didUnderstand	Specifies whether a mandatory SOAP header has been understood as part of processing the request
mustUnderstand	Specifies whether it is mandatory for the recipient of the SOAP header to process the header

actor

Type: String

The **actor** property of the **JadeWebServiceSoapHeader** class contains the URL of the SOAP header recipient. Use this property when you do not intend all parts of a SOAP message to be sent to the ultimate destination of the SOAP message but to one or more intermediaries on the message path.

Only the consumer of a SOAP header can receive the SOAP header; that is, a consumer of a SOAP header cannot forward the header to the next application in the SOAP message path. The consumer can insert a similar header, which can be provided to another consumer of the SOAP header.

The value of the SOAP header actor is a Uniform Resource Identifier (URI). The special URI http://schemas.xmlsoap.org/soap/actor/next indicates that the SOAP header is intended for the first SOAP application to process the message.

If you do not specify an actor of the SOAP header, the recipient (consumer) of the Web service provider is the ultimate destination of the SOAP message.

didUnderstand

Type: Boolean

The **didUnderstand** property of the **JadeWebServiceSoapHeader** class specifies whether a mandatory SOAP header has been understood as part of processing the request. A mandatory SOAP request has the value of the **mustUnderstand** property set to **true**.

If the header is understood, your code should set the value of the **didUnderstand** property to **true**. If the header is not understood, a SOAP *Must Understand* error is returned as the response.

mustUnderstand

Type: Boolean

The **mustUnderstand** property of the **JadeWebServiceSoapHeader** class specifies whether the SOAP header is mandatory for the Web service consumer recipients to process.

As the value of this property is **false** by default, the SOAP header is optional.

When you set this property to **true** at run time to make the SOAP header mandatory, the consumer of the header entry must obey the semantics conveyed by the fully qualified name of the SOAP header and process correctly to those semantics, or it must fail processing the message.

As SOAP headers that must be understood can modify the semantics of their parent or peer headers, those who do not fully understand them cannot ignore the semantics.

JadeWebServiceUnknownHeader Class

The **JadeWebServiceUnknownHeader** class encapsulates an unknown SOAP header in a Web service provider application; that is, a SOAP header included with the request that the provider was unable to process.

For details about the properties defined in the **JadeWebServiceUnknownHeader** class, see "JadeWebServiceUnknownHeader Properties", in the following subsection.

Inherits From: JadeWebService

Inherited By: (None)

JadeWebServiceUnknownHeader Properties

The properties defined in the **JadeWebServiceUnknownHeader** class are summarized in the following table.

Property	Contains …
headerXML	The XML content of the unknown SOAP header
webService	A reference to the **JadeWebService** that contained the unknown SOAP header

headerXML

Type: String

The **headerXML** property of the **JadeWebServiceUnknownHeader** class contains the XML content of the unknown SOAP header.

webService

Type: JadeWebService

The **webService** property of the **JadeWebServiceUnknownHeader** class specifies the **JadeWebService** for which the unknown SOAP header could not be processed.

JadeWebSocket Class

The **JadeWebSocket** class is the base class for handling a WebSocket connection.

Create your own **JadeWebSocket** subclasses to define the required WebSocket behavior for your application. From the perspective of a JADE developer, Web Socket processing:

1. Starts with a call to your ***JadeWebSocket-subclass*.onOpen** method
2. Is followed by zero or more calls to your ***JadeWebSocket-subclass*.onMsg** method
3. Ends with a call to your ***JadeWebSocket-subclass*.onClose** method

For details about the property and methods defined in the **JadeWebSocket** class, see "JadeWebSocket Property" and "JadeWebSocket Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeWebSocket Property

The property defined in the **JadeWebSocket** class is summarized in the following table.

Property	Description
id	WebSocket identifier

id

Type: Integer64

The read-only **id** property of the **JadeWebSocket** class contains the identifier of the WebSocket connection.

JadeWebSocket Methods

The methods defined in the **JadeWebSocket** class are summarized in the following table.

Method	Description
onClose	Called when the WebSocket is closed
onMsg	Called when the WebSocket receives a message from the client side of the connection
onOpen	Called when the WebSocket is opened from the client side of the connection
send	Sends a binary message via the WebSocket connection
sendText	Sends a UTF8-encoded message via the WebSocket connection

onClose

Signature `onClose() protected;`

The **onClose** method of the **JadeWebSocket** class is called when the WebSocket is closed.

onMsg

Signature
```
onMsg(msg:           Binary;
      utf8encoded:   Boolean;
      finalFragment: Boolean) updating, protected;
```

The **onMsg** method of the **JadeWebSocket** class is called when the WebSocket receives a message from the client side of the connection. The **msg** parameter contains the original WebSocket message sent by the client.

The value of the **utf8encoded** parameter is **true** if the message contains Unicode text in a UTF8-encoded format; otherwise the value is **false**.

The value of the **finalFragment** parameter is **true** if the message is the final data fragment sent by the client; otherwise the value is **false**.

onOpen

Signature
```
onOpen(fullUrl: String) protected;
```

The **onOpen** method of the **JadeWebSocket** class is called when the WebSocket is opened from the client side of the connection.

The **fullUrl** parameter specifies the full URL of the WebSocket (for example, **http://host.domain.co.nz:80/WebSocketBrowserClientTest/websocket.ws**) used by the WebSocket initiating client (for example, the web browser).

send

Signature
```
send(msg:           Binary
     finalFragment: Boolean);
```

The **send** method of the **JadeWebSocket** class sends the binary message specified in the **msg** parameter to the client via the WebSocket connection.

The value of the **finalFragment** parameter is **true** if the message is the final data fragment to send; otherwise the value is **false**.

sendText

Signature
```
sendText(msg:           utf8;
         finalFragment: Boolean);
```

The **sendText** method of the **JadeWebSocket** class sends the Unicode text in a UTF8-encoded format specified in the **msg** parameter to the client via the WebSocket connection.

The value of the **finalFragment** parameter is **true** if the message is the final data fragment to send; otherwise the value is **false**.

JadeWebSocketServer Class

The **JadeWebSocketServer** class handles all incoming TCP/IP connections from the **JadeWebSocketIISNativeModule** on a specific interface and TCP port.

For details about the methods defined in the **JadeWebSocketServer** class, see "JadeWebSocketServer Methods", in the following subsection.

Inherits From: Object

Inherited By: (None)

JadeWebSocketServer Methods

The methods defined in the **JadeWebSocketServer** class are summarized in the following table.

Method	Description
getWebSocket	Retrieves the instance of a subclass of **JadeWebSocket** identified by its web socket id.
run	Starts a WebSocket server, accepting TCP connections from IIS
stop	Signals an active **run** method to stop listening on the local interface and TCP/IP port, and terminates all WebSocket connections

getWebSocket

Signature `getWebSocket(wsId: Integer64): JadeWebSocket;`

The **getWebSocket** method of the **JadeWebSocketServer** class retrieves the instance of a subclass of **JadeWebSocket** identified by its web socket id. The web socket id is stored in the **id** property of the associated **JadeWebSocket** object.

This method returns null if the **JadeWebSocket** does not exist.

Note The WebSockets are automatically deleted when the TCP/IP connection is closed.

run

Signature
```
run(localInterface: String;
    tcpPort:        Integer;
    webSocketClass: Class) updating;
```

The **run** method of the **JadeWebSocketServer** class starts listening on the local interface and TCP/IP port specified in the respective **localInterface** and **tcpPort** parameters for connections from WebSocket clients and creates an instance of **WebSocketClass** for every new WebSocket connection.

You can use the value of the **localInterface** parameter to limit the interfaces on which incoming connections are accepted. To accept connections on all interfaces, the **localInterface** parameter must have a null (**""**) value.

The **tcpPort** parameter specifies the TCP port on which the server listens for incoming connections.

The value of the **webSocketClass** parameter specifies the user subclass of **JadeWebSocket**.

stop

Signature `stop() updating;`

The **stop** method of the **JadeWebSocketServer** class signals an active **run** method to stop listening on the local interface and TCP/IP port, and terminates all WebSocket connections.

JadeX509Certificate Class

The transient **JadeX509Certificate** class stores the digital certificates in X509 format, which are written to disk in Privacy-Enhanced Electronic Mail (PEM)-encoded certificate (PEM) format, for use with the **JadeSSLContext** class that provides secure connections when the **TcpIpConnection** class **sslContext** property contains a reference to a **JadeSSLContext** transient object.

JadeSSLContext connections use digital certificates in X509 format. For details about the properties and methods defined in the **JadeX509Certificate** class, see "JadeX509Certificate Properties" and "JadeX509Certificate Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeX509Certificate Properties

The properties defined in the **JadeX509Certificate** class are summarized in the following table.

Property	Contains …
endDate	The expiry date of the certificate
issuer	The issuer string from the certificate in readable format
purpose	A comma-separated list of the purposes of the certificate
startDate	The first valid date of the certificate
subject	The subject string of the certificate in readable format

endDate

Type: TimeStamp

The read-only **endDate** property of the **JadeX509Certificate** class contains the timestamp of the expiry date of the certificate.

issuer

Type: String

The read-only **issuer** property of the **JadeX509Certificate** class contains the issuer string from the certificate in readable format.

purpose

Type: String

The read-only **purpose** property of the **JadeX509Certificate** class contains a comma-separated list of the purposes indicated by the certificate.

startDate

Type: TimeStamp

The read-only **startDate** property of the **JadeX509Certificate** class contains the timestamp of the first valid date of the certificate.

subject

Type: JadeSSLContext

The read-only **subject** property of the **JadeX509Certificate** class contains the subject string of the certificate in readable format.

JadeX509Certificate Methods

The methods defined in the **JadeX509Certificate** class are summarized in the following table.

Method	Description
readCertificateDataFromFile	Reads certificate data from the specified file
readPrivateKeyDataFromFile	Reads private data from the specified file using the specified password

readCertificateDataFromFile

Signature `readCertificateDataFromFile(fileName: String) updating;`

The **readCertificateDataFromFile** method of the **JadeX509Certificate** class reads data from the certificate specified in the **fileName** parameter, as shown in the following example.

```
vars
    x509 : JadeX509Certificate;
begin
    create x509 transient;
    x509.readCertificateDataFromFile("c:\certificates\mycert.pem");
epilog
    delete x509;
end;
```

readPrivateKeyDataFromFile

Signature
```
readPrivateKeyDataFromFile(fileName: String;
                           password: String) updating;
```

The **readPrivateKeyDataFromFile** method of the **JadeX509Certificate** class reads private key data from the file specified in the **fileName** parameter, using the password specified in the **password** parameter.

The following example shows the use of the **readPrivateKeyDataFromFile** method.

```
vars
    x509 : JadeX509Certificate;
begin
    create x509 transient;
    x509.readPrivateKeyDataFromFile("c:\certificates\myprivate.pem",
                                    "zz99zz99demo");
```

```
epilog
    delete x509;
end;
```

JadeXMLAttribute Class

The **JadeXMLAttribute** class defines the behavior for attributes of XML elements in a document tree. An attribute has a name, an optional namespace, and a value.

For details about the properties and method defined in the **JadeXMLAttribute** class, see "JadeXMLAttribute Properties" and "JadeXMLAttribute Method", in the following sections.

Inherits From: JadeXMLNode

Inherited By: (None)

JadeXMLAttribute Properties

The properties defined in the **JadeXMLAttribute** class are summarized in the following table.

Property	Contains the …
element	Owning element of the attribute
localName	Local name (without prefix) of the attribute
name	Qualified name (with prefix) of the attribute
namespaceURI	Namespace URI of the attribute
value	Value of the attribute

element

Type: JadeXMLElement

The read-only **element** property of the **JadeXMLAttribute** class contains a reference to the owning element of the attribute.

localName

Type: String

The **localName** property of the **JadeXMLAttribute** class contains the local name (without a prefix) of the attribute.

name

Type: String

The **name** property of the **JadeXMLAttribute** class contains the qualified name (with prefix) of the attribute.

namespaceURI

Type: String

The **namespaceURI** property of the **JadeXMLAttribute** class contains the namespace Uniform Resource Identifier (URI) of the attribute or it contains null (**""**) if the attribute has no namespace URI.

value

Type: String

The **value** property of the **JadeXMLAttribute** class contains the value of the attribute.

JadeXMLAttribute Method

The method defined in the **JadeXMLAttribute** class is summarized in the following table.

Method	Description
namespacePrefix	Returns the namespace prefix

namespacePrefix

Signature `namespacePrefix(): String;`

The **namespacePrefix** method of the **JadeXMLAttribute** class returns a string containing the namespace prefix of the attribute or it returns null (**""**) if the namespace prefix is unspecified.

JadeXMLCDATA Class

The **JadeXMLCDATA** class represents a **CDATA** section in an XML document tree.

CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup.

Note If you do not want the framework to interpret the XML special characters (that is, **<**, **>**, **&**, and **"**) for a string, call the **String** primitive type **makeXMLCData** method, which returns a new string of the receiver prepended with **<![CDATA[**and appended with **]]>**.

Inherits From: JadeXMLCharacterData

Inherited By: (None)

JadeXMLCharacterData Class

The **JadeXMLCharacterData** class is the abstract superclass of character-based nodes in an XML document tree; that is, the text, **CDATA**, and comment nodes.

For details about the property defined in the **JadeXMLCharacterData** class, see "JadeXMLCharacterData Property", in the following section.

Inherits From: JadeXMLNode

Inherited By: JadeXMLCDATA, JadeXMLComment, JadeXMLText

JadeXMLCharacterData Property

The property defined in the **JadeXMLCharacterData** class is summarized in the following table.

Property	Contains the …
data	Text value of the node

data

Type: String

The **data** property of the **JadeXMLCharacterData** class contains the text value of the node.

JadeXMLComment Class

The JadeXMLComment class represents the content of a comment in an XML document; that is, all of the characters between the starting '**<!--**' and the ending '**-->**'.

Inherits From: JadeXMLCharacterData

Inherited By: (None)

JadeXMLDocument Class

The **JadeXMLDocument** class represents an XML document as a tree of nodes. It defines the owning object of all objects in the tree.

Note As the **getElementByTagName**, **getElementByTagNameNS**, **getElementsByTagName**, and **getElementsByTagNameNS** methods scan sequentially to locate requested elements, they always returned requested elements in document sequence. To improve performance, you can use the **findElementByNameNS**, **findElementByTagName**, **findElementsByNameNS**, and **findElementsByTagName** methods to retrieve elements more directly through a collection, using the collection sequence. JADE fully supports the use of a mixture of the document and collection sequence methods to locate the requested elements. The *find* methods may locate the elements in a different sequence from the *get* methods.

The collection sequence methods provide a performance boost only if a **localName** or **tagName** parameter value is explicitly specified in the calling parameters. If you specify **"*"** in the **localName** or **tagName** parameter, the access method reverts to the functionality and performance of the document sequence methods to locate the requested elements.

For details about the properties and methods defined in the **JadeXMLDocument** class, see "JadeXMLDocument Properties" and "JadeXMLDocument Methods", in the following sections.

Inherits From: JadeXMLNode

Inherited By: (None)

JadeXMLDocument Properties

The properties defined in the **JadeXMLDocument** class are summarized in the following table.

Property	Description
docType	Document type of the document
endOfLine	End-of-line separator for output
indentString	Indentation string for output
keepWhitespace	Specifies whether extra whitespace is discarded
outputDeclaration	Specifies whether the XML declaration is output
rootElement	Root element of the document

docType

Type: JadeXMLDocumentType

The read-only **docType** property of the **JadeXMLDocument** class contains a reference to the document type of the XML document. (See also the **JadeXMLDocumentType** class.) If the document has no specified document type, this property contains a null value.

endOfLine

Type: String

The **endOfLine** property of the **JadeXMLDocument** class contains the end-of-line separator that is used to delimit lines when writing XML documents. By default, this property contains carriage return and line feed (**Cr/Lf**) characters.

indentString

Type: String

The **indentString** property of the **JadeXMLDocument** class contains the indentation string used to indent each level of the tree when writing XML documents. The default value is two spaces. If you do not want indentation to occur, set this property to null (**""**).

keepWhitespace

Type: Boolean

The **keepWhitespace** property of the **JadeXMLDocument** class specifies whether extra whitespace between adjacent tags is discarded during parsing in XML documents. As JADE assumes that the XML document contains data, this property is set to **false** by default. This optimization improves parsing performance and reduces the size of the object tree because the extra whitespace does not need to be stored as text nodes.

If the extra whitespace is significant and this property is set to **true**, you should set the **indentString** property to null (**""**) to turn off the automatic indentation when writing the document.

outputDeclaration

Type: Boolean

The **outputDeclaration** property of the **JadeXMLDocument** class specifies whether the XML declaration (that is, **<?xml version="1.0"?>**) is output when writing XML documents.

As the default value is **true**, set this property to **false** if you do *not* want the XML declaration output.

rootElement

Type: JadeXMLElement

The read-only **rootElement** property of the **JadeXMLDocument** class contains a reference to the root element of the XML document.

The root element is the top-level element in the document, and all the other elements are its children. An XML document must have one root element only.

JadeXMLDocument Methods

The methods defined in the **JadeXMLDocument** class are summarized in the following table.

Method	Description
addComment	Creates and adds a comment

Method	Description
addCommentObject	Adds a comment object
addDocumentType	Creates and adds a document type
addDocumentTypeObject	Adds a document type object
addElement	Creates and adds an element
addElementNS	Creates and adds an element with a namespace
addElementObject	Adds an element object
addElementObjectNS	Adds an element object with a namespace
addProcessingInstruction	Creates and adds a processing instruction
addProcessingInstructionObject	Adds a processing instruction object
findElementByNameNS	Returns an element with the specified namespace URI and local name
findElementByTagName	Returns an element with the specified tag name
findElementsByNameNS	Fills an array with all elements in the document with the specified namespace URI and local name
findElementsByTagName	Fills an array with all elements in the document with the specified tag name
getElementByTagName	Returns the first element with the specified tag name
getElementByTagNameNS	Returns the first element with the specified namespace URI and local name
getElementsByTagName	Fills an array with all elements in the document with the specified tag name
getElementsByTagNameNS	Fills an array with all elements in the document with the specified namespace URI and local name
parseFile	Parses an XML document file
parseString	Parses an XML document string
writeToFile	Writes the XML representation of the document to a file

addComment

Signature `addComment(text: String): JadeXMLComment updating;`

The **addComment** method of the **JadeXMLDocument** class creates and adds the comment node specified in the **text** parameter and returns a reference to the created **JadeXMLComment** node instance.

addCommentObject

Signature
```
addCommentObject(comment: JadeXMLComment;
                 text: String): JadeXMLComment updating;
```

The **addCommentObject** method of the **JadeXMLDocument** class adds a comment object with the text specified in the **text** parameter and returns a reference to the added **JadeXMLComment** object instance.

addDocumentType

Signature
```
addDocumentType(name:          String;
                publicId:      String;
                systemId:      String;
                internalSubset: String): JadeXMLDocumentType updating;
```

The **addDocumentType** method of the **JadeXMLDocument** class creates and adds a document type node with the specified parameter values and returns a reference to the created **JadeXMLDocumentType** node instance.

The **addDocumentType** method parameters are listed in the following table.

Parameter	Description
name	Name of the Document Type Definition (DTD)
publicId	Public identifier of the external subset
systemId	System identifier of the external subset
internalSubset	Internal subset

addDocumentTypeObject

```
addDocumentTypeObject(documentType: JadeXMLDocumentType;
                      name:         String;
                      publicId:     String;
                      systemId:     String;
                      internalSubset: String): JadeXMLDocumentType updating;
```

The **addDocumentTypeObject** method of the **JadeXMLDocument** class adds a document type object with the specified parameter values and returns a reference to the added **JadeXMLDocumentType** object instance.

The **addDocumentTypeObject** method parameters are listed in the following table.

Parameter	Description
name	Name of the Document Type Definition (DTD)
publicId	Public identifier of the external subset
systemId	System identifier of the external subset
internalSubset	Internal subset

addElement

Signature
```
addElement(tagName: String): JadeXMLElement updating;
```

The **addElement** method of the **JadeXMLDocument** class creates and adds an element node with the tag name specified in the **tagName** parameter and returns a reference to the created **JadeXMLElement** node instance.

addElementNS

Signature
```
addElementNS(namespaceURI:  String;
             qualifiedName: String): JadeXMLElement updating;
```

The **addElementNS** method of the **JadeXMLDocument** class creates and adds an element node with the namespace and qualified name specified in the **namespaceURI** and **qualifiedName** parameters, respectively, and returns a reference to the created **JadeXMLElement** node instance.

addElementObject

Signature
```
addElementObject(element: JadeXMLElement;
                 tagName: String): JadeXMLElement updating;
```

The **addElementObject** method of the **JadeXMLDocument** class adds an element object with the tag name specified in the **tagName** parameter and returns a reference to the added **JadeXMLElement** object instance.

addElementObjectNS

Signature
```
addElementObjectNS(element:       JadeXMLElement;
                   namespaceURI:  String;
                   qualifiedName: String): JadeXMLElement updating;
```

The **addElementObjectNS** method of the **JadeXMLDocument** class adds an element object with the namespace and qualified name specified in the **namespaceURI** and **qualifiedName** parameters, respectively, and returns a reference to the added **JadeXMLElement** object instance.

addProcessingInstruction

Signature
```
addProcessingInstruction(target: String;
                         data:  String): JadeXMLProcessingInstruction updating;
```

The **addProcessingInstruction** method of the **JadeXMLDocument** class creates and adds a processing instruction node with the values specified in the **target** and **data** parameters, and returns a reference to the created **JadeXMLProcessingInstruction** node instance.

addProcessingInstructionObject

Signature
```
addProcessingInstructionObject(procInstr: JadeXMLProcessingInstruction;
                               target:    String;
                               data: String): JadeXMLProcessingInstruction updating;
```

The **addProcessingInstructionObject** method of the **JadeXMLDocument** class adds a processing instruction object with the values specified in the **target** and **data** parameters, and returns a reference to the added **JadeXMLProcessingInstruction** object instance.

findElementByNameNS

Signature
```
findElementByNameNS(namespaceURI: String;
                    localName:    String): JadeXMLElement;
```

The **findElementByNameNS** method of the **JadeXMLDocument** class returns a reference to a **JadeXMLElement** instance that has the namespace URI and local name specified in the **namespaceURI** and **localName** parameters, respectively.

Note As the search uses the collection sequence, the located element may not be the first element with the matching namespace URI and local name in the document sequence.

If you want to match any namespace URIs or any local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter. Note, however, that if you specify **"*"** in the **localName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

findElementByTagName

Signature `findElementByTagName(tagName: String): JadeXMLElement;`

The **findElementByTagName** method of the **JadeXMLDocument** class returns a reference to a **JadeXMLElement** instance that has the tag name specified in the **tagName** parameter.

Note As the search uses the collection sequence, the located element may not be the first element with the matching tag name in the document sequence.

If you want to match any tag names, specify an asterisk character (**'*'**) in the **tagName** parameter. Note, however, that if you specify **"*"** in the **tagName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

findElementsByNameNS

Signature
```
findElementsByNameNS(namespaceURI:  String;
                     localName:     String;
                     elements:      JadeXMLElementArray input);
```

The **findElementsByNameNS** method of the **JadeXMLDocument** class fills the elements array with all elements that have the values specified in the **namespaceURI** and **localName** parameters, respectively.

Note As the search uses the collection sequence, the elements may not be in the document sequence.

If you want to match all namespace URIs or all local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter. Note, however, that if you specify **"*"** in the **localName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

findElementsByTagName

Signature
```
findElementsByTagName(tagName:  String;
                      elements: JadeXMLElementArray input);
```

The **findElementsByTagName** method of the **JadeXMLDocument** class fills the elements array with all elements that have the value specified in the **tagName** parameter.

Note As the search uses the collection sequence, the elements may not be in the document sequence.

If you want to match all tag names, specify an asterisk character (**'*'**) in the **tagName** parameter. Note, however, that if you specify **"*"** in the **tagName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

getElementByTagName

Signature `getElementByTagName(tagName: String): JadeXMLElement;`

The **getElementByTagName** method of the **JadeXMLDocument** class returns a reference to the first **JadeXMLElement** instance that has the tag name specified in the **tagName** parameter.

Tip To improve performance when the sequence is not important, use the **findElementByTagName** method to retrieve the element more directly through a collection, by using the collection sequence.

getElementByTagNameNS

Signature
```
getElementByTagNameNS(namespaceURI:  String;
                      localName:     String): JadeXMLElement;
```

The **getElementByTagNameNS** method of the **JadeXMLDocument** class returns a reference to the first **JadeXMLElement** instance that has the namespace URI and local name specified in the **namespaceURI** and **localName** parameters, respectively.

If you want to match all namespace URIs or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter.

Tip To improve performance when the sequence is not important, use the **findElementByNameNS** method to retrieve the element more directly through a collection, by using the collection sequence.

getElementsByTagName

Signature
```
getElementsByTagName(tagName:  String;
                     elements: JadeXMLElementArray input);
```

The **getElementsByTagName** method of the **JadeXMLDocument** class fills the elements array with all elements in document order (that is, using a preorder traversal) that have the value specified in the **tagName** parameter.

If you want to match all tags, specify an asterisk character (**'*'**) in the **tagName** parameter.

Tip To improve performance when the sequence is not important, use the **findElementsByTagName** method to retrieve elements more directly through a collection, by using the collection sequence.

getElementsByTagNameNS

Signature
```
getElementsByTagNameNS(namespaceURI:  String;
                       localName:     String;
                       elements:      JadeXMLElementArray input);
```

The **getElementsByTagNameNS** method of the **JadeXMLDocument** class fills the elements array with all elements in document order (that is, using a preorder traversal) that have the values specified in the **namespaceURI** and **localName** parameters, respectively.

If you want to match all namespace URIs or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter.

Tip To improve performance when the sequence is not important, use the **findElementsByNameNS** method to retrieve elements more directly through a collection, by using the collection sequence.

parseFile

Signature `parseFile(fileName: String) updating;`

The **parseFile** method of the **JadeXMLDocument** class parses the XML document file specified in the **fileName** parameter and creates a tree of nodes representing the document. (See also the **JadeXMLDocumentParser** class **parseDocumentFile** method.)

Any existing child nodes in the document object are removed before the tree is created.

parseString

Signature `parseString(inputDocument: String) updating;`

The **parseString** method of the **JadeXMLDocument** class parses the XML document string specified in the **inputDocument** parameter and creates a tree of nodes representing the document. (See also the **JadeXMLDocumentParser** class **parseDocumentString** method.)

The following example shows the use of the **parseString** method.

```
vars
    doc : JadeXMLDocument;
begin
    create doc;
    doc.parseString('<employee>John Smith</employee>');
end;
```

Any existing child nodes in the document object are removed before the tree is created.

writeToFile

Signature `writeToFile(fileName: String);`

The **writeToFile** method of the **JadeXMLDocument** class writes the XML representation of the document to the file specified in the **fileName** parameter.

You can control the format of the output by setting the **JadeXMLDocument** class formatting properties of the document object; for example, the **endOfLine** and **indentString** properties.

JadeXMLDocumentParser Class

The **JadeXMLDocumentParser** class is the transient class that represents the interface for parsing XML documents into a tree of objects.

The parser reads an XML document and creates a tree of object nodes that are instances of the **JadeXMLNode** classes or user subclasses.

The **JadeXMLDocumentParser** class provides a more-flexible method of parsing document trees compared to the **parseFile** and **parseString** methods of the **JadeXMLDocument** class. Because you can set up a mapping of user subclasses to node classes, the **JadeXMLDocumentParser** class enables you to parse documents into persistent trees that reside in user-specified map files.

For details about the methods defined in the **JadeXMLDocumentParser** class, see "JadeXMLDocumentParser Methods", in the following section.

Inherits From: JadeXMLParser

Inherited By: (None)

JadeXMLDocumentParser Methods

The methods defined in the **JadeXMLDocumentParser** class are summarized in the following table.

Method	Description
comment	Receives notification of a comment
parseDocumentFile	Parses an XML document file
parseDocumentString	Parses an XML document string
processingInstruction	Receives notification of a processing instruction
setClassMapping	Sets the mapping of a **JadeXMLNode** class to a user subclass
startCDATA	Receives notification of the start of DTD declarations

comment

Signature `comment(text: String) updating, protected;`

The **comment** event method of the **JadeXMLDocumentParser** class receives notification of an XML comment specified in the characters in the **text** parameter.

The parser calls this event method (if implemented) to report comments anywhere in the XML document (that is, inside or outside the root element).

parseDocumentFile

Signature
```
parseDocumentFile(doc:      JadeXMLDocument input;
                  fileName: String) updating;
```

The **parseDocumentFile** method of the **JadeXMLDocumentParser** class parses the XML document file specified in the **fileName** parameter and creates a tree of nodes in the document object specified in the **doc** parameter.

Any existing child nodes in the document object are removed before the tree is created.

parseDocumentString

Signature `parseDocumentString(doc: JadeXMLDocument input;
str: String) updating;`

The **parseDocumentString** method of the **JadeXMLDocumentParser** class parses the XML document string specified in the **str** parameter and creates a tree of nodes in the document object specified in the **doc** parameter.

Any existing child nodes in the document object are removed before the tree is created.

processingInstruction

Signature `processingInstruction(target: String;
data: String) updating, protected;`

The **processingInstruction** event method of the **JadeXMLDocumentParser** class receives notification of a processing instruction.

The **processingInstruction** event method parameters are listed in the following table.

Parameter	Description
target	The processing instruction target.
data	The processing instruction data or null (**""**) if none was supplied. The data does not include any whitespace separating it from the target.

The parser invokes this event method (if implemented) for each processing instruction that it locates.

Note Processing instructions can occur before or after the root element.

setClassMapping

Signature `setClassMapping(nodeClass: Class;
mappedClass: Class) updating;`

The **setClassMapping** method of the **JadeXMLDocumentParser** class enables you to specify the class of tree instances created during parsing; for example:

```
setClassMapping(JadeXMLElement, MyElement);
```

This method sets the mapping of the **JadeXMLNode** class specified in the **nodeClass** parameter to the user-specified class in the **mappedClass** parameter.

When creating persistent documents, an exception is raised if all concrete **JadeXMLNode** classes are not mapped to user subclasses.

startCDATA

Signature `startCDATA() updating, protected;`

The **startCDATA** event method of the **JadeXMLDocumentParser** class receives notification of the start of a CDATA section.

The contents of the CDATA section are reported through the regular **characters** event method.

JadeXMLDocumentType Class

The **JadeXMLDocumentType** class represents the document type declaration in an XML document tree. A reference to the document type (if the document has one) is stored in the **docType** property of the **JadeXMLDocument** class.

For details about the properties defined in the **JadeXMLDocumentType** class, see "JadeXMLDocumentType Properties", in the following section.

Inherits From: JadeXMLNode

Inherited By: (None)

JadeXMLDocumentType Properties

The properties defined in the **JadeXMLDocumentType** class are summarized in the following table.

Property	Contains the …
internalSubset	Internal subset
name	Name of the Document Type Definition (DTD)
publicId	Public identifier of the external subset
systemId	System identifier of the external subset

internalSubset

Type: String

The **internalSubset** property of the **JadeXMLDocumentType** class contains the internal subset of the document type, or null (**""**) if there is no internal subset. When building a document using the parser, this property is set to null (**""**).

name

Type: String

The **name** property of the **JadeXMLDocumentType** class contains the name of the Document Type Definition (DTD), which is the name immediately following the **DOCTYPE** keyword in the XML document.

publicId

Type: String

The **publicId** property of the **JadeXMLDocumentType** class contains the public identifier of the external subset.

systemId

Type: String

The **systemId** property of the **JadeXMLDocumentType** class contains the system identifier of the external subset.

JadeXMLElement Class

The **JadeXMLElement** class represents an XML element in a document tree. An element can have attributes, child nodes, and textual content. For an example of using methods defined in the **JadeXMLElement** class to create an XML document, see "Creating XML Tree Documents", in Chapter 12 of the *JADE Developer's Reference*.

Note As the **getAllElementsByTagNameNS** and **getAllElementsByTagName** methods scan sequentially to locate all requested elements, they always return requested elements in document sequence. To improve performance, you can use the **findAllElementsByNameNS** and **findAllElementsByTagName** methods to retrieve elements more directly through a collection, using the collection sequence. JADE fully supports the use of a mixture of the document and collection sequence methods to locate the requested elements. The *find* methods may locate the elements in a different sequence from the *get* methods.

The collection sequence methods provide a performance boost only if a **localName** or **tagName** parameter value is explicitly specified in the calling parameters. If you specify **"*"** in the **localName** or **tagName** parameter, the access method reverts to the functionality and performance of the document sequence methods to locate the requested elements.

For details about the properties and methods defined in the **JadeXMLElement** class, see "JadeXMLElement Properties" and "JadeXMLElement Methods", in the following sections.

Inherits From: JadeXMLNode

Inherited By: (None)

JadeXMLElement Properties

The properties defined in the **JadeXMLElement** class are summarized in the following table.

Property	Contains the …
attributes	Array of attributes of the element
localName	Local name (without the prefix) of the element
namespaceURI	Namespace URI of the element
tagName	Qualified name (with the prefix) of the element
textData	Text data of a text-only element

attributes

Type: JadeXMLAttributeArray

The read-only **attributes** property of the **JadeXMLElement** class contains a reference to an array of attributes of the element.

localName

Type: String

The **localName** property of the **JadeXMLElement** class contains the local name (without a prefix) of the element.

namespaceURI

Type: String

The **namespaceURI** property of the **JadeXMLElement** class contains the namespace Uniform Resource Identifier (URI) of the element or it contains null (**""**) if the element has no namespace URI.

tagName

Type: String

The **tagName** property of the **JadeXMLElement** class contains the qualified name (with the prefix) of the element.

textData

Type: String

The read-only **textData** property of the **JadeXMLElement** class contains the text data of a text-only element.

Note As an optimization, an element that contains only a single block of text can have its text content stored in the **textData** property, rather than in a separate child text node.

JadeXMLElement Methods

The methods defined in the **JadeXMLElement** class are summarized in the following table.

Method	Description
addAttribute	Creates and adds an attribute
addAttributeNS	Creates and adds an attribute with a namespace
addAttributeObject	Adds an attribute object
addAttributeObjectNS	Adds an attribute object with a namespace
addCDATA	Creates and adds a CDATA node
addCDATAObject	Adds a CDATA object
addComment	Creates and adds a comment
addCommentObject	Adds a comment object
addElement	Creates and adds an element
addElementNS	Creates and adds an element with a namespace
addElementObject	Adds an element object
addElementObjectNS	Adds an element object with a namespace
addProcessingInstruction	Creates and adds a processing instruction
addProcessingInstructionObject	Adds a processing instruction object
addText	Creates and adds a text node
addTextObject	Adds a text object

Method	Description
findAllElementsByNameNS	Fills an array with all descendant elements with the specified namespace URI and local name
findAllElementsByTagName	Fills an array with all descendant elements with the specified tag name
getAllElementsByTagName	Fills an array with all descendant elements with the specified tag name
getAllElementsByTagNameNS	Fills an array with all descendant elements with the specified namespace URI and local name
getAttributeByName	Returns the attribute with the specified name
getAttributeByNameNS	Returns the attribute with the specified namespace URI and local name
getElementByTagName	Returns the first immediate child element with the specified tag name
getElementByTagNameNS	Returns the first immediate child element with the specified namespace URI and local name
getElementsByTagName	Fills an array with the immediate child elements with the specified tag name
getElementsByTagNameNS	Fills an array with the immediate child elements with the specified namespace URI and local name
namespacePrefix	Returns the namespace prefix
parentElement	Returns the parent element of the element
setText	Sets the text content of the element
text	Returns the text content of the element

addAttribute

Signature
```
addAttribute(name:  String;
             value: String): JadeXMLAttribute updating;
```

The **addAttribute** method of the **JadeXMLElement** class creates and adds an attribute node with the values specified in the **name** and **value** parameters and returns a reference to the created **JadeXMLAttribute** node instance.

addAttributeNS

Signature
```
addAttributeNS(namespaceURI:  String;
               qualifiedName: String;
               value:         String): JadeXMLAttribute updating;
```

The **addAttributeNS** method of the **JadeXMLElement** class creates and adds an attribute node with the values specified in the **namespaceURI**, **qualifiedName**, and **value** parameters and returns a reference to the created **JadeXMLAttribute** node instance.

addAttributeObject

Signature
```
addAttributeObject(attribute: JadeXMLAttribute;
                   name:      String;
                   value:     String): JadeXMLAttribute updating;
```

The **addAttributeObject** method of the **JadeXMLElement** class adds an attribute object with the values specified in the **name** and **value** parameters and returns a reference to the added **JadeXMLAttribute** object instance.

addAttributeObjectNS

Signature `addAttributeObjectNS(attribute: JadeXMLAttribute;
namespaceURI: String;
qualifiedName: String;
value: String): JadeXMLAttribute updating;`

The **addAttributeObjectNS** method of the **JadeXMLElement** class adds an attribute object with the values specified in the **namespaceURI**, **qualifiedName**, and **value** parameters and returns a reference to the added **JadeXMLAttribute** object instance.

addCDATA

Signature `addCDATA(data: String): JadeXMLCDATA updating`

The **addCDATA** method of the **JadeXMLElement** class creates and adds a CDATA node with the value specified in the **data** parameter and returns a reference to the created **JadeXMLCDATA** node instance.

addCDATAObject

Signature `addCDATAObject(cdata: JadeXMLCDATA;
data: String): JadeXMLCDATA updating`

The **addCDATAObject** method of the **JadeXMLElement** class adds a CDATA object with the value specified in the **data** parameter and returns a reference to the added **JadeXMLCDATA** object instance.

addComment

Signature `addComment(text: String): JadeXMLComment updating;`

The **addComment** method of the **JadeXMLElement** class creates and adds a comment node with the value specified in the **text** parameter and returns a reference to the created **JadeXMLComment** node instance.

addCommentObject

Signature `addCommentObject(comment: JadeXMLComment;
text: String): JadeXMLComment updating;`

The **addCommentObject** method of the **JadeXMLElement** class adds a comment object with the value specified in the **text** parameter and returns a reference to the added **JadeXMLComment** object instance.

addElement

Signature `addElement(tagName: String): JadeXMLElement updating;`

The **addElement** method of the **JadeXMLElement** class adds a new **JadeXMLElement** node with the value specified in the **tagName** parameter to the receiver element and returns a reference to the created **JadeXMLElement** instance.

addElementNS

Signature `addElementNS(namespaceURI: String;
qualifiedName: String): JadeXMLElement updating;`

The **addElementNS** method of the **JadeXMLElement** class adds a new **JadeXMLElement** node with the values specified in the **namespaceURI** and **qualifiedName** parameters to the receiver element and returns a reference to the created **JadeXMLElement** instance.

addElementObject

Signature `addElementObject(element: JadeXMLElement;
tagName: String): JadeXMLElement updating;`

The **addElementObject** method of the **JadeXMLElement** class adds a new **JadeXMLElement** object with the value specified in the **tagName** parameter to the receiver element and returns a reference to the added **JadeXMLElement** object instance.

addElementObjectNS

Signature `addElementObjectNS(element: JadeXMLElement;
namespaceURI: String;
qualifiedName: String): JadeXMLElement updating;`

The **addElementObjectNS** method of the **JadeXMLElement** class adds a new **JadeXMLElement** object with the values specified in the **namespaceURI** and **qualifiedName** parameters to the receiver element and returns a reference to the added **JadeXMLElement** object instance.

addProcessingInstruction

Signature `addProcessingInstruction(target: String;
data: String): JadeXMLProcessingInstruction updating;`

The **addProcessingInstruction** method of the **JadeXMLElement** class creates and adds a processing instruction node with the values specified in the **target** and **data** parameters and returns a reference to the created **JadeXMLProcessingInstruction** node instance.

addProcessingInstructionObject

Signature `addProcessingInstructionObject(procInstr: JadeXMLProcessingInstruction;
target: String;
data: String): JadeXMLProcessingInstruction updating;`

The **addProcessingInstructionObject** method of the **JadeXMLElement** class adds a processing instruction object with the values specified in the **target** and **data** parameters and returns a reference to the added **JadeXMLProcessingInstruction** object instance.

addText

Signature `addText(data: String): JadeXMLText updating;`

The **addText** method of the **JadeXMLElement** class creates and adds a text node with the value specified in the **data** parameter and returns a reference to the created **JadeXMLText** node instance.

addTextObject

Signature `addTextObject(text: JadeXMLText;
data: String): JadeXMLText updating;`

The **addTextObject** method of the **JadeXMLElement** class adds a text object with the value specified in the **data** parameter and returns a reference to the added **JadeXMLText** object instance.

findAllElementsByNameNS

Signature `findAllElementsByNameNS(namespaceURI: String;
localName: String;
elements: JadeXMLElementArray input);`

The **findAllElementsByNameNS** method of the **JadeXMLElement** class fills the elements array with all descendant elements that have the values specified in the **namespaceURI** and **localName** parameters, respectively.

Note As the search uses the collection sequence, the elements may not be in the document sequence.

If you want to match all namespaces or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter. Note, however, that if you specify **"*"** in the **localName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

findAllElementsByTagName

Signature `findAllElementsByTagName(tagName: String;
elements: JadeXMLElementArray input);`

The **findAllElementsByTagName** method of the **JadeXMLElement** class fills the elements array with all descendant elements that have the value specified in the **tagName** parameter.

Note As the search uses the collection sequence, the elements may not be in the document sequence.

If you want to match all tag names, specify an asterisk character (**'*'**) in the **tagName** parameter. Note, however, that if you specify **"*"** in the **tagName** parameter, the access method uses the document sequence to locate the requested elements rather than the collection sequence that optimizes performance.

getAllElementsByTagName

Signature `getAllElementsByTagName(tagName: String;
elements: JadeXMLElementArray input);`

The **getAllElementsByTagName** method of the **JadeXMLElement** class fills the elements array with all descendant elements in document order (that is, using a preorder traversal) that have the value specified in the **tagName** parameter.

If you want to match all tags, specify an asterisk character (**'*'**) in the **tagName** parameter.

Tip To improve performance when the sequence is not important, use the **findAllElementsByTagName** method to retrieve elements more directly through a collection, by using the collection sequence.

getAllElementsByTagNameNS

Signature
```
getAllElementsByTagNameNS(namespaceURI: String;
                          localName:    String;
                          elements:     JadeXMLElementArray input);
```

The **getAllElementsByTagNameNS** method of the **JadeXMLElement** class fills the elements array with all descendant elements in document order (that is, using a preorder traversal) that have the values specified in the **namespaceURI** and **localName** parameters, respectively.

If you want to match all namespaces or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter.

Tip To improve performance when the sequence is not important, use the **findAllElementsByNameNS** method to retrieve elements more directly through a collection, by using the collection sequence.

getAttributeByName

Signature
```
getAttributeByName(name: String): JadeXMLAttribute;
```

The **getAttributeByName** method of the **JadeXMLElement** class returns a reference to the **JadeXMLAttribute** node instance that has the name specified in the **name** parameter.

getAttributeByNameNS

Signature
```
addAttributeByNameNS(namespaceURI: String;
                     localName:    String): JadeXMLAttribute;
```

The **addAttributeByNameNS** method of the **JadeXMLElement** class returns a reference to the **JadeXMLAttribute** node instance that has the namespace URI and local name specified in the **namespaceURI** and **localName** parameters, respectively.

getElementByTagName

Signature
```
getElementByTagName(tagName: String): JadeXMLElement;
```

The **getElementByTagName** method of the **JadeXMLElement** class returns a reference to the first immediate child **JadeXMLElement** node instance that has the tag name specified in the **tagName** parameter. If you want to return the first child element, specify an asterisk character (**'*'**) in the **tagName** parameter.

getElementByTagNameNS

Signature
```
getElementByTagNameNS(namespaceURI: String;
                      localName:    String): JadeXMLElement;
```

The **getElementByTagNameNS** method of the **JadeXMLElement** class returns a reference to the first immediate child **JadeXMLElement** node instance that has the namespace URI and local name specified in the **namespaceURI** and **localName** parameters, respectively.

If you want to match all namespaces or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter.

getElementsByTagName

Signature `getElementsByTagName(tagName: String; elements: JadeXMLElementArray input);`

The **getElementsByTagName** method of the **JadeXMLElement** class fills the elements array with the immediate child elements in document order (that is, using a preorder traversal) that have the value specified in the **tagName** parameter.

If you want to match all tags, specify an asterisk character (**'*'**) in the **tagName** parameter.

getElementsByTagNameNS

Signature
```
getElementsByTagNameNS(namespaceURI: String;
                       localName:    String;
                       elements:     JadeXMLElementArray input);
```

The **getElementsByTagNameNS** method of the **JadeXMLElement** class fills the elements array with the immediate child elements in document order (that is, using a preorder traversal) that have the values specified in the **namespaceURI** and **localName** parameters, respectively.

If you want to match all namespaces or local names, specify an asterisk character (**'*'**) in the **namespaceURI** or **localName** parameter.

namespacePrefix

Signature `namespacePrefix(): String;`

The **namespacePrefix** method of the **JadeXMLElement** class returns the namespace prefix of the element or it returns null (**""**) if the namespace is not specified.

parentElement

Signature `parentElement(): JadeXMLElement;`

The **parentElement** method of the **JadeXMLElement** class returns the parent element of the receiver element.

All elements except the root element have a parent element reference.

setText

Signature `setText(data: String) updating;`

The **setText** method of the **JadeXMLElement** class sets the text of the receiver element with the text specified in the **data** parameter and deletes all existing child elements.

text

Signature `text(): String;`

The **text** method of the **JadeXMLElement** class returns the text content of the receiver element. The returned text is the value of the **textData** property if the element has no children or it is the concatenated text of all immediate **JadeXMLText** and **JadeXMLCDATA** child nodes.

JadeXMLException Class

The **JadeXMLException** class is the transient class that defines behavior for exceptions that occur as a result of XML processing.

For an example of using methods defined in the **JadeXMLException** class to check that the XML document files are well-formed, see "Handling XML Tree Exceptions", in Chapter 12 of the *JADE Developer's Reference*.

The **Exception** class **errorItem** and **extendedErrorText** properties and the **JadeXMLException** class properties (for details, see "JadeXMLException Properties", in the following section) are used to describe the XML processing exception in more detail. For details about Web service XML exceptions, see the error messages in the range 8900 through 8999 in "Error Messages and System Messages", in the **JADEMsgs.pdf** file. See also the **JadeXMLParser** class.

Inherits From: NormalException

Inherited By: (None)

JadeXMLException Class Constants

The constants provided by the **JadeXMLException** class are listed in the following table.

Constant	Value	Returned when you attempt to…
CannotParsePersistent	8910	Parse an XML document into a persistent object tree using the **parseFile** or **parseFile** method of the **JadeXMLDocument** class. This is not allowed. To parse persistent documents, you must use the **JadeXMLDocumentParser** class and set up a mapping of node.
DocTypeAlreadyDefined	8903	Add a document type declaration to an XML document and one exists already. An XML document can have only one document type declaration.
InvalidClassMapping	8909	Set an invalid mapping for a **JadeXMLNode** class. The mapping is used when instances are created during the parsing of an XML document.
InvalidHierarchyRequest	8905	Add a node to an XML document at an invalid position; for example, moving an element to before a document or attribute.
NullNode	8904	Pass a null node reference to an XML processing method and the parameter cannot be null; for example, specifying a null value as the destination position when moving a node in the document tree.
ParserCreateFailed	8900	Create an instance of the XML parsing engine that cannot be created.
ParserError	8901	Parse an XML document and an error occurs; for example, the document is not well-formed.
ParserNodeMismatch	8908	Access an XML parser object on a different node to the one that created the parser; for example, when a parser is opened on a client node and a server method attempts to use the parser, this exception is raised.
RootElementAlreadyDefined	8902	Add a root (top-level) element to an XML document and one exists already. An XML document must have a single root element.

Constant	Value	Returned when you attempt to…
StringToUTF8Failed	8906	Parse an XML document that cannot be converted from JADE native format to UTF8 format.
UTF8ToStringFailed	8907	Parse an XML document that cannot be converted from UTF8 format to JADE native format.

JadeXMLException Properties

The properties defined in the **JadeXMLException** class are summarized in the following table.

Property	Contains the…
columnNumber	Column number of text where the exception occurred
fileName	Name of the file where the exception occurred
lineNumber	Line number of text where the exception occurred

columnNumber

Type: Integer

The **columnNumber** property of the **JadeXMLException** class contains the column number of the text where the exception occurred or it contains **-1** if no column number is available.

The first column in a line is position **1**.

fileName

Type: String

The **fileName** property of the **JadeXMLException** class contains the name of the file in which the exception occurred or it contains null (**""**) if no file name is available.

lineNumber

Type: Integer

The **lineNumber** property of the **JadeXMLException** class contains the line number of the text where the exception occurred or it contains **-1** if no line number is available.

The first line in an XML document is position **1**.

JadeXMLNode Class

The **JadeXMLNode** class is the abstract superclass of all nodes in an XML document tree. A node has an owning document and it can have child nodes and a parent node.

A node can be copied, moved, or removed, and it can have its XML representation output. (See also "Using the XML Tree Model", in Chapter 12 of the *JADE Developer's Reference*.)

For an example of using methods defined in the **JadeXMLNode** class to search the library document, list all books with a specified author, read a document, and print the names of the elements in that document, indented to show the hierarchy, see "Retrieving Information from XML Tree Documents", in Chapter 12 of the *JADE Developer's Reference*.

For details about the properties and methods defined in the **JadeXMLNode** class, see "JadeXMLNode Properties" and "JadeXMLNode Methods", in the following sections.

For more details and detailed examples, see the *XML in JADE White Paper*.

Inherits From: Object

Inherited By: JadeXMLAttribute, JadeXMLCharacterData, JadeXMLDocument, JadeXMLDocumentType, JadeXMLElement, JadeXMLProcessingInstruction

JadeXMLNode Properties

The properties defined in the **JadeXMLNode** class are summarized in the following table.

Property	Contains…
childNodes	Array of children of the node
document	Owning document of the node
parentNode	Parent of the node

childNodes

Type: JadeXMLNodeArray

The read-only **childNodes** property of the **JadeXMLNode** class contains a reference to an array of children of the receiver node.

document

Type: JadeXMLDocument

The read-only **document** property of the **JadeXMLNode** class contains a reference to the owning document of the receiver node.

parentNode

Type: JadeXMLNode

The read-only **parent** property of the **JadeXMLNode** class contains a reference to the parent of the receiver node.

JadeXMLNode Methods

The methods defined in the **JadeXMLNode** class are summarized in the following table.

Method	Description
copyAfter	Copies the node and inserts it after the specified node
copyAsChildOf	Copies the node and inserts it as a child of the specified node
copyBefore	Copies the node and inserts it before the specified node
descendsFrom	Specifies whether the specified XML node is an ancestor of the receiver **JadeXMLNode** class
moveAfter	Moves the node to the position after the specified node
moveAsChildOf	Moves the node to the position as a child of the specified node
moveBefore	Moves the node to the position before the specified node
remove	Removes the node from the XML tree and then deletes the node
writeToString	Writes the node to a string

copyAfter

Signature `copyAfter(siblingNode: JadeXMLNode input): JadeXMLNode;`

The **copyAfter** method of the **JadeXMLNode** class creates a copy of the node and all of its descendant nodes, inserts it into the XML tree after the node specified in the **siblingNode** parameter, and returns a reference to the created **JadeXMLNode** node instance.

copyAsChildOf

Signature `copyAsChildOf(parentNode: JadeXMLNode input): JadeXMLNode;`

The **copyAsChildOf** method of the **JadeXMLNode** class creates a copy of the node and all of its descendant nodes, inserts it into the XML tree as a child of the node specified in the **parentNode** parameter, and returns a reference to the created **JadeXMLNode** node instance.

copyBefore

Signature `copyBefore(siblingNode: JadeXMLNode input): JadeXMLNode;`

The **copyBefore** method of the **JadeXMLNode** class creates a copy of the node and all of its descendant nodes, inserts it into the XML tree before the node specified in the **siblingNode** parameter, and returns a reference to the created **JadeXMLNode** node instance.

descendsFrom

Signature `descendsFrom(nod: JadeXMLNode): Boolean;`

The **descendsFrom** method of the **JadeXMLNode** class returns **true** if the **JadeXMLNode** node specified in the **nod** parameter is an ancestor of the receiver **JadeXMLNode** object.

moveAfter

Signature `moveAfter(siblingNode: JadeXMLNode input): JadeXMLNode updating;`

The **moveAfter** method of the **JadeXMLNode** class moves the node to the position in the XML tree after the node specified in the **siblingNode** parameter, and returns a reference to the moved **JadeXMLNode** node instance.

moveAsChildOf

Signature `moveAsChildOf(parentNode: JadeXMLNode input): JadeXMLNode updating;`

The **moveAsChildOf** method of the **JadeXMLNode** class moves the node to the position in the XML tree as a child of the node specified in the **parentNode** parameter, and returns a reference to the moved **JadeXMLNode** node instance.

moveBefore

Signature `moveBefore(siblingNode: JadeXMLNode input): JadeXMLNode updating;`

The **moveBefore** method of the **JadeXMLNode** class moves the node to the position in the XML tree before the node specified in the **siblingNode** parameter, and returns a reference to the moved **JadeXMLNode** node instance.

remove

Signature `remove() updating;`

The **remove** method of the **JadeXMLNode** class removes the node from the XML tree and then deletes the node instance.

writeToString

Signature `writeToString(): String;`

The **writeToString** method of the **JadeXMLNode** class writes the XML representation of the node to a string and then returns the string.

You can control the format of the output by setting the **JadeXMLDocument** class formatting properties of the document object; for example, the **endOfLine** and **indentString** properties.

The following example parses a simple document string and formats the print output.

```
write1();
vars
    doc : JadeXMLDocument;
begin
    create doc;
    doc.indentString := '    ';
    doc.parseString('<name><first>John</first><last>Smith</last></name>');
    write doc.writeToString;
    delete doc;
end;
```

The output from the **write1** method shown in the previous example is as follows.

```
<?xml version="1.0"?>
<name>
    <first>John</first>
    <last>Smith</last>
</name>
```

JadeXMLParser Class

The **JadeXMLParser** class is the abstract transient class that defines behavior for parsing XML documents. The parser reads an XML document and reports basic document-related events; for example, the start and end of elements and character data. (See also "JADE XML Parser Model", in Chapter 12 of the *JADE Developer's Reference*.)

The JADE XML Parser model reads an XML document from beginning to end. As it encounters start-tags, end-tags, text, comments, and so on, it notifies the client application by calling event handler methods defined by the application.

Define a subclass of the **JadeXMLParser** class in your application and implement callback methods for events about which you require notification. The application creates an instance of the subclass and then calls the **parseFile** or **parseString** method to parse an XML document.

The parser raises a **JadeXMLException** object if it detects any errors resulting from XML documents that are not well-formed.

Note The order of events is very important, and mirrors the order of information in the document itself. For example, all contents of a **JadeXMLElement** (that is, character data, processing instructions, and any sub-elements) appear in order between the **startElement** event and the corresponding **endElement** event.

For examples of using methods defined in the **JadeXMLParser** class to parse a document and print the names of the elements in that document, showing the hierarchy, and to check that documents are well-formed, see "Parsing an XML Document" and "Handling XML Parser Exceptions", respectively, in Chapter 12 of the *JADE Developer's Reference*.

For details about the methods defined in the **JadeXMLParser** class, see "JadeXMLParser Methods", in the following section. See also "Parsing an XML Tree Document" and "Parsing an XML Document", in Chapter 12 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: JadeXMLDocumentParser

JadeXMLParser Methods

The methods defined in the **JadeXMLParser** class are summarized in the following table.

Method	Description
characters	Receives notification of character data
columnNumber	Returns the column number at which the current document event ends
comment	Receives notification of a comment
endCDATA	Receives notification of the end of a CDATA section
endDTD	Receives notification of the end of DTD declarations
endElement	Receives notification of the end of an element
fileName	Returns the file name for the current document event
getAttribute	Retrieves the attribute with the specified index
getAttributeValueByName	Retrieves the value of the attribute with the specified qualified name

Method	Description
getAttributeValueByNameNS	Retrieves the value of the attribute with the specified namespace URI and local name
lineNumber	Returns the line number at which the current document event ends
parseFile	Parses the specified XML document file
parseString	Parses the specified XML document string
processingInstruction	Receives notification of a processing instruction
startCDATA	Receives notification of the start of a CDATA section
startDTD	Receives notification of the start of DTD declarations
startElement	Receives notification of the beginning of an element

characters

Signature `characters(text: String);`

The **characters** event method of the **JadeXMLParser** class receives notification of the character data specified in the **text** parameter.

The parser calls this event method (if implemented) to report each chunk of character data. The parser may return all contiguous character data in a single chunk or it may split it into several chunks.

columnNumber

Signature `columnNumber(): Integer;`

The **columnNumber** method of the **JadeXMLParser** class returns the number of the column at which the current document event ends.

Note This is the column number of the first character after the text associated with the document event. The first column in a line is position **1**.

If the column number is not available, **-1** is returned.

comment

Signature `comment(text: String);`

The **comment** event method of the **JadeXMLParser** class receives notification of an XML comment specified in the characters in the **text** parameter.

The parser calls this event method (if implemented) to report comments anywhere in the XML document (that is, inside or outside the root element).

endCDATA

Signature `endCDATA();`

The **endCDATA** event method of the **JadeXMLParser** class receives notification of the end of a CDATA section.

endDTD

Signature `endDTD();`

The **endDTD** event method of the **JadeXMLParser** class receives notification of the end of DTD declarations.

The parser invokes this event method (if implemented) at the end of the **DOCTYPE** declaration. If the document has no **DOCTYPE** declaration, this method is not invoked.

endElement

Signature
```
endElement(namespaceURI:  String;
           localName:     String;
           qualifiedName: String);
```

The **endElement** event method of the **JadeXMLParser** class receives notification of the end of an element. The **endElement** event method parameters are listed in the following table.

Parameter	Description
namespaceURI	Namespace URI or null (**""**) if the element has no namespace URI
localName	Local name (without the prefix)
qualifiedName	Qualified name (with the prefix)

The parser invokes this event method (if implemented) at the end of every element in the XML document.

There is a corresponding **startElement** event method for each **endElement** event, even when the element is empty.

fileName

Signature `fileName(): String;`

The **fileName** method of the **JadeXMLParser** class returns the file name for the current document event or null (**""**) if it is not available.

getAttribute

Signature
```
getAttribute(index:         Integer;
             namespaceURI:  String output;
             localName:     String output;
             qualifiedName: String output;
             type:          String output;
             value:         String output): Boolean;
```

The **getAttribute** method of the **JadeXMLParser** class retrieves the attribute with the value specified in the **index** parameter. The **getAttribute** method parameters are listed in the following table.

Parameter	Description
index	The index (starting from **1**)
namespaceURI	Namespace URI or null (**""**) if the name has no namespace URI

Parameter	Description
localName	Local name (without the prefix)
qualifiedName	Qualified name (with the prefix)
type	Reserved for future use
value	The value of the attribute

The **getAttribute** method returns **true** if the specified index is in range or it returns **false** if it is not in range.

The number of attributes attached to the element is passed as a parameter to the **startElement** event method.

The **getAttribute** method returns valid results only during the scope of the **startElement** method invocation.

getAttributeValueByName

Signature
```
getAttributeValueByName(qualifiedName: String;
                        value:         String output): Boolean;
```

The **getAttributeValueByName** method of the **JadeXMLParser** class retrieves the value of the attribute with the qualified name specified in the **qualifiedName** parameter.

The **getAttributeValueByName** method parameters are listed in the following table.

Parameter	Description
qualifiedName	The qualified name (with the prefix)
value	The value of the attribute

The **getAttributeValueByName** method returns **true** if the specified name was found or it returns **false** if it is not found.

This method returns valid results only during the scope of the **startElement** method invocation.

getAttributeValueByNameNS

Signature
```
getAttributeValueByNameNS(namespaceURI: String;
                          localName:    String;
                          value:        String output): Boolean;
```

The **getAttributeValueByNameNS** method of the **JadeXMLParser** class retrieves the value of the attribute with the namespace URI and local name specified in the **namespaceURI** and **localName** parameters, respectively.

The **getAttributeValueByName** method parameters are listed in the following table.

Parameter	Description
namespaceURI	Namespace URI or null (**""**) if the name has no namespace URI
localName	The local name (without the prefix)
value	The value of the attribute

The **getAttributeValueByNameNS** method returns **true** if the specified name was found or it returns **false** if it is not found. This method returns valid results only during the scope of the **startElement** method invocation.

lineNumber

Signature `lineNumber(): Integer;`

The **lineNumber** method of the **JadeXMLParser** class returns the number of the line at which the current document event ends.

Note This is the line position of the first character after the text associated with the document event. The first line in a document is position **1**.

If the line number is not available, **-1** is returned.

parseFile

Signature `parseFile(fileName: String);`

The **parseFile** method of the **JadeXMLParser** class parses the XML document file specified by the **fileName** parameter. This method is synchronous, and it will not return until parsing has ended. An application in which you may require early termination of parsing should raise an exception.

Note As applications cannot invoke this method while a parse operation is in progress, create a new **JadeXMLParser** object instead for each nested XML document. When a parse is complete, the application can reuse the same **JadeXMLParser** object.

During the parse operation, the JADE Parser provides information about the XML document through the implemented callback methods.

parseString

Signature
```
parseString(str:     String
            isFinal: ParamListType);
```

The **parseString** method of the **JadeXMLParser** class parses the XML document string (or the next part of the document string) specified in the **str** parameter. The optional **isFinal** parameter is a Boolean parameter (which is **true** by default) that informs the parser that this is the last piece of the document. The following example shows the use of the **parseString** method.

```
vars
    doc : JadeXMLDocument;
begin
    create doc;
    doc.parseString('<employee>John Smith</employee>');
end;
```

This method is synchronous, and it will not return until parsing has ended. An application in which you may require early termination of parsing should raise an exception.

Note As applications cannot invoke this method while a parse operation is in progress, create a new **JadeXMLParser** object instead for each nested XML document. When a parse is complete, the application can reuse the same **JadeXMLParser** object.

During the parse operation, the JADE Parser provides information about the XML document through the implemented callback methods.

processingInstruction

Signature
```
processingInstruction(target: String;
                      data   String);
```

The **processingInstruction** event method of the **JadeXMLParser** class receives notification of a processing instruction.

The **processingInstruction** event method parameters are listed in the following table.

Parameter	Description
target	The processing instruction target.
data	The processing instruction data or null (**""**) if none was supplied. The data does not include any whitespace separating it from the target.

The parser invokes this event method (if implemented) for each processing instruction that it locates.

Note Processing instructions can occur before or after the root element.

startCDATA

Signature
```
startCDATA();
```

The **startCDATA** event method of the **JadeXMLParser** class receives notification of the start of a CDATA section. The contents of the CDATA section are reported through the regular **characters** event method.

startDTD

Signature
```
startDTD(name:     String;
         publicId: String;
         systemId: String);
```

The **startDTD** event method of the **JadeXMLParser** class receives notification of the start of DTD declarations, if any. The **startDTD** event method parameters are listed in the following table.

Parameter	Description
name	Document-type name
publicId	Declared public identifier for the external DTD subset or null (**""**) if none was declared
systemId	Declared system identifier for the external DTD subset or null (**""**) if none was declared

The parser invokes this method (if implemented) at the beginning of the **DOCTYPE** declaration. If the document has no **DOCTYPE** declaration, this method is not invoked.

startElement

Signature

```
startElement(namespaceURI:   String;
             localName:      String;
             qualifiedName:  String;
             attributeCount: Integer);
```

The **startElement** event method of the **JadeXMLParser** class receives notification of the beginning of an element. The **startElement** event method parameters are listed in the following table.

Parameter	Description
namespaceURI	Namespace URI or null (**""**) if the element has no namespace URI.
localName	Local name (without the prefix).
qualifiedName	Qualified name (with the prefix).
attributeCount	Number of attributes attached to the element. You can retrieve attributes by calling the get attribute methods (that is, the **getAttribute**, **getAttributeValueByName**, or **getAttributeValueByNameNS** method) from within the **startElement** method.

The parser invokes this method (if implemented) at the beginning of every element in the XML document.

There is a corresponding **endElement** event for every **startElement** event (even when the element is empty).

All of the content of an element is reported, in order, before the corresponding **endElement** event.

JadeXMLProcessingInstruction Class

The **JadeXMLProcessingInstruction** class represents an XML processing instruction (that is, an application-specific instruction on how to handle an XML document after the document has been parsed).

For details about the properties defined in the **JadeXMLProcessingInstruction** class, see "JadeXMLProcessingInstruction Properties", in the following section.

Inherits From: JadeXMLNode

Inherited By: (None)

JadeXMLProcessingInstruction Properties

The properties defined in the **JadeXMLProcessingInstruction** class are summarized in the following table.

Property	Contains the …
data	Content of the processing instruction
target	Target of the processing instruction

data

Type: String

The **data** property of the **JadeXMLProcessingInstruction** class contains the content of the processing instruction.

target

Type: String

The **target** property of the **JadeXMLProcessingInstruction** class contains the target of the processing instruction; that is, the name of the application to which the processing instruction should be passed.

JadeXMLText Class

The **JadeXMLText** class represents the textual content within an XML document tree.

If there is no markup inside the content of an element, the text may be stored directly in the **JadeXMLElement** class **textData** property of the element rather than as a child text node. This optimization reduces the size of the document tree and improves parsing performance.

Inherits From: JadeXMLCharacterData

Inherited By: (None)

List Class

The **List** class encapsulates behavior required to reference objects by their position in the collection. This position is sometimes referred to as an *index* or *subscript*.

Note A list is an ordered collection.

For details about the methods defined in the **List** class, see "List Methods", in the following subsection.

Inherits From: Collection

Inherited By: Array

List Methods

The methods defined in the **List** class are summarized in the following table.

Method	Description
clear	Clears all entries from the collection
copy	Copies entries from the receiver to a compatible collection
purge	Deletes all object references in the collection

clear

Signature `clear() updating;`

The **clear** method of the **List** class clears all entries from the collection array, as shown in the following example.

```
countNotes();
vars
    noteArray : NotificationArray;
begin
    create noteArray transient;
    system.getNotes(noteArray, true, 32000);
    write noteArray.size.String & ' transient notifications';
    noteArray.clear;
    system.getNotes(noteArray, false, 32000);
    write noteArray.size.String & ' persistent notifications';
epilog
    delete noteArray;
end;
```

copy

Signature `copy(toColl: Collection input);`

The **copy** method of the **List** class copies entries from the receiver collection to a compatible collection passed as the **toColl** parameter. In this case, compatible means that the memberships of the receiver and destination collections are type-compatible.

Note By default, entries copied from the receiver collection are *added* to entries that already exist in the collection to which you copy them.

purge

Signature `purge() updating;`

The **purge** method of the **List** class deletes all objects referenced in a list and clears the list collection; that is, **size = 0**.

The code fragment in the following example shows the use of the **purge** method.

```
myPopupColumnList.purge;
```

Locale Class

The **Locale** class is the persistent class that defines the locales (languages) supported by a schema. Although a schema inherits all of the locales of its superschemas, a locale can be updated only in the schema in which it was defined.

In JADE thin client mode, all locale information is based on the locale of the presentation client that initiated the application. Only the options defined by the application server for that locale apply. This locale must be installed on the application server workstation. Any local changes on the presentation client to the locale options are ignored (for example, the date format).

As supported Windows operating systems do not allow different threads of the same process to use different locales, each presentation client application uses the default locale for the application server workstation.

For details about the constants, properties, and methods defined in the **Locale** class, see "Locale Class Constants", "Locale Properties", and "Locale Methods", in the following subsections.

Inherits From: SchemaEntity

Inherited By: (None)

Locale Class Constants

The constants provided by the **Locale** class are listed in the following table.

Constant	Character Value	Specifies that the …
INHERITED	'I'	Current locale is inherited from a superschema
LOCAL	'L'	Locale is local to (defined in) the current schema

Locale Properties

The properties defined in the **Locale** class are summarized in the following table.

Property	Contains …
cloneOf	The locale from which the receiver is cloned
clones	All of the locales that are clones of the receiver
forms	All of the forms for the locale
languageId	The Windows language identifier of the locale
schema	The schema in which the locale is defined
translatableStrings	All translatable strings for the locale

cloneOf

Type: Locale

The **cloneOf** property of the **Locale** class is a protected property that is for internal system use only. It contains a reference to the locale from which the receiver is cloned.

clones

Type: LocaleNDict

The **clones** property of the **Locale** class is a protected property that is for internal system use only. It contains a reference to all of the locales that are clones of the receiver.

forms

Type: FormNameDict

The **forms** property of the **Locale** class is a protected property that is for internal system use only. It contains a reference to all of the forms for the locale.

languageId

Type: Integer

The read-only **languageId** property of the **Locale** class contains the Windows language identifier for the locale.

schema

Type: Schema

The **schema** property of the **Locale** class is a protected property that is for internal system use only. It contains a reference to the schema in which the locale of the receiver is defined.

translatableStrings

Type: ConstantNDict

The **translatableStrings** property of the **Locale** class is a protected property that is for internal system use only. It contains a reference to all translatable strings for the locale.

Locale Methods

The methods defined in the **Locale** class are summarized in the following table.

Method	Description
getAllTranslatableStrings	Returns all translatable strings for this locale in the current schema and superschemas
getForms	Returns all forms for the locale
getStringValue	Returns the definition of the specified translatable string in the receiver locale
getTranslatableStringLocal	Returns the translatable string with the specified name in the receiver locale
getTranslatableStrings	Returns the translatable strings ordered by name in the current locale
getTranslatableStringsByNum	Returns the translatable strings ordered by number in the current locale
hasClones	Specifies whether clones of the locale exist
isClone	Specifies whether the locale is a clone of another locale
makeLocaleName	Returns the name of the locale

getAllTranslatableStrings

Signature `getAllTranslatableStrings(): ConstantNDict;`

The **getAllTranslatableStrings** method of the **Locale** class returns a reference to a dictionary containing the translatable strings for the receiver locale in the current schema and all superschemas except for the **RootSchema**.

getForms

Signature `getForms(): FormNameDict;`

The **getForms** method of the **Locale** class returns a reference to a dictionary of all forms for the locale.

getStringValue

Signature `getStringValue(xltStringName: String): String;`

The **getStringValue** method of the **Locale** class returns a string containing the value (definition) of the translatable string specified in the **xltStringName** parameter.

This method is valid only for translatable strings that can be evaluated at compile time; that is, they do not have parameters and they do not reference other translatable strings or constants.

getTranslatableStringLocal

Signature `getTranslatableStringLocal(name: String): TranslatableString;`

The **getTranslatableStringLocal** method of the **Locale** class returns a translatable string from the collection of translatable strings in the receiver locale with the name specified by the value of the **name** parameter.

getTranslatableStrings

Signature `getTranslatableStrings(): ConstantNDict;`

The **getTranslatableStrings** method of the **Locale** class returns a reference to a dictionary containing the translatable strings in the locale of the receiver ordered by name.

If the receiver is a clone, the collection is that of the associated base locale.

getTranslatableStringsByNum

Signature `getTranslatableStringsByNum():SchemaEntityNumberDict;`

The **getTranslatableStringsByNum** method of the **Locale** class returns a reference to a dictionary containing the translatable strings in the locale of the receiver ordered by number.

If the receiver is a clone, the collection is that of the associated base locale.

hasClones

Signature `hasClones(): Boolean;`

The **hasClones** method of the **Locale** class returns **true** if clones of the locale exist.

isClone

Signature `isClone(): Boolean;`

The **isClone** method of the **Locale** class returns **true** if the locale is a clone of another locale.

makeLocaleName

Signature `makeLocaleName(): String;`

The **makeLocaleName** method of the **Locale** class returns a string containing the name of the locale; for example, **English (New Zealand)**.

LocaleFormat Class

The **LocaleFormat** class defines the common protocol for locale format information.

For details about the property defined in the **LocaleFormat** class, see "LocaleFormat Property", in the following subsection.

Inherits From: Feature

Inherited By: DateFormat, NumberFormat, TimeFormat

LocaleFormat Property

The property defined in the **LocaleFormat** class is summarized in the following table.

Property	Description
schema	Contains the schema in which the locale format is defined

schema

Type: Schema

The **schema** property of the **LocaleFormat** class is a protected property that is for internal system use only.

This property contains a reference to the schema in which the locale format is defined.

LocaleFullInfo Class

The **LocaleFullInfo** class is used to provide Windows locale information for the current workstation. An instance of the **LocaleFullInfo** class is created when an application starts, and it is accessible from the **currentLocaleInfo** property of the **Application** class.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of **dd/mm/yyyy**, and it has been overridden with a short date format of ***yyyy-mm-dd***, this is returned in the default **dd/mm/yyyy** format.

For details about the constants and properties defined in the **LocaleFullInfo** class, see "LocaleFullInfo Class Constants" and "LocaleFullInfo Properties", in the following subsections.

Inherits From: LocaleNameInfo

Inherited By: (None)

LocaleFullInfo Class Constants

The constants provided by the **LocaleFullInfo** class are listed in the following table.

Constant	Integer Value
Imperial	1
Metric	0

LocaleFullInfo Properties

The properties defined in the **LocaleFullInfo** class are summarized in the following table.

Property	Contains …
currencyInfo	Currency formatting information for the current locale
dateInfo	Date formatting information for the current locale
defaultCodePage	Code page associated with the current locale
defaultCountryCode	The default country code for the current locale
defaultLanguageId	The default language for the current locale
listSeparator	The separator used to separate elements in a list in the current locale
measurementSystem	The measurement system of the current locale
nativeDigits	The ten characters in the native encoding equivalent to the ASCII values **0** through **9**
numericInfo	Numeric formatting information for the current locale
timeInfo	Time formatting information for the current locale

currencyInfo

Type: CurrencyFormat

Availability: Read-only at any time

The **currencyInfo** property of the **LocaleFullInfo** class contains a reference to the currency formatting information for the current locale. For details, see the **CurrencyFormat** class.

dateInfo

Type: DateFormat

Availability: Read-only at any time

The **dateInfo** property of the **LocaleFullInfo** class contains a reference to date formatting information for the current locale. For details, see the **DateFormat** class.

defaultCodePage

Type: Integer

Availability: Read-only at any time

The **defaultCodePage** property of the **LocaleFullInfo** class contains the default code page for the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

defaultCountryCode

Type: Integer

Availability: Read-only at any time

The **defaultCountryCode** property of the **LocaleFullInfo** class contains the default country code for the current locale. The default country code is the code of the principal language for this locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

defaultLanguageId

Type: Integer

Availability: Read-only at any time

The **defaultLanguageId** property of the **LocaleFullInfo** class contains the default language identifier for the current locale.

The default language identifier is the identifier of the principal language for this locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

listSeparator

Type: String[20]

Availability: Read-only at any time

The **listSeparator** property of the **LocaleFullInfo** class contains the list separator used to separate elements in a list for the current locale.

measurementSystem

Type: Integer

Availability: Read-only at any time

The **measurementSystem** property of the **LocaleFullInfo** class contains the measurement system for the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

The measurement can be one of the values specified in the following table.

Value	LocaleFullInfo Class Constant	Description
0	Metric	Metric measurement system
1	Imperial	Imperial measurement system

nativeDigits

Type: String[10]

Availability: Read-only at any time

The **nativeDigits** property of the **LocaleFullInfo** class contains the ten characters in the native encoding equivalent to the ASCII values **0** through **9** for the current locale; for example, Arabic or Kanji native digits.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

numericInfo

Type: NumberFormat

Availability: Read-only at any time

The **numericInfo** property of the **LocaleFullInfo** class contains a reference to the numeric formatting information for the current locale. For details, see the **NumberFormat** class.

timeInfo

Type: TimeFormat

Availability: Read-only at any time

The **timeInfo** property of the **LocaleFullInfo** class contains a reference to time formatting information for the current locale. For details, see the **TimeFormat** class.

LocaleNameInfo Class

The **LocaleNameInfo** class is used to provide Windows locale information for the current workstation. An instance of **LocaleFullInfo** (a subclass of **LocaleNameInfo)** is created when an application starts, and it is accessible from the **currentLocaleInfo** property of the **Application** class.

All properties of the **LocaleNameInfo** class are available from the **currentLocaleInfo** property of the **Application** class.

For details about the properties defined in the **LocaleNameInfo** class, see "LocaleNameInfo Properties", in the following subsection.

Inherits From: Object

Inherited By: LocaleFullInfo

LocaleNameInfo Properties

The properties defined in the **LocaleNameInfo** class are summarized in the following table.

Property	Contains the …
abbreviatedCountryName	Abbreviated country name
abbreviatedLangName	Abbreviated language name
countryCode	Country code
englishCountryName	English language name of the country
englishLangName	Language name in English
languageId	Language identifier of the locale
localeId	Locale identifier
localizedCountryName	Localized name of the country
localizedLangName	Localized name of the language
nativeCountryName	Native name of the country
nativeLangName	Native name of the language

abbreviatedCountryName

Type: String[127]

Availability: Read-only at any time

The **abbreviatedCountryName** property of the **LocaleNameInfo** class contains the abbreviated name of the country (conforming to ISO Standard 3166) for the current locale.

abbreviatedLangName

Type: String[10]

Availability: Read-only at any time

The **abbreviatedLangName** property of the **LocaleNameInfo** class contains the abbreviated name of the language for the current locale.

The abbreviated language name consists of the two-letter ISO Standard 639 language abbreviation followed by a third letter that represents the language form, or dialectical variant, as appropriate.

countryCode

Type: Integer

Availability: Read-only at any time

The **countryCode** property of the **LocaleNameInfo** class contains the country code of the current locale.

The country code consists of the telephone country code.

englishCountryName

Type: String[127]

Availability: Read-only at any time

The **englishCountryName** property of the **LocaleNameInfo** class contains the full English name of the country of the current locale.

englishLangName

Type: String[127]

Availability: Read-only at any time

The **englishLangName** property of the **LocaleNameInfo** class contains the full English name of the language from ISO Standard 639.

languageId

Type: Integer

Availability: Read-only at any time

The **languageId** property of the **LocaleNameInfo** class contains the Windows language identifier for the current locale.

localeId

Type: Integer

Availability: Read-only at any time

The **localeId** property of the **LocaleNameInfo** class contains the Windows locale identifier for the current locale.

localizedCountryName

Type: String[127]

Availability: Read-only at any time

The **localizedCountryName** property of the **LocaleNameInfo** class contains the full localized name of the country for the current locale.

localizedLangName

Type: String[127]

Availability: Read-only at any time

The **localizedLangName** property of the **LocaleNameInfo** class contains the full localized name of the language for the current locale.

nativeCountryName

Type: String[127]

Availability: Read-only at any time

The **nativeCountryName** property of the **LocaleNameInfo** class contains the native name of the country for the current locale.

nativeLangName

Type: String[127]

Availability: Read-only at any time

The **nativeLangName** property of the **LocaleNameInfo** class contains the native name of the language for the current locale.

Lock Class

Instances of the **Lock** class are used to describe the locks and lock requests maintained by the system.

For details about the class constants, properties, and method defined in the **Lock** class, see "Lock Class Constants", "Lock Properties", and "Lock Method", in the following subsections.

Inherits From: Object

Inherited By: (None)

Lock Class Constants

The constants provided by the **Lock** class are listed in the following table.

Constant	Character Value	Description
Kind_Local	'01'	Applies to stable objects and represents a shared, transient duration lock that has an associated node lock (**Kind_Node**) entry in the database server lock tables. There is no individual lock entry for the process in the database server lock tables (unless the process is a server application).
Kind_Node	'02'	Applies to stable objects and represents a shared, transient duration lock that can be held by one or more processes on the node associated with the node lock. The associated background process of the node is used as the locking process. A node lock is released when an exclusive lock request is received or the object is removed from the node's cache and there are no processes on the associated node that have the object locked with local locks.
Kind_Normal	'00'	Represents a lock held by a process. It is released when the process unlocks the object.

For more details, see the **kind** property.

Lock Properties

The properties defined in the **Lock** class are summarized in the following table.

Property	Description
duration	Specifies when the object is automatically unlocked
elapsedTime	Contains the time that the lock has been in place
kind	Contains the type of lock (that is, normal, local, node, or node lock to be removed)
lockedBy	Contains the process that currently holds a lock
requestedBy	Contains the process that submitted the lock request
requestTime	Contains the date and time of the lock request
type	Contains the type of lock request
waitTime	Contains the length of time the lock request waits

duration

Type: Character[1]

The read-only **duration** property of the **Lock** class is set to the value of the **duration** parameter of the lock request.

The **duration** parameter of the lock request specifies if the object is automatically unlocked at the end of transaction time or at the end of the current session (that is, the current thread, or process).

If a manual unlock is issued, the object is unlocked only if you are not in transaction or load state.

In persistent transaction state, all unlock requests for persistent objects are ignored. Similarly, in transient transaction state, all unlock requests for shared transient objects are ignored. A session lock is therefore not released if the unlock request is made while in transaction state. To release a session lock, the unlock request must be made while not in transaction state.

The character values correspond to the **Transaction_Duration** and **Session_Duration** global constants in the **LockDurations** category, respectively.

The following example shows the use of the **duration** property.

```
vars
    lock       : Lock;
    locksArray : LockArray;
begin
    create locksArray transient;
    system.getLocks(locksArray, 40);
        foreach lock in locksArray do     //access the lock entry properties
            write lock.duration.Integer.String;
    endforeach;
    locksArray.purge;
    delete locksArray;
end;
```

elapsedTime

Type: Time

The read-only **elapsedTime** property of the **Lock** class is set to the time that the lock request has been in place.

For a queued lock entry, this is the time that the request has been waiting. (If you want to obtain the total length of time from the first attempt to obtain the lock up to the time it times out, use the **Lock** class **waitTime** property.)

kind

Type: Character[1]

The read-only **kind** property of the **Lock** class is set to the kind of node lock for stable objects. (For details, see "Stable Objects" under "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.)

The character values correspond to the **Lock** class constants listed in the following table.

Constant	Character Value	Description
Kind_Local	'01'	Applies to stable objects and represents a shared, transient duration lock that has an associated node lock (**Kind_Node**) entry in the database server lock tables. There is no individual lock entry for the process in the database server lock tables (unless the process is a server application).
Kind_Node	'02'	Applies to stable objects and represents a shared, transient duration lock that can be held by one or more processes on the node associated with the node lock. The associated background process of the node is used as the locking process. A node lock is released when an exclusive lock request is received or the object is removed from the node's cache and there are no processes on the associated node that have the object locked with local locks.
Kind_Normal	'00'	Represents a lock held by a process. It is released when the process unlocks the object.

Kind_Local locks are present only on the local node on which the lock request was issued.

You can use the **lockedBy** property to determine the node associated with a **Kind_Node**, as it contains a reference to the background process of the associated node.

The code fragment in the following example shows the use of the **kind** property.

```
foreach lock in locksArray do
    if lock.kind = lock.Kind_Node then
        write "Exclusive lock request pending";
        write lock.lockedBy.String;
    endif;
endforeach;
```

lockedBy

Type: Process

The read-only **lockedBy** property of the **Lock** class is a read-only property that is set to the process that currently holds a lock on the target object of the lock.

If the lock request is not waiting in the locks queue, this reference is set to **null**.

The code fragment in the following example shows the use of the **lockedBy** property.

```
foreach lock in locksArray do
    write lock.target.String;
    write lock.lockedBy.String;
endforeach;
```

requestedBy

Type: Process

The read-only **requestedBy** property of the **Lock** class is a read-only property that is set to the process that submitted the lock request.

The code fragment in the following example shows the use of the **requestedBy** property.

```
foreach lock in locksArray do
    listBoxQueue.addItem(lock.requestedBy.node.system.name.String);
endforeach;
```

requestTime

Type: TimeStamp

The read-only **requestTime** property of the **Lock** class is set to the date and time of the lock request.

type

Type: Character[1]

The read-only **type** property of the **Lock** class is set to the value of the **type** parameter of the lock request. The character values correspond to the **Update_Lock**, **Share_Lock**, **Reserve_Lock**, and **Exclusive_Lock** global constant in the **Locks** category.

For more details, see "Locks Category", in Appendix A of the *JADE Encyclopaedia of Primitive Types*, and "JADE Locking", in Chapter 6 of the *JADE Developer's Reference*.

The code fragment in the following example shows the use of the **type** property.

```
foreach lock in locksArray do
    if lock.type.Integer = Share_Lock then
        write "Shared lock";
    elseif lock.type.Integer = Exclusive_Lock then
        write "Exclusive lock";
    elseif lock.type.Integer = Reserve_Lock then
        write "Reserve lock";
    endif;
endforeach;
```

waitTime

Type: Time

The read-only **waitTime** property of the **Lock** class is set to the **timeout** parameter of the lock request. This property specifies the total length of time the lock request waits from the time it first attempted to get the lock until a lock exception is reported back if the object is currently locked by another user.

The following example shows the use of the **waitTime** property.

```
vars
    lock       : Lock;
    locksArray : LockArray;
begin
    create locksArray transient;
    system.getQueuedLocks(locksArray, 40);
    foreach lock in locksArray do        //access the lock entry properties
        write lock.requestedBy.String;
        write lock.elapsedTime.String;
        write lock.waitTime.String;
    endforeach;
```

```
    locksArray.purge;
    delete locksArray;
end;
```

If you want to obtain the length of time the lock request has been in place, use the **Lock** class **elapsedTime** property.

Lock Method

The method defined in the **Lock** class is summarized in the following table.

Method	Description
target	Gets the object that is the target of the lock request

target

Signature `target(): Object;`

The **target** method of the **Lock** class is used to obtain a reference to the object that is the target of a lock request.

The following example shows the use of the **target** method.

```
vars
    lock       : Lock;
    locksArray : LockArray;
begin
    create locksArray transient;
    system.getLocks(locksArray, 40);
        foreach lock in locksArray do
            //access the lock entry properties
            write lock.requestedBy.String;
            write lock.target.String;
        endforeach;
    locksArray.purge;
    delete locksArray;
end;
```

LockArray Class

The **LockArray** class is the transient class that encapsulates behavior required to access **Lock** objects in an array.

The locks are referenced by their position in the collection.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a lock array.

Inherits From: ObjectArray

Inherited By: (None)

LockContentionInfo Class

The **LockContentionInfo** class is the class that is used to retrieve information about lock contentions. A lock contention occurs when an attempt to lock a persistent object is queued because the object is already locked. Lock contention information is recorded on the database server node. By default, lock contentions are not recorded.

The **beginLockContentionStats**, **clearLockContentionStats**, and **endLockContentionStats** methods defined in the **System** class enable you control the recording of lock contentions. The **getLockContentionStats** and **getLockContentionInfo** methods defined in the **System** class enable you to retrieve recorded lock contention information.

For details about the properties and method defined in the **LockContentionInfo** class and an example of displaying lock contention information, see "LockContentionInfo Properties", "LockContentionInfo Method", and "Example of Displaying Lock Contention Information", in the following subsections.

Inherits From: Object

Inherited By: None

LockContentionInfo Properties

The properties defined in the **LockContentionInfo** class are summarized in the following table.

Property	Description
maxWaitTime	The longest time in milliseconds that any process spent queued waiting to obtain a lock on the object
totalContentions	The number of lock contentions recorded for the object
totalWaitTime	The total time in milliseconds that all processes spent queued waiting to obtain locks on the object

maxWaitTime

Type: Integer64

The **maxWaitTime** property of the **LockContentionInfo** class contains the longest time in milliseconds that any process spent queued waiting to obtain a lock on the object.

totalContentions

Type: Integer64

The **totalContentions** property of the **LockContentionInfo** class contains the number of lock contentions recorded for the object.

totalWaitTime

Type: Integer64

The **totalWaitTime** property of the **LockContentionInfo** class contains the total time in milliseconds that all processes spent queued waiting to obtain locks on the object.

LockContentionInfo Method

The method defined in the **LockContentionInfo** class is summarized in the following table.

Method	Description
target	Returns a reference to the object to which the lock contention information relates

target

Signature `target(): Object;`

The **target** method of the **LockContentionInfo** class returns a reference to the object to which the lock contention information relates.

Example of Displaying Lock Contention Information

The following example shows how to display recorded lock contention information.

```
showLockContentions();
vars
    oa : ObjectArray;
    o : Object;
    lci : LockContentionInfo;
    ts : TimeStamp;
    avgWaitTime : Decimal[10,1];
begin
    create oa transient;
    system.beginLockContentionStats(10000);
    process.sleep(60000); //record 1 minute of lock activity
    system.getLockContentionStats(oa, 10000, 5, ts);
    write "FIRST MINUTE";
    foreach o in oa do
        lci := o.LockContentionInfo;
        write CrLf & "Object= " & lci.target.String;
        write "Contentions= " & lci.totalContentions.String;
        avgWaitTime := (lci.totalWaitTime / lci.totalContentions);
        write "Average wait time= " & avgWaitTime.String;
    endforeach;
    oa.purge;
    //Repeat
    system.clearLockContentionStats();
    process.sleep(60000); //record 1 minute of lock activity
    system.getLockContentionStats(oa, 10000, 5, ts);
    write CrLf & "SECOND MINUTE";
    foreach o in oa do
        lci := o.LockContentionInfo;
        write CrLf & "Object= " & lci.target.String;
        write "Contentions= " & lci.totalContentions.String;
        avgWaitTime := (lci.totalWaitTime / lci.totalContentions);
        write "Average wait time= " & avgWaitTime.String;
    endforeach;
    oa.purge;
    system.endLockContentionStats();
```

```
epilog
    delete oa;
end;
```

The output from the **LockContentionInfo** method shown in the previous example is as follows.

```
FIRST MINUTE

Object= Branch/2631.1
Contentions= 8
Average wait time= 4516.0

Object= Account/2635.2
Contentions= 6
Average wait time= 3531.0

SECOND MINUTE

Object= Branch/2631.1
Contentions= 9
Average wait time= 2640.0

Object= Account/2635.1
Contentions= 6
Average wait time= 2250.0

Object= Account/2635.2
Contentions= 6
Average wait time= 1937.0

Object= Account/2635.3
Contentions= 8
Average wait time= 5046.0
```

LockException Class

The **LockException** class is the transient class that defines the behavior of exceptions raised as a result of locking conflicts. This class enables you to write a generic lock exception handler that can retry a lock operation.

Global lock exceptions can be handled in your logic in the following way.

```
on LockException do global.lockException(exception) global;
```

Each process can have up to 128 global exception handlers armed at any one time.

Lock exceptions are *continuable*; that is, the **continuable** property is set to **true**. A lock exception handler could therefore attempt to acquire the lock using the **tryLock** method and if successful, return **Ex_Continue**.

For details about the properties and methods defined in the **LockException** class, see "LockException Properties" and "LockException Methods", in the following subsections.

Inherits From: SystemException

Inherited By: (None)

LockException Properties

The properties defined in the **LockException** class are summarized in the following table.

Property	Description
lockDuration	Contains the duration of the lock
lockTimeout	Contains the timeout period of the lock
lockType	Contains the type of lock
retryCount	Contains the number of lock retries that were encountered
targetLockedBy	Contains the process that locked the object

lockDuration

Type: Integer

The read-only **lockDuration** property of the **LockException** class contains the duration of the lock that was encountered in a multiuser environment.

The lock durations (whose values are provided by global constants in the **LockDurations** category) that can raise exceptions are listed in the following table.

Global Constant	Integer
Persistent_Duration	2
Session_Duration	1
Transaction_Duration	0

The following example shows the use of the **lockDuration** property.

```
handleLockException(le: LockException): Integer;
//Example using tryLock to retry a lock
```

```
vars
    result  : Integer;
    message : String;
begin
    message := 'Cannot get lock for ' & le.lockTarget.String &
               '. It is locked by user ' ;
    result := app.msgBox('Lock Error', message & le.targetLockedBy.userCode
              & '. Retry?', MsgBox_Question_Mark_Icon + MsgBox_Yes_No);
    if result = MsgBox_Return_Yes then
        app.mousePointer := Busy;
        while not tryLock(le.lockTarget, le.lockType, le.lockDuration,
                          LockTimeout_Server_Defined) do
            app.mousePointer := Idle;
            result := app.msgBox('Lock Error', message &
                      le.targetLockedBy.userCode & '. Retry?',
                      MsgBox_Question_Mark_Icon + MsgBox_Yes_No);
            if result = MsgBox_Return_No then
                return Ex_Abort_Action;
            endif;
        app.mousePointer := Busy;
        endwhile;
        return Ex_Resume_Next;
    endif;
    return Ex_Abort_Action;
epilog
    app.mousePointer := Idle;
end;
```

lockTimeout

Type: Integer

The read-only **lockTimeout** property of the **LockException** class contains the timeout period of the lock that was encountered in a multiuser environment.

The timeout periods (whose values are provided by global constants in the **LockTimeouts** category) that can raise exceptions are listed in the following table.

Global Constant	Integer
LockTimeout_Immediate	-1
LockTimeout_Infinite	Max_Integer (#7FFFFFFF, equates to 2147483647)
LockTimeout_Process_Defined	-2 (use the process-defined default)
LockTimeout_Server_Defined	0 (use the server-defined default)

You can set the process-defined default lock request timeout programmatically, by calling the **Process** class **setDefaultLockTimeout** method. By default (that is, if you do *not* call this method), the default lock timeout for a process is the value of the **ServerTimeout** parameter in the [JadeServer] section of the JADE initialization file.

lockType

Type: Integer

The read-only **lockType** property of the **LockException** class contains the type of lock that was encountered in a multiuser environment. (For an example of the use of the **lockType** property, see the **LockException** class **lockDuration** property.)

The types of lock (whose values are provided by global constants in the **Locks** category) that can raise exceptions are listed in the following table.

Global Constant	Integer Value	Description
Exclusive_Lock	3	No other process can lock the same object.
Get_Lock	0	Not valid for lock requests. This lock type indicates a process is waiting to acquire a lock that will cause all other lock requests for the object to be queued (for example, when upgrading a lock from update to exclusive).
Reserve_Lock	2	When you place a **reserve** lock on an object, other processes attempting to acquire an **exclusive** lock or **reserve** lock on that same object wait until the **reserve** lock is relinquished, but those attempting to acquire a shared lock succeed.
Share_Lock	1	When you place a **shared** lock on an object, other processes attempting to update the object or explicitly acquire an **exclusive** lock wait until the lock is released but can acquire a **shared** lock or a **reserve** lock.
Update_Lock	4	Placing an **update** lock allows you to update the object, while still allowing other processes to acquire **shared** locks to view the most recently committed edition.

retryCount

Type: Integer

The read-only **retryCount** property of the **LockException** class is a work area for the user that contains the number of times that the lock encountered in a multiuser environment user application was retried.

targetLockedBy

Type: Process

The read-only **targetLockedBy** property of the **LockException** class contains a reference to the process that locked the object in a multiuser environment. (For an example of the use of the **targetLockedBy** property, see the **LockException** class **lockDuration** property.)

LockException Methods

The methods defined in the **LockException** class are summarized in the following table.

Method	Description
lockTarget	Returns the target object of the lock

Method	Description
retryLock	Retries the lock in a multiuser environment user application, increments the **retryCount** property, and returns **true** if the lock was obtained
showDialog	Displays the default lock exception dialog

lockTarget

Signature `lockTarget(): Object;`

The **lockTarget** method of the **LockException** class returns a reference to the object that is the target of the lock on which an exception is raised.

The following example shows the use of the **lockTarget** method.

```
handleLockException(le: LockException): Integer;
vars
    result  : Integer;
    message : String;
begin
    message := "Cannot get lock for " & le.lockTarget.String &
               ". It is locked by user " ;
    result  := app.msgBox(message & le.targetLockedBy.userCode & ". Retry?",
               "Lock Error", MsgBox_Question_Mark_Icon + MsgBox_Yes_No);
    if result = MsgBox_Return_Yes then
        app.mousePointer := Busy;
        while not tryLock(le.lockTarget, le.lockType, le.lockDuration,
                          LockTimeout_Server_Defined) do
            app.mousePointer := Idle;
            result := app.msgBox(message & le.targetLockedBy.userCode &
                      ". Retry?", "Lock Error", MsgBox_Question_Mark_Icon +
                      MsgBox_Yes_No);
            if result = MsgBox_Return_No then
                return Ex_Abort_Action;
            endif;
            app.mousePointer := Busy;
        endwhile;
        return Ex_Resume_Next;
    else
        return Ex_Abort_Action;
    endif;
epilog
    app.mousePointer := Idle;
end;
```

retryLock

Signature `retryLock(): Boolean;`

The **retryLock** method of the **LockException** class retries the lock in a multiuser environment user application, increments the **retryCount** property, and returns **true** if the lock was obtained.

If the lock was not obtained, this method returns **false**.

showDialog

Signature `showDialog(): Boolean;`

The **showDialog** method of the **LockException** class displays the default lock exception dialog.

If the **showDialog** method returns **true**, the action is resumed. If this method returns **false**, the action is aborted.

MemberKeyDictionary Class

The **MemberKeyDictionary** class encapsulates the behavior required to access entries in member key dictionary subclasses. Member key dictionaries are dictionaries for which the keys are properties in the member objects.

For details about the methods defined in the **MemberKeyDictionary** class, see "MemberKeyDictionary Methods", in the following subsection.

Inherits From: Dictionary

Inherited By: JadeSkinApplicationNameDict, JadeSkinCategoryNameDict, JadeSkinControlNameDict, JadeSkinEntityNameDict, JadeSkinFormNameDict, JadeSkinMenuNameDict, JadeSkinSimpleButtonNameDict, JadeSkinWindowStateNameDict, JadeUserProfileShortCutByAction, JadeUserProfileShortCutDict, JadeWebSocketDictionary, ProcessDict

With the exception of the **JadeSkinEntityNameDict** class, which allows duplicate keys, the member key dictionaries for subclasses of the **JadeSkinEntity** class are defined with one key (that is, the **name** property of the **JadeSkinEntity** class), which is case-sensitive and does not allow for duplicate keys. These dictionaries are referenced by the **JadeSkinRoot** class and are automatically maintained by inverses defined using the **mySkinRoot** property of the **JadeSkinEntity** class.

In addition, each skin entity has a **JadeSkinEntityNameDict** dictionary of other skin entities that reference that skin. This **myOwners** dictionary is automatically maintained by inverses between the referencing property and the dictionary.

For details about specifying and maintaining JADE skins, see Chapter 9 of the *JADE Developer's Reference*. See also "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

MemberKeyDictionary Methods

The methods defined in the **MemberKeyDictionary** class are summarized in the following table.

Method	Description
add	Adds an object to a dictionary
includes	Returns **true** if the member key dictionary contains a specified object
indexNear	Returns an approximate index of an object in a collection
indexNear64	Returns an approximate index of an object in a collection
purge	Deletes all object references in a member key dictionary
remove	Removes an item from a dictionary with member keys

add

Signature `add(value: MemberType);`

The **add** method of the **MemberKeyDictionary** class adds the object specified in the **value** parameter to the receiver.

If there is already an entry with the same key and the collection does not allow duplicate entries, an exception is raised.

The following example shows the use of the **add** method to populate a member key dictionary referenced by **customerDict** with 80 customer instances.

```
load() updating;
vars
    count : Integer;
    cust  : Customer;
begin
    beginTransaction;
        count := 1;
        while count < 81 do
            create cust;
            cust.key     := count;
            cust.name    := "Customer " & count.String;
            cust.address := "Address " & count.String;
            cust.phoneNo := "364589" & count.String;
            self.customerDict.add(cust);
            count := count + 1;
        endwhile;
    commitTransaction;
end;
```

includes

Signature `includes(value: MemberType): Boolean;`

The **includes** method of the **MemberKeyDictionary** class searches the dictionary using the member keys of the object and returns **true** if the object is located with its current member key values.

This method will not find an object if its key has changed since it was added to a manually maintained dictionary. (A manually maintained dictionary can contain an object with a key that differs from the value that is currently in the attribute on which the dictionary is keyed.)

This method returns **true** only if the object is in the dictionary with its current key values; for example:

```
if self.myEmployees.includes(emp) then
    return true;
else
    foreach child in self.myEmployees do
        if child.isEmployee(emp) = true then
            return true;
            break;
        endif;
    endforeach;
endif;
```

indexNear

Signature `indexNear(value: MemberType): Integer;`

The **indexNear** method of the **MemberKeyDictionary** class returns an approximate index for the entry specified in the **value** parameter if it exists in the collection or it returns zero (**0**) if it does not exist. (See also the **Iterator** class **startNearIndex** method.)

If the specified value occurs more than once in the collection, the approximate index of the first occurrence is returned.

Notes This method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf** method.

Use the **indexNear64** method instead of the **indexNear** method if the number of entries in the collection could exceed the maximum integer value of 2,147,483,647.

indexNear64

Signature `indexNear64(value: MemberType): Integer64;`

The **indexNear64** method of the **MemberKeyDictionary** class returns an approximate index for the entry specified in the **value** parameter if it exists in the collection or it returns zero (**0**) if it does not exist. (See also the **Iterator** class **startNearIndex** method.)

If the specified value occurs more than once in the collection, the approximate index of the first occurrence is returned.

Note This method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf64** method.

purge

Signature `purge();`

The **purge** method of the **MemberKeyDictionary** class deletes all objects in a member key dictionary and clears the dictionary; that is, **size = 0**.

Caution The objects that are removed are physically deleted.

The following example shows the use of the **purge** method.

```
unload() updating;
begin
    beginTransaction;
        self.customerDict.purge;
    commitTransaction;
end;
```

remove

Signature `remove(value: MemberType);`

The **remove** method of the **MemberKeyDictionary** class removes the item specified in the **value** parameter from a dictionary with member keys.

If the collection does not contain the specified item, an exception is raised.

The code fragment in the following is an example of the use of the **remove** method.

```
self.custNameDict.remove(cust);
```

MenuItem Class

The **MenuItem** class contains the definition of each menu item (command) on a menu. A menu for a form is constructed in the JADE development environment, by using the Menu Design window in the JADE Painter. You cannot add a subclass to the **MenuItem** class.

A menu can include items, submenu titles, and separator bars. A menu can have submenu items. Define menu items with names to which they can be referred at run time.

Menus are displayed in four columns: check mark, picture, text, and shortcut text. The width of the picture, text, and shortcut text columns is set to the maximum width of all the menu items in the popup menu that is being displayed.

Menus are drawn in the .NET style. A menu is drawn with a left gutter border and draws the selected background using the current Windows theme set. In addition, menus activated from the form's menu are drawn as though the form menu item is part of the activated menu. The form's menu bar is also drawn using the current Windows theme. The exception to this is when a skin is currently active for the form, in which case the skin definition of any menu elements is still used instead.

Menu items are generated as static text for HTML on Web pages. All menu items that have an associated **click** event also have a HyperText link.

Each menu item can be defined with the following set of attributes.

- A separator (all other properties are then ignored).
- A check mark can be displayed to the left of a menu item. The default value is **none**. Menu items that are popup menus or top-level items ignore this attribute.
- A picture, displayed after the check mark.
- A caption, displayed after the picture.
- A menu that is not top-level or a popup menu can have a shortcut key assigned to it, with a default value of **none**. The menu item executes its **click** event logic when the shortcut key is pressed. A textual description of the key is displayed after the caption.
- Can be enabled (the default) or disabled (the caption will be gray, or dimmed). If the item has subitems, the menu does not drop down when selected.
- Can be initially visible (the default) or hidden. If the item has subitems, then they are also hidden.
- A context-sensitive help identifier (**helpContextId**) or keyword (**helpKeyword**).
- One top-level menu item can have a standard set of help items added to it, by checking the **Help** list option. This list includes an Index entry for the help file of the application and an About box for the application. No logic is required to handle these options. These entries can be moved and deleted, but they cannot be altered. The Help menu item is moved to the end of the top menu items at run time.
- One top-level menu item can also have a standard set of items added to it for MDI window control, by checking the **Window** list option. This list includes the ability to cascade, tile, or arrange the icons, and create a new copy of an MDI form. No logic is required to handle these options. These entries can be moved and deleted, but they cannot be altered.

  These options are ignored if they are included in the menu for a non-MDI form. The Window menu item is moved to the end of the top menu items (but before any Help item) at run time. (Alternatively, you can make an MDI child form invisible if you do not want to include it in the list of currently open MDI forms.)

When you create a Multiple-Document Interface (MDI) application, the menu bar on the MDI child form replaces the menu bar on the MDI form when the child form is active.

Menu items and submenu items can be loaded at run time, by using the **loadMenu** and **loadSubMenu** methods, respectively. The loaded menu items can then be accessed by using the **getMenuItem** method.

A menu item can be deleted at run time, by using the **delete** instruction (as opposed to making it invisible). Deleted menus cannot be reinstated. If the deleted menu item is a popup menu, all members of that popup menu are also deleted.

Notes If you invoke the JADE Debugger while processing JADE menu logic, Windows discards subsequent menu actions. For example, if you break in the **click** event of a popup menu, the menu is not displayed.

An exception is raised if a **MenuItem** method is invoked from a server method.

For the arrays associated with control and menu item children (for example, the **Window** class **allControlChildren** and **MenuItem** class **children** properties), the only methods that are implemented are **at** (which allows the use of square brackets to access the elements), **createIterator** (which allows logic to do a **foreach** over the array), **size**, and **size64**.

For details about the constants, properties, methods, and events defined in the **MenuItem** class, see "MenuItem Class Constants", "MenuItem Properties", "MenuItem Methods", and "MenuItem Events", in the following subsections.

Inherits From: MenuItemData

Inherited By: (None)

MenuItem Class Constants

The constants provided by the **MenuItem** class are listed in the following table.

Constant	Bit Value	Description
ShortCutFlag_Alt	#10	The Alt key must also be pressed
ShortCutFlag_Ctrl	#8	The Ctrl key must also be pressed
ShortCutFlag_Shift	#4	The Shift key must also be pressed

MenuItem Properties

The properties defined in the **MenuItem** class are summarized in the following table.

Property	Description
allChildren	Contains an array of all children of a menu item, including children of children
caption	Contains the text displayed in the menu items caption
checked	Specifies whether a check mark is displayed next to a menu caption
children	Contains an array of immediate children of a menu item
description	Contains a textual description of the Window object
disableReason	Contains a reason for the menu item being disabled
enabled	Specifies whether the menu can respond to user-generated events
form	Contains the form on which the menu is placed

Property	Description
helpContextId	Contains an associated context number for an object
helpKeyword	Contains text used to access the help file while the menu item is selected
index	Contains an identifier to differentiate between menu items that have been created by logic
name	Contains the name used in logic to identify a menu item object
picture	Contains a graphic to be displayed in a menu
securityLevelEnabled	Specifies whether the menu is automatically disabled
securityLevelVisible	Specifies whether the menu is automatically made invisible
userObject	Contains an object to associate with the menu
visible	Specifies whether a menu is visible or hidden
webFileName	Contains the name of the image displayed in a menu on a Web page

allChildren

Type: MenuItemArray

Availability: Read at run time only

The **allChildren** property of the **MenuItem** class contains a reference to an array of all of the children of the menu item, including children of children. The collection is ordered according to the defined menu item list.

The following code fragment is an example of the use of the **allChildren** property.

```
foreach control in frameLeft.allChildren do
    if control.isKindOf(Label) then
        control.fontBold := true;
    endif;
endforeach;
```

caption

Type: String[100]

Availability: Read or write at any time

The **caption** property of the **MenuItem** class contains the text displayed in the menu item caption.

You can use the **caption** property to assign an access key to a menu. In the caption, include an ampersand character (**&**) immediately preceding the character that you want for an accelerator key. The accelerator key character is underlined. To activate that menu or item, press Alt and the key of the underlined character.

To include an ampersand in a caption without creating an access key, enter two ampersand characters (**&&**). A single ampersand is displayed in the caption and no character is underlined.

This property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

The code fragment in the following example shows the use of the **caption** property.

```
// For multilanguage, check locale and display the appropriate message string
MenuItemGreeting.caption := $Hello;
```

Tip To dynamically add a separator to a menu at run time, specify a **caption** property value of **"-"** (that is, a hyphen character) so that the item is displayed as a separator line when the menu is displayed.

The maximum length of a menu caption is 100 characters.

checked

Type: Boolean

Availability: Read or write at any time

The **checked** property of the **MenuItem** class specifies whether a check mark is displayed next to a menu caption. If the menu item has a submenu or it is a top-level menu item, the **checked** property has no effect.

The **checked** property settings are listed in the following table.

Setting	Description
true	Places a check mark next to the menu command
false	Removes a check mark from a menu command (the default value)

The code fragment in the following example shows the use of the **checked** property.

```
if menuItem1.caption = self.myTable.text then
    menuItem1.checked := false;
    return;
endif;
```

children

Type: MenuItemArray

Availability: Read at run time only

The **children** property of the **MenuItem** class contains a reference to an array of all of the immediate children of the menu item (that is, the menu item is the direct parent of the menu items in the collection). The collection is ordered according to the defined menu item list.

The following code fragment is an example of the use of the **children** property.

```
foreach menu in mnuOptions.children do
    if menu.checked then
        .... // do some processing here
    endif;
endforeach;
```

description

Type: String

Availability: Read or write at any time

The **description** property of the **MenuItem** class contains a textual description of the Window object. The description can be in the range **0** through **32,767** characters. (This description is not automatically displayed anywhere.)

Any change to the value at run time is not retained after the form on which the control (or the form itself) is unloaded.

disableReason

Type: String

Availability: Read or write at any time

The **disableReason** property of the **MenuItem** class contains a reason for the menu item being disabled.

JADE does not use this property. It is your responsibility to display the text, as appropriate.

enabled

Type: Boolean

Availability: Read or write at any time

The **enabled** property of the **MenuItem** class specifies whether the menu can respond to user-generated events.

The **enabled** property settings are listed in the following table.

Setting	Description
true	Enables the object to respond to events (the default)
false	Prevents the object from responding to events

This property allows menu items to be enabled or disabled at run time.

form

Type: Form

Availability: Read-only at run time

The **form** property of the **MenuItem** class provides access to the form on which the menu is placed. Use this property when the menu object is passed as a parameter to a generalized method, so that the logic can still access the form of the menu.

helpContextId

Type: Integer

Availability: Read or write at any time

The **helpContextId** property of the **MenuItem** class contains an associated context number for a menu item object. This property is used to provide context-sensitive help for your menu. If the **helpKeyword** property is also set, the keyword is used in preference to the context number.

For context-sensitive help on an object in your application, you must assign the same context number to both the object and to the associated help topic when you compile your help file.

If you have created a Windows environment help file for your application (that is, a .**hlp** or .**chm** file), JADE automatically calls help when a user presses F1 and requests the topic identified by the current context number (or the **helpKeyword** property).

The current context number is the value of the **helpContextId** property for the object that has the focus or the selected menu item. If this property is set to zero (and its **helpKeyword** property value is **null**), JADE looks in the **helpContextId** property (and the **helpKeyword** property) of the form of the object. If a non-zero current context number cannot be found, the Contents section of the help file is requested. If the **helpFile** property of the **Application** class is not set, no help file is opened.

Note Building a help file requires the Adobe Acrobat application, Microsoft Windows Help Compiler, or any other Windows help compiler.

helpKeyword

Type: String

Availability: Read or write at any time

If a help keyword is provided for a menu, the **helpKeyword** property of the **MenuItem** class contains text that is used to access the help file when the user presses F1 for help while the menu item is selected.

The current keyword is the value of the **helpKeyword** property for the object that has the focus or a selected menu item. If the **helpKeyword** property is empty and its **helpContextId** property is set to zero (**0**), JADE looks in the **helpKeyword** property (and the **helpContextId** property) of the form of the object. If no help keyword or context number can be found, the Contents section of the help file is requested. If the **helpFile** property of the **Application** class is not set, no help file is opened. If the **helpContextId** property is also set, the keyword is used in preference to the context number.

This property can be translated when the value of the **Schema** class **formsManagement** property is **FormsMngmt_Single_Multi** (**2**).

When help is requested, if the help file specifies a:

- Portable Document Format (PDF) file (detected by the .**pdf** file suffix), JADE attempts to execute Adobe Acrobat to handle the file. JADE checks the Windows registry for the Acrobat Reader (**AcroRd32**) or for the **acrobat** executable program. If Adobe Reader is not found, the help request is ignored and entries explaining the cause of the failure are output to the **jommsg.log** file. If Adobe Reader is located, it is initiated for the PDF help file defined in JADE.

  For a **helpKeyword** help request, the **helpKeyword** property is passed to Acrobat as a **named destination**, which Acrobat uses to position the help file display. As there are no equivalent concepts in a PDF file to any other type of help request (for example, **helpContextId**, index request, and so on), only the first page of the PDF file is displayed for help requested using anything other than the **helpKeyword** property.

- Windows help file (detected by the .**hlp** file suffix), JADE automatically calls help and requests the topic identified by the current **helpKeyword** property or the **helpContextId** property.
- Compiled help file (detected by the .**chm** file suffix), JADE calls the **HtmlHelp** entry point of the **htmlhelp.dll** file and requests the topic identified by the current **helpKeyword** property or the **helpContextId** property. You can use the compiled help file (.**chm**) format files when producing online help for HTML thin client applications, for example.

For more details, see "Creating Context Links to Your Own Application Help File", in Chapter 2 of the *JADE Development Environment User's Guide*.

Note Building a help file requires the Adobe Acrobat application, Microsoft Windows Help Compiler, or any other Windows help compiler.

index

Type: Integer

Availability: Read-only at run time only

The **index** property of the **MenuItem** class is used only when menu items are created by logic; that is, when they are cloned by calling the **loadMenu** or **loadSubMenu** method.

Logic calling the **loadMenu** or **loadSubMenu** method passes a unique identifier (id) that is assigned to the created menu, which is usually an index value stored in the **index** property. As this property is read-only, any attempt to change the value is rejected.

You can also use the value of the **index** property to distinguish between menu items passed to the same menu **click** or **select** event method defined for the base menu item and used by all menu items cloned from that menu. Most commonly, you would assign the **index** values sequentially, using them like indexes.

The **index** property of menus created in the JADE Painter is set to zero (**0**).

name

Type: String[100]

Availability: Read or write at design time, read-only at run time

The **name** property of the **MenuItem** class contains the name used in logic to identify a menu item object.

Menu items are defined in the JADE database as properties, and the first character of the name is converted to a lowercase character.

A **name** property of an object must start with a letter, with a maximum length of 100, although 7 characters are reserved by JADE. This property can include numbers and underscore characters, but it cannot include punctuation symbols or spaces.

Subclassed forms cannot have menus with the same name as a control or menu on a superclass of the form.

picture

Type: Binary

Availability: Read or write at any time

The **picture** property of the **MenuItem** class contains a graphic to be displayed in a menu.

The **picture** property settings are listed in the following table.

Setting	Description
(none)	No picture (the default).
Any valid picture format (that is, .**bmp**, .**cur**, .**ico**, .**jpg**, .**png**, .**wmf**, .**gif**, and so on)	Specifies a graphic. You can load the graphic from the **Picture** list box in the **Menu Item** sheet of the JADE development environment Menu Design dialog. At run time, you can set this property, by using the **loadPicture** method on a bitmap, icon, metafile, or other valid picture.

When setting the **picture** property from JADE Painter, the graphic is saved and loaded with the form. When you load a graphic at run time, the graphic is not saved with the application. The graphic can be set by setting the **picture** property of the control to the **picture** property of another control, or by using the **loadPicture** method.

For menus, the picture can be any valid picture format; for example, a bitmap, icon, cursor, or metafile. The picture is drawn at actual size, except for a metafile, which is scaled to fit the menu line size. Menus are drawn in four columns, as follows.

```
checkMark : picture : text : accelerator text
```

The width of each column is defined to be the maximum of all the displayed items in that popup menu.

See also the **picture** property defined in subclasses of the **Window** class.

securityLevelEnabled

Type: Integer

Availability: Read or write at any time

The **securityLevelEnabled** property of the **MenuItem** class determines whether the menu is automatically disabled when its form is created and loaded or when this property is changed.

If the value of the **securityLevelEnabled** property of the menu is greater than the value of the **userSecurityLevel** property of the **Application** class (that is, **app.userSecurityLevel**), it is disabled regardless of the value of its **enabled** property when it is created.

securityLevelVisible

Type: Integer

Availability: Read or write at any time

The **securityLevelVisible** property of the **MenuItem** class determines whether the menu is automatically made invisible when its form is created and loaded or when this property is changed.

If the value of the **securityLevelVisible** property of the menu is greater than the value of the **userSecurityLevel** property of the **Application** class (that is, **app.userSecurityLevel**), it is made invisible regardless of the value of its **visible** property when it is created.

userObject

Type: Object

Availability: Read or write at run time only

The **userObject** property allows you to associate an object with any object of the **MenuItem** class.

This is a run time-only property that is not used by any JADE process. It is defined only for your convenience. The default value for the **userObject** property is **null**.

visible

Type: Boolean

Availability: Read or write at any time

The **visible** property of the **MenuItem** class specifies whether a menu is visible or hidden. Hiding an item with submenu items also hides all of the subitems.

The settings of the **visible** property are listed in the following table.

Setting	Description
true	The menu is visible (the default)
false	The menu is hidden

To hide a menu at start up, set the **visible** property to **false** in the JADE development environment. Setting this property in logic enables you to hide and later redisplay a menu at run time in response to a specific event.

webFileName

Type: String

Availability: Read or write at any time

The **webFileName** property of the **MenuItem** class contains the name of the image that is to be displayed in a menu on the Web page; for example, "**image.jpg**" or "**mypic.png**".

Tip Use this property for static images, as performance is greatly improved.

MenuItem Methods

The methods defined in the **MenuItem** class are summarized in the following table.

Method	Description
getLevel	Returns the level of the menu item
getMenuItem	Accesses a dynamically created menu item
loadMenu	Dynamically creates a menu item
loadSubMenu	Dynamically creates a submenu item
setEventMapping	Dynamically sets the method executed for a menu event at run time
setEventMappingEx	Dynamically sets the method executed for a menu event at run time
setShortCutKey	Dynamically sets the shortcut key displayed for most development environment and editor shortcut keys to values of your choice

Note An exception is raised if event methods in this class are invoked from a server method.

getLevel

Signature `getLevel(): Integer;`

The **getLevel** method of the **MenuItem** class returns the level of the menu item.

An exception is raised if a **MenuItem** method is invoked from a server method.

getMenuItem

Signature `getMenuItem(id: Integer): MenuItem;`

The **getMenuItem** method of the **MenuItem** class accesses a dynamically created menu item. This method returns **null** if there is no menu for the unique identifier of the menu specified in the **id** parameter.

An exception is raised if a **MenuItem** method is invoked from a server method.

loadMenu

Signature `loadMenu(index: Integer): MenuItem;`

The **loadMenu** method of the **MenuItem** class enables an existing menu item to be "cloned" at run time; that is, one or more copies of that menu item can be created at run time in the menu of the form.

The cloned menu can be a menu item created at run time (to control ordering) or it can be one created in JADE Painter.

The menu item is added to the menu directly after the menu item that is being cloned. New clone menu items are created using the runtime copy of the menu item.

Each menu item calls the same methods defined for the original menu item, passing their own menu item object as the first parameter. In addition, you must assign a unique identifier to each menu item, which is passed to the **loadMenu** method in the **index** parameter.

Menu items that are created in the JADE Painter have an **index** parameter value of zero (**0**).

Most commonly, the value of the index would be just that: an index. The values need not be sequential, but they cannot be duplicated.

An exception is raised if a **MenuItem** method is invoked from a server method.

The code fragment in the following example shows the creation of a new menu item.

```
menu := menuItem.loadMenu(count);
```

In this example, the **menuItem** value is the menu item that is being cloned, the **menu** value is the new copy of the menu item created, and the **count** value is the unique copy index supplied by the caller.

Any menu item that is added is automatically deleted when the form is destroyed. You can also delete these menu items dynamically, by using the **delete** instruction. To access these cloned menu items, use the **getMenuItem** method.

loadSubMenu

Signature `loadSubMenu(index: Integer): MenuItem;`

The **loadSubMenu** method of the **MenuItem** class enables an existing submenu item to be "cloned" at run time; that is, one or more copies of that submenu item can be created at run time in the submenu of the form as subitems of that menu item.

The cloned submenu can be a submenu item created at run time (to control ordering) or it can be one created in JADE Painter.

The submenu item is added to the end of existing subitems of the menu that is being cloned or as the first subitem if that menu did not previously have any subitems. New clone submenu items are created using the runtime copy of the submenu item.

Each submenu item calls the same methods defined for the original submenu item, passing their own submenu item object as the first parameter. In addition, you must assign a unique identifier to each submenu item, which is passed to the **loadSubMenu** method in the **index** parameter.

Submenu items that are created in the JADE Painter have an **index** parameter value of zero (**0**).

Most commonly, the value of the **index** parameter would be just that: an index. The values need not be sequential, but they cannot be duplicated.

An exception is raised if a **MenuItem** method is invoked from a server method.

The code fragment in the following example shows the creation of a new submenu item.

```
submenu := myTestSubmenu.loadSubMenu(count);
```

In this example, the **myTestSubmenu** value is the submenu item that is being cloned, the **submenu** value is the new copy of the submenu item created, and the **count** value is the unique copy index supplied by the caller.

Any submenu item that is added is automatically deleted when the form is destroyed. You can also delete these submenu items dynamically, by using the **delete** instruction. To access these cloned submenu items, use the **getMenuItem** method.

setEventMapping

Signature
```
setEventMapping(eventName:  String;
                mappedName: String);
```

The **setEventMapping** method of the **MenuItem** class enables the method that is to be executed for an event to be dynamically set at run time. (See also the **setEventMapping** method of the **Window** class.)

Tip This method is equivalent to the **setEventMappingEx** method but it is less efficient, as it must find the methods by name. You should therefore use the **setEventMappingEx** method to improve performance.

An exception is raised if a **MenuItem** method is invoked from a server method.

By default, the JADE development environment allows the definition of event methods (*menu-name_event-name*) for a menu; for example:

```
mTickerTimer_click(menuItem: MenuItem input) updating;
```

The parameters of the **setEventMapping** method are listed in the following table.

Setting	Description
eventName	Must be a defined event name for the menu; for example, **click**.
mappedName	The name of the method that is to be called. This method must exist on the form that is the parent of the menu or window for which you are calling the **setEventMapping** method.

The code fragment in the following example shows the use of the **setEventMapping** method.

```
mTickerTimer.setEventMapping("click", "myOtherMenu");
```

The method checks that:

- The event method is valid for the menu
- The method to be called exists
- The signature of the method matches the event method signature

setEventMappingEx

Signature `setEventMappingEx(eventMethod: Method;`
`mappedMethod: Method);`

The **setEventMappingEx** method of the **MenuItem** class enables the method that is to be executed for an event to be dynamically set at run time and the mapping cached on each JADE node. (See also the **setEventMappingEx** method of the **Window** class.)

An exception is raised if a **MenuItem** method is invoked from a server method.

Repeat calls for a mapping that has been previously used is recognized and the signature check is not repeated unless the timestamp of the mapped method has changed since the previous signature check. The cost of reloading a form that assigns event mappings is therefore subsequently less expensive on that node. If an application server is involved, only the first assignment by any user performs the signature check. Subsequent repeat calls for any user on that application server avoid that overhead.

The event method being mapped and the mapped method are passed as parameters. The underlying logic, therefore, does not have to find the methods by name, making the execution more efficient; for example:

```
mnuCustomer.setEventMethodEx(MenuItem::click, CustomerDialog::customerMenu);
```

Tip This method is equivalent to the **setEventMapping** method but as it is more efficient, you should use the **setEventMappingEx** method to improve performance.

The parameters of the **setEventMappingEx** method are listed in the following table.

Setting	Description
eventMethod	Specifies the event method, which must belong to the class of the receiver of the **setEventMappingEx** call, and must be a defined event name for the menu; for example, **click**.
mappedMethod	The method that is to be called. This method must exist on the form that is the parent of the menu or window for which you are calling the **setEventMappingEx**.

The code fragment in the following example shows the use of the **setEventMappingEx** method.

```
mTickerTimer.setEventMapping(Button::click, MyForm::myOtherMenu);
```

The method checks that:

- The event method is valid for the menu
- The method to be called exists
- The signature of the method matches the event method signature

setShortCutKey

Signature `setShortCutKey(key: Character; flags: Integer);`

The **setShortCutKey** method of the **MenuItem** class enables you to set the shortcut key displayed for a menu item at run time in your JADE system with the values specified in the **key** and **flags** method parameters.

The value of the **key** parameter must be one of **"0"** through **"9"**, **"A"** through **"Z"**, **J_key_F1** though **J_key_F12**, **J_key_Delete**, **J_key_Insert**, **J_key_Back**, **J_key_UpArrow**, or **J_key_DownArrow**. (The **J_key_** values are global constants in the **KeyCharacterCodes** category.)

The value of the **flags** parameter can be zero (**0**) if there is no shortcut flag or it can be a combination of the following **MenuItem** class constants.

Constant	Bit Value	Description
ShortCutFlag_Alt	#10	The Alt key must also be pressed
ShortCutFlag_Ctrl	#8	The Ctrl key must also be pressed
ShortCutFlag_Shift	#4	The Shift key must also be pressed

The method generates an exception if the parameters are invalid.

The code fragment in the following example displays Shift+Ctrl+Alt+Delete as the menu accelerator.

```
menu1.setShortCutKey(J_key_Delete.Character, MenuItem.ShortCutFlag_Alt +
                     MenuItem.ShortCutFlag_Ctrl + MenuItem.ShortCutFlag_Shift);
```

MenuItem Events

The events defined in the **MenuItem** class are summarized in the following table.

Event	Description
click	Occurs when the user presses and then releases the left mouse button over a menu item or when the menu item is activated by a control sequence or an attached accelerator key
select	Occurs when a menu item is highlighted

Note An exception is raised if the event methods in this class are invoked from a server method.

click

Signature `click(menuItem: MenuItem input);`

The **click** event of the **MenuItem** class occurs when the user presses and then releases the left mouse button over a menu item or when the menu item is activated by a control sequence or an attached accelerator key. If the menu item has a submenu, logic in this event allows the contents of the submenu to be changed before it becomes visible.

An exception is raised if a **MenuItem** method is invoked from a server method.

Typically, you attach a **click** event to a menu to carry out commands and command-like actions.

The following is an example of the event definition.

```
menuEmpList_click(menuItem: MenuItem) updating;
vars
    form : ListEmp;
begin
    create form;
    list.show;
end;
```

select

Signature
```
select(menuItem: MenuItem input;
       closed:   Boolean);
```

The **select** event of the **MenuItem** class is generated for a menu item when the user selects or deselects a menu item. The **select** event occurs when menu item is highlighted. When another menu item is selected or the menu operation is completed (or cancelled), the previously selected menu item has another **select** event generated, indicating that it was deselected.

The **closed** parameter is set to **true** when the menu item is deselected and to **false** when it is selected.

An exception is raised if a **MenuItem** method is invoked from a server method.

The following is an example of the event definition.

```
menuEmpList_select(menuItem: MenuItem;
                   closed:   Boolean);
begin
    if not closed then
        StatusLine.caption := "List of Employees";
    else
        StatusLine.caption := "";
    endif;
end;
```

In this example, the **menuEmpList** is the menu item that is clicked. The **closed** parameter returns **false** for the initial select call and **true** for the deselect call.

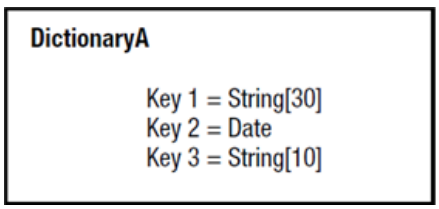
This event is designed for the developer to be able to display help text about the menu item while the user considers whether to click the item. The deselect event call enables that help text to be cleared.

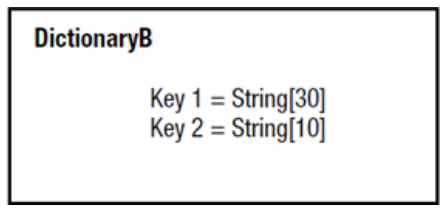
The **select** event also occurs for disabled menu items when they are selected; for example, by holding the left mouse button down and dragging over those items.

MergeIterator Class

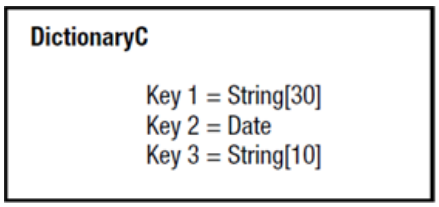
The **MergeIterator** class encapsulates the behavior required to sequentially access objects from a merged view of two or more compatible dictionary instances. Dictionary instances need not have the same membership but must have at least the first key in common.

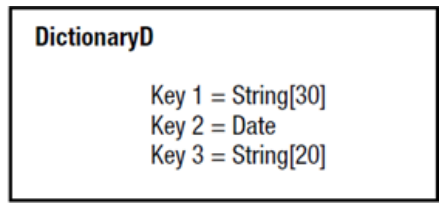
In the first example, only the first keys of **DictionaryA** and **DictionaryB** are compatible.





In the second example, the first two keys of **DictionaryC** and **DictionaryD** are compatible.





When iterating multiple dictionaries, the merged iterator returns objects in key sequence for the compatible keys.

To iterate a single collection, the iterator is created and associated with the collection by using the **createIterator** method on the collection object. To iterate a merged view of more than one collection, first create the iterator and the **addCollection** method called for each dictionary to be attached to the iterator, as shown in the following example.

```
vars
    iter : MergeIterator;
    dict1, dict2 : CustomsByNameAndAddress;
    cust : Customer;
begin
    // Assign dict1 and dict2
    create iter transient;
    iter.addCollection(dict1);
    iter.addCollection(dict2);
    while iter.next(cust) do
        write cust.name;
    endwhile;
end;
```

For details about the property and methods defined in the **MergeIterator** class, see "MergeIterator Property" and "MergeIterator Methods", in the following subsections.

Inherits From: Iterator

Inherited By: (None)

MergeIterator Property

The property defined in the **MergeIterator** class is summarized in the following table.

Property	Description
ignoreDuplicates	Skips duplicate entries in the merged iterator view

ignoreDuplicates

Type: Boolean

Default Value: True

The **ignoreDuplicates** property of the **MergeIterator** class specifies whether duplicate entries in the merged view should be skipped when iterating using the **next** and **back** methods.

Duplicate entries can occur in the merged view when an object is included in more than one of the attached dictionaries.

MergeIterator Methods

The methods defined in the **MergeIterator** class are summarized in the following table.

Method	Description
addCollection	Adds the specified dictionary to the merged iterator view
back	Accesses entries in reverse order in the merged iterator view
current	Returns the last value iterated by the **back** or **next** method
getCollectionAt	Returns the dictionary at the specified index in the collection of dictionaries making up the merged iterator view
getCollectionCount	Returns the number of dictionaries
getCurrentCollection	Returns the dictionary containing the last value iterated by the **back** or **next** method
getCurrentKey	Retrieves a single key from a dictionary while iterating through the merged iterator view
getCurrentKeys	Retrieves keys from a dictionary while iterating through the merged iterator view
isValid	Returns **true** if the receiver is a valid iterator
next	Accesses successive entries in the merged iterator view
removeCollection	Removes the specified dictionary from the merged iterator view
reset	Initializes the iterator
startAtObject	Sets the starting position of the iterator at the position of the specified object
startKeyGeq	Sets a start position within the merged iterator view at the object equal to or after the specified key
startKeyGtr	Sets a start position within the merged iterator view at the object after the specified key
startKeyLeq	Sets a start position within the merged iterator view at the object equal to or before the specified key

Method	Description
startKeyLss	Sets a start position within the merged iterator view at the object before the specified key

addCollection

Signature `addCollection(dict: Dictionary);`

The **addCollection** method of the **MergeIterator** class adds the collection specified by the value of the **dict** parameter to the merged iterator view.

The parameter value must be a **Dictionary** type with a membership compatible with existing collections associated with the iterator. When multiple dictionaries are added to the iterator, they must also have a common compatible subset of keys.

An exception is raised if you attempt to add a dictionary that is already attached to the iterator and therefore part of the merged iterator view.

back

Signature `back(value: Any output): Boolean;`

The **back** method of the **MergeIterator** class accesses entries in reverse order one at a time in the dictionaries comprising the merged iterator view.

This method returns **true** when a prior entry is found, and the entry is assigned to the **value** parameter. It returns **false** when a prior entry is not found because the iterator is positioned before the first entry in the merged view, and the **value** parameter becomes a **null** reference.

When the **back** method is used with an iterator where that iterator has been passed to a method as a method parameter, the iterator must be defined as a usage input; that is, the iterator cannot be modified by the called method.

The following example shows the use of the **back** method.

```
getReversedPosition(pObj: Object; pIter: MergeIterator input): Integer;
vars
    pos : Integer;
    obj : Object;
begin
    while pIter.back(obj) do
        pos := pos - 1;
        if obj = pObj then
            return pos;
        endif;
    endwhile;
    return 0;
end;
```

current

Signature `current(value: Any output): Boolean;`

The **current** method of the **MergeIterator** class returns the last value iterated by using the **back** or **next** method.

This method returns **true** if the iterator is positioned on an entry in the merged view, or it returns **false** if the iterator is reset or it is positioned beyond the start or end of the merged view. The **value** parameter receives the entry of the current iterator position in the merged view.

getCollectionAt

Signature `getCollectionAt(index: Integer): Dictionary;`

The **getCollectionAt** method of the **MergeIterator** class returns the dictionary at the index position specified by the **index** parameter in the array of collections attached to the iterator.

getCollectionCount

Signature `getCollectionCount(): Integer;`

The **getCollectionCount** method of the **MergeIterator** class returns the number of dictionaries that have been attached to the iterator by using the **addCollection** method.

getCurrentCollection

Signature `getCurrentCollection(): Dictionary;`

The **getCurrentCollection** method of the **MergeIterator** class returns the dictionary containing the last value iterated by using the **back** or **next** method.

getCurrentKey

Signature `getCurrentKey(ordinal: Integer): Any;`

The **getCurrentKey** method of the **MergeIterator** class retrieves the keys from a dictionary while iterating through the merged view and returns the value of a single key at the current position.

This method can be used to access the keys of an external key dictionary or to access key properties in a member key dictionary directly from the dictionary without having to access the member object itself. The **ordinal** parameter specifies the relative key by ordinal position of the iterator in the associated dictionary and should be a number in the range 1 through the number of keys in the dictionary.

When you use this method for filtering based on key conditions or populating list views with key data, judicious use of this method may result in performance improvements. (Performance improvements occur when you can avoid fetching objects from the server to access key properties.)

getCurrentKeys

Signature `getCurrentKeys(keys: ParamListType output);`

The **getCurrentKeys** method of the **MergeIterator** class retrieves one or more keys at the current iterator position in the merged view.

This method can be used to access the keys of an external key dictionary or to access key properties in a member key dictionary from the iterator without having to access the member object itself.

The method can be called with a partial key list; for example, when iterating dictionaries with three keys, you can pass one, two, or three parameters to receive the output. The parameters must be of the same type as the keys or of type **Any**. If the parameter types do not match the key types or are not of type **Any**, a runtime exception is raised.

The following example shows the use of the **getCurrentKeys** method.

```
demoGetKeys(pIter: MergeIterator): Integer;
vars
    iter : Iterator;
    cust : Customer;
    name, address : String;  // variables to receive dictionary key values
begin
    while pIter.next(cust) do
        // retrieve the first key
        iter.getCurrentKeys(name);
        // retrieve the first two keys
        iter.getCurrentKeys(name, city);
    endwhile;
end;
```

When you use the **getCurrentKeys** method for filtering based on key conditions or populating list views with key data, judicious use of this method may result in performance improvements. (Performance improvements occur when you can avoid fetching objects from the server to access key properties.)

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **MergeIterator** class returns **true** if the receiver is a valid iterator for all of the dictionaries in the merged view.

next

Signature `next(value: Any output): Boolean;`

The **next** method of the **MergeIterator** class accesses successive entries in the dictionaries comprising the merged iterator view.

This method returns **true** when a next entry is found, and the entry is assigned to the **value** parameter. It returns **false** when a next entry is not found because the iterator is positioned after the last entry in the merged view, and the **value** parameter becomes a **null** reference.

When the **next** method is used with an iterator where that iterator has been passed to a method as a method parameter, the iterator must be defined as a usage input; that is, the iterator cannot be modified by the called method.

The following example shows the use of the **next** method.

```
getPosition(pObj: Object; pIter: MergeIterator input): Integer;
vars
    pos : Integer;
    obj : Object;
begin
    while pIter.next(obj) do
        pos := pos + 1;
        if obj = pObj then
            return pos;
        endif;
    endwhile;
```

```
        return 0;
    end;
```

removeCollection

Signature `removeCollection(dict: Dictionary)`

The **removeCollection** method of the **MergeIterator** class removes the dictionary specified by the **dict** parameter from the array of dictionaries associated with the iterator.

An exception is raised if you attempt to remove a dictionary that is not attached to the iterator and therefore is not part of the merged iterator view.

reset

Signature `reset() updating;`

The **reset** method of the **MergeIterator** class restarts an iteration. After executing this method, the following **next** method would start from the first entry in the merged view; that is, it would apply to all dictionaries. Similarly, the **back** method would start from the last entry in the merged view.

startAtObject

Signature `startAtObject(object: Object) updating;`

The **startAtObject** method of the **MergeIterator** class sets the starting position of the iterator for the merged view at the position of the object specified in the **object** parameter.

An exception is raised if this object is not compatible with the membership of the collection being iterated.

Note If a collection does not allow duplicates keys and the **startAtObject** method is called with an object that is not in the collection but the object has the same keys as an object that is in the collection, the iterator will be positioned to return the object with that key in the collection when either the **next** or **back** method is called. If the **next** method is called, the object will be returned even if the instance identifier is less than the instance identifier of the **startAtObject** method **object** parameter value. If the **back** method is called, the object will be returned even if the instance identifier is greater than the instance identifier of the **startAtObject** method **object** parameter value.

If a collection allows duplicates keys and the **startAtObject** method is called with an object that is not in the collection but the object has the same keys as one or more objects that are in the collection, the instance identifier of the object passed to the **startAtObject** method is taken into account when positioning the iterator. Only the objects in the collection with an instance identifier greater than the object identifier of the **startAtObject** method will be returned for the **next** method and less than the object identifier of the **startAtObject** method for the **back** method.

---

startKeyGeq

Signature `startKeyGeq(keys: ParamListType);`

The **startKeyGeq** method of the **MergeIterator** class sets a start position of the iterator for the merged view at the object equal to or after the key specified in the **keys** parameter. This method is used in conjunction with the **next** method.

startKeyGtr

Signature `startKeyGtr(keys: ParamListType);`

The **startKeyGtr** method of the **MergeIterator** class sets a start position of the iterator for the merged view at the object after the key specified in the **keys** parameter. This method is used in conjunction with the **next** method.

startKeyLeq

Signature `startKeyLeq(keys: ParamListType);`

The **startKeyLeq** method of the **MergeIterator** class sets a start position of the iterator for the merged view at the object equal to or before the key specified in the **keys** parameter. This method is used in conjunction with the **back** method.

startKeyLss

Signature `startKeyLss(keys: ParamListType);`

The **startKeyLss** method of the **MergeIterator** class sets a start position of the iterator for the merged view at the object before the key specified in the **keys** parameter. This method is used in conjunction with the **back** method.

MethodCallDesc Class

The **MethodCallDesc** class provides information at run time about currently active method calls. It is the abstract class that defines the behavior of the **MethodCallDesc** objects that contain a reference to prior method call descriptions.

MethodCallDesc objects are used to represent the execution history of a JADE application thread and they are linked together in a stack. These objects are created only on demand, when it is necessary to take a "snapshot" of the current execution stack.

Use the **currentStack** method of the **Process** class to obtain the call stack for the current process. In addition, when an exception is raised, a **MethodCallDesc** object is attached to the **Exception** object, to represent the method and position where the exception was raised. **ObjMethodCallDesc** objects are created when the receiver is an instance of a class. **PrimMethodCallDesc** objects are created when the receiver is a primitive value.

For details about the properties and methods defined in the **MethodCallDesc** class, see "MethodCallDesc Properties" and "MethodCallDesc Methods", in the following subsections.

Inherits From: Object

Inherited By: ObjMethodCallDesc, PrimMethodCallDesc

MethodCallDesc Properties

The properties defined in the **MethodCallDesc** class are summarized in the following table.

Property	Contains the …
invocationMode	Mode in which the method was sent its message
method	Method executing in the context of the "stack frame" represented by the method call description
position	Position in the source of a method where an operation occurred

invocationMode

Type: Character[1]

The **invocationMode** property of the **MethodCallDesc** class contains the mode in which the method was sent its message. The following example shows the use of this method to display the mode.

```
vars
    mod : MethodCallDesc;
begin
    ...                     // do some processing here
    write mod.invocationMode.Integer.String;
    ...
end;
```

The values of the **invocationMode** property are listed in the following table.

Value	Message sent…
0	From another method, or message was the result of an event

Value	Message sent...
1	To the constructor of the object (that is, the **create** method of the object)
2	To the destructor of the object (that is, the **delete** method of the object)
3	To a mapping method to retrieve an attribute value
4	To a mapping method to alter an attribute value

method

Type: Method

The **method** property of the **MethodCallDesc** class contains a reference to the method executing in the context of the "stack frame" represented by the method call description.

position

Type: Integer

The **position** property of the **MethodCallDesc** class contains the position in the source of a method at which an operation occurred, such as a send message or get or set property that requires a call to the JADE kernel. (For external methods, this is the line number.)

MethodCallDesc Methods

The methods defined in the **MethodCallDesc** class are summarized in the following table.

Method	Description
getName	Returns the name of the current receiver
getReceiver	Returns the receiver
logSelf	Appends a description of the exception object to a file

getName

Signature `getName(): String;`

The **getName** method of the **MethodCallDesc** class returns a string representing the name of the current method in the stack.

getReceiver

Signature `getReceiver(): Object;`

The **getReceiver** method of the **MethodCallDesc** class is an abstract method that returns a reference to the receiver object.

logSelf

Signature `logSelf(logFileName: String);`

The **logSelf** method of the **MethodCallDesc** class enables you to log diagnostic information from an exception handler.

This method appends a description of the receiver to the file specified in the **logFileName** parameter.

MultiMediaType Class

The **MultiMediaType** class encapsulates the behavior for all types of multimedia subclasses; for example, sound and video.

For details about the property defined in the **MultiMediaType** class, see "MultiMediaType Property", in the following subsection.

Inherits From: Object

Inherited By: Sound

MultiMediaType Property

The property defined in the **MultiMediaType** class is summarized in the following table.

Property	Description
usePresentationFileSystem	Specifies whether the **Sound** class **loadFromFile** method is processed on the application server or presentation client when the receiver is running in JADE thin client mode

usePresentationFileSystem

Type: Boolean

Default Value: True

The **usePresentationFileSystem** property of the **MultiMediaType** class specifies whether the **Sound** class **loadFromFile** method is processed on the application server or presentation client when the receiver is running in JADE thin client mode.

If you are not running in JADE thin client mode, the value of this property has no effect.

This property is set to **true** by default, when running in JADE thin client mode. Set this property to **false** to cause the file system where the application server is running to be used when running the application in JADE thin client mode.

Any change to this property is ignored if the file has already been opened.

The **Sound** class **loadFromFile** method is processed on the presentation client when the **MultiMediaType** class **usePresentationFileSystem** property is set to **true**, *including* shared transient instances of the **Sound** class. The **loadFromFile** method loads the data from the file into the object, which can then be played by any user of that object.

Although a file opened on one presentation client cannot be accessed by another client, as the file access occurs only on the load (which could be from the presentation client), that file is not used thereafter.

NamedPipe Class

The **NamedPipe** class, a subclass of the **Connection** class, provides a generalized interface for communicating with external systems. The **NamedPipe** class uses the Windows Named Pipe feature to establish a two-way communication channel between a JADE process and another JADE or non–JADE process.

One process must offer the server end of the Named Pipe channel, and another process can then connect to the client end of the channel. After the connection is made and while it remains valid, both sides of the pipe have equal status (that is, the terms *server* and *client* do not apply).

Note The client node has the same software and hardware requirements as a JADE presentation client and the server node has the same software and hardware requirements as a JADE standard client. For details about the current operational requirements, see "Software Requirements" and "Hardware Requirements", in Chapter 1 of the *JADE Installation and Configuration Guide*.

The **NamedPipe** class objects are transient. If an attempt is made to create a persistent **NamedPipe** object, an exception is raised. Multiple instances of the pipe can be opened, by running multiple copies of the JADE application from the same **jade.exe** executable program, where each application opens the same pipe name.

The **NamedPipe** class supports both synchronous and asynchronous operations, as follows.

- Synchronous methods have no defined timeout mechanism at the **NamedPipe** class level. These operations will wait forever for completion.
- Asynchronous methods have a receiver object and a message (method name) specified as parameters.

  When the method (I/O) completes, the specified (callback) method of the object is called. The callback method must match the signature required by the calling asynchronous method.

Only one synchronous or asynchronous read operation can be in effect at each end of each instance of the pipe. Multiple asynchronous write operations can be in effect.

Opening the server end of the pipe waits until the other end of the pipe is connected. Opening the client end of the pipe fails immediately if the server end of the pipe has not been offered.

The **timeout** property and the **listenContinuous** and **listenContinuousAsynch** methods reimplemented from the **Connection** superclass are not supported for the **NamedPipe** class. An exception is raised when attempting to use these methods.

For details about the property and methods defined in the **NamedPipe** class, see "NamedPipe Property" and "NamedPipe Methods", in the following subsections.

Inherits From: Connection

Inherited By: InternetPipe

NamedPipe Property

The property defined in the **NamedPipe** class is summarized in the following table.

Property	Description
serverName	Contains the name of the server workstation

serverName

Type: String[128]

Availability: Read or write at any time

For the client end of the named pipe, the **serverName** property of the **NamedPipe** class contains the name of the server on which the named pipe connection is offered; for example, **"JADE_Dev_2"**.

This property must be set before the connection **open** or **openAsynch** method is attempted.

By default, this property contains a null string (**""**), indicating that the client and server ends of the connection are on the same workstation.

NamedPipe Methods

The methods defined in the **NamedPipe** class are summarized in the following table.

Method	Description
close	Closes a connection to a remote application
closeAsynch	Closes a connection to a remote application and returns immediately
getMaxMessageSize	Gets the maximum message size that can be sent or received at one time
listen	Offers a connection to a remote application and returns when established
listenAsynch	Offers a connection to a remote application and returns immediately
open	Attempts to open the client end of a named pipe connection
openAsynch	Attempts to open a connection to a named pipe and returns immediately
readBinary	Reads binary data from the connection and returns when the data has been read
readBinaryAsynch	Initiates a read of binary data from the connection and returns immediately
writeBinary	Writes binary data to the connection and returns when the operation is complete
writeBinaryAsynch	Initiates a write of binary data to the connection and returns immediately

close

Signature `close();`

The **close** method of the **NamedPipe** class closes a connection to a remote application and returns when the connection is closed. The closure occurs immediately for a named pipe, and there is no delay. This method can be called when the connection is in any state.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

The other end of the pipe is notified the next time that it performs a read or write operation on the pipe, or if an asynchronous operation is currently in progress. The pipe is then automatically closed; that is, the value of the **Connection** superclass **state** property is set to **Disconnected** (**0**).

You can reopen a closed pipe again by using the **listen**, **listenAsynch**, **open**, or **openAsynch** method. You should delete the pipe instance when you have finished with it.

Caution When an application is terminated, failure to close a pipe when there are asynchronous operations in progress may result in a fatal crash.

closeAsynch

Signature `closeAsynch(receiver: Object; msg: String);`

The **closeAsynch** method of the **NamedPipe** class closes a connection to a named pipe and returns immediately. When the connection is closed, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter. The **closeAsynch** method can be called when the connection is in any state.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

When the **closeAsynch** method completes, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **closeAsynch** method, as follows.

Signature `closeCallback(pipe: NamedPipe);`

getMaxMessageSize

Signature `getMaxMessageSize(): Integer;`

The **getMaxMessageSize** method of the **NamedPipe** class always returns zero (**0**), indicating that there is no upper limit to the allowable message size.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

listen

Signature `listen();`

The **listen** method of the **NamedPipe** class offers the server end of a named pipe connection and returns only when the offer has been accepted. The open offer remains in effect until a connection is established. The offer can be accepted only by a process that opens the client end of the offered pipe (for details, see the **open** method).

The **listen** method can be called only when the value of the **Connection** superclass **state** property is set to **Disconnected** (**0**).

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

The connection is established by opening a pipe name using the contents of the **name** property. Each end of the named pipe must open the pipe by using the same name.

The **Connection** superclass **fillReadBuffer** property determines whether the pipe is opened in bytes or in message mode. Both ends of the pipe must use the same mode.

The value of the **state** property changes to **Connected** (**2**) when the connection is open.

Multiple instances of the pipe can be opened by the same process or by multiple copies of the JADE application running from the same copy of the **jade.exe** executable program.

listenAsynch

Signature `listenAsynch(receiver: Object; msg: String);`

The **listenAsynch** method of the **NamedPipe** class offers a connection to a remote application and returns immediately.

The **listenAsynch** method can be called only when the value of the **Connection** superclass **state** property is **Disconnected** (**0**).

When the **listenAsynch** method is called, the value of the **state** property is changed to **Connecting** (**1**).

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

When the connection is established, the callback method name specified in the **msg** parameter is called for the receiver object. The callback method must match the signature required by the calling **listenAsynch** method, as follows.

Signature `listenCallback(pipe: NamedPipe);`

The connection is established by opening a pipe name using the contents of the **Connection** superclass **name** property. Each end of the named pipe must open the pipe by using the same name.

The **Connection** superclass **fillReadBuffer** property determines whether the pipe is opened in bytes or in message mode. Both ends of the pipe must use the same mode.

The value of the **state** property changes to **Connected** (**2**) when the connection is open.

Multiple instances of the pipe can be opened by the same process or by multiple copies of the JADE application running from the same **jade.exe** executable program.

If the **close** method is called before a connection is made, the **listenAsynch** callback routine is not called. If the **listenAsynch** method call fails, a connection exception is raised.

open

Signature `open();`

The **open** method of the **NamedPipe** class attempts to open the client end of an offered connection and returns immediately, if successful. If the connection attempt fails, an exception is raised. The Windows implementation of the Named Pipe connection does not allow the client side of the connection to wait for the offer to be made.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

The **open** method can be called only when the value of the **Connection** superclass **state** property is **Disconnected** (**0**).

The connection is established by opening a pipe name using the contents of the **Connection** superclass **name** property. Each end of the named pipe must open the pipe by using the same name.

The pipe can be opened across a network, by specifying the name of the server end of the pipe in the **serverName** property. If the processes opening both ends of the pipe are on the same workstation, the **serverName** property must be set to **null** or to the name of the current workstation.

The **Connection** superclass **fillReadBuffer** property determines whether the pipe is opened in bytes or in message mode. Both ends of the pipe must use the same mode.

The value of the **state** property changes to **Connected** (**2**) when the connection is open.

Multiple instances of the pipe can be opened by the same process, or by multiple copies of the JADE application running from the same **jade.exe** executable program.

openAsynch

Signature `openAsynch(receiver: Object;`
`msg: String);`

The **openAsynch** method of the **NamedPipe** class attempts to establish a connection to a named pipe and returns immediately. When the connection is established, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter.

If the connection attempt fails, an exception is raised. The connection attempt fails unless the server end of the connection is being offered.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

The **openAsynch** method can be called only when the value of the **Connection** class **state** property is **Disconnected** (**0**). When this method is called, the value of the **state** property is changed to **Connecting** (**1**).

When the **openAsynch** method establishes a connection, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **openAsynch** method, as follows.

Signature `openCallback(pipe: NamedPipe);`

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **NamedPipe** class reads binary data from the connection.

If the value of the **Connection** superclass **fillReadBuffer** property is **true**, the **readBinary** method returns when the number of bytes of data specified in the **length** parameter have been read.

If the value of the **fillReadBuffer** property is **false**, the method returns when the entire message has been received. The **readBinary** method uses the **length** parameter as the block size for reading and assembling the entire message. If the **length** parameter is set to zero (**0**), 4,000 bytes are used as the block size.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

This method can be called only when the value of the **Connection** superclass **state** property is **Connected** (**2**).

One synchronous or asynchronous read operation only can be performed at a time for any one connection.

readBinaryAsynch

Signature `readBinaryAsynch(length: Integer;`
`receiver: Object;`
`msg: String);`

The **readBinaryAsynch** method of the **NamedPipe** class initiates a read of binary data from the connection and returns immediately.

If the value of the **Connection** superclass **fillReadBuffer** property is **true**, when the bytes of data specified in the **length** parameter have been read, the callback method name is called for the object specified in the **receiver** parameter.

If the value of the **fillReadBuffer** property is **false**, when the entire message has been read, the callback method name specified in the **msg** parameter is called for the object specified in the **receiver** parameter. The **length** parameter is used as the block size for reading and assembling the entire message. If the **length** parameter is set to zero (**0**), 4,000 bytes are used as the block size.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

One synchronous or asynchronous read operation only can be performed at a time for any one connection.

The **readBinaryAsynch** method can be called only when the value of the **Connection** superclass **state** property is **Connected** (**2**).

The callback method must match the signature required by the calling **readBinaryAsynch** method, as follows.

Signature
```
readBinaryCallback(pipe:   NamedPipe;
                   buffer: Binary);
```

If the read fails, a connection exception is raised, specifying a Windows error number and description.

Note If the other end of the connection has been closed, this end of the connection is also closed. If this occurs, the value of the **Connection** superclass **state** property is **Disconnected** (**0**) and the Microsoft Windows exception that is raised is usually **109** (broken pipe).

writeBinary

Signature
```
writeBinary(buffer: Binary);
```

The **writeBinary** method of the **NamedPipe** class writes binary data to the connection and returns when the operation is complete.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

The **writeBinary** method can be called only when the value of the **Connection** superclass **state** property is **Connected** (**2**).

writeBinaryAsynch

Signature
```
writeBinaryAsynch(buffer:   Binary:
                  receiver: Object;
                  msg:      String);
```

The **writeBinaryAsynch** method of the **NamedPipe** class writes binary data to the connection and returns immediately.

When the operation is complete, the callback method name specified in the **msg** parameter is called for the receiver object parameter. The callback method must match the signature required by the calling **writeBinaryAsynch** method, as follows.

Signature
```
writeBinaryCallback(pipe: NamedPipe);
```

If the write operation fails, a connection exception is raised, with a Windows error number.

An exception is raised if this method is invoked from a server method when the server node is not running under a Windows operating system that supports services.

Note If the other end of the connection has been closed, this end of the connection is also closed. If this occurs, the value of the **Connection** superclass **state** property is **Disconnected** (**0**) and the exception raised is usually **109** (broken pipe).

The **writeBinaryAsynch** method can be called only when the value of the **Connection** superclass **state** property is **Connected** (**2**).

Another asynchronous write operation can be issued before the previous write operation is complete if it calls the same object and method on completion.

Node Class

The **Node** class is the class for which an instance exists for each node in a system. A *node* is a physical workstation participating in a particular application. A node can be a server node or a client node.

One node object exists for each *logical* workstation connected to the server node workstation. There is one fixed server node and one, none, or many client nodes.

A node represents a workstation that hosts the execution of one or several processes and it contains a dictionary of the processes currently active in the node. A node object is created for each JADE executable program that is running; that is, a workstation that is running two JADE applications has two node objects, or logical workstation connections, to the server.

For details about the constants, properties, and methods defined in the **Node** class, see "Node Class Constants", "Node Properties", and "Node Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

Node Class Constants

The constants provided by the **Node** class are listed in the following table.

Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)
ExternalProcess_Failed	External process failed, due to an exception (**modal** parameter is set to **true**)
ExternalProcess_InitiateFailed	External process failed to initiate
ExternalProcess_InitiateOK	External process initiated successfully (**modal** parameter is set to **false**)
ExternalProcess_InvalidParam	Invalid parameter in the external process
ExternalProcess_Successful	External process was successful (**modal** parameter is set to **true**)
OSUnknown	Operating system is unrecognized or cannot be determined
OSWindows	Operating system is Microsoft Windows
OSWindowsEnterprise	Operating system is Microsoft Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008

Constant	Description
OSWindowsHome	Operating system is Microsoft Windows 98 (not a supported operating system)
OSWindowsMobile	Operating system is Microsoft Windows CE (not a supported operating system)
Role_Replay	Replay node role
Role_Standard	Standard node role
Role_Unknown	Unknown node role
Type_Undefined	Undefined
Type_DatabaseServer	Database server (**jadrap** or **jadserv**)
Type_ApplicationServer	Application server (**jadapp** or **jadappb** in multiuser mode)
Type_ApplicationServerAndDatabaseServer	Application server and database server (**jadapp** or **jadappb** in single user mode)
Type_StandardClient	Standard client node (**jade** in multiuser mode; not as a thin client)
Type_StandardClientAndDatabaseServer	Standard client node and database server (**jade** in single user mode)
Type_NonGuiClient	Non-GUI (**jadclient**) node
Type_NonGuiClientAndDatabaseServer	Non-GUI (**jadclient**) node and database server
Type_DatabaseAdmin	Database administration (**jdbadmin**) node
Type_DatabaseAdminAndDatabaseServer	Database administration (**jdbadmin**) node and database server

Non-GUI nodes include user-written executables that use the JADE Object Manger API (C++) and the JADE .NET API (C#).

Node Properties

The properties defined in the **Node** class are summarized in the following table.

Property	Description
accessPatterns	Reserved for future use
name	Contains a read-only string of the node name
osID	Contains the process identifier of the operating system for the node
processes	Contains all processes currently executing in the node
system	Contains a read-only reference to the system object
userExitCode	Contains a value returned by your application when the JADE program exits

accessPatterns

Type: ProcessDict

The **accessPatterns** property of the **Node** class is not yet implemented. It is reserved for future use.

name

Type: String[255]

The **name** property of the **Node** class contains a read-only string of the node name. The computer name, obtained from the operating system, has a unique numeric identifier appended to it.

osID

Type: Integer

The **osID** property of the **Node** class contains the read-only process identifier of the operating system for the node.

You should use this property in preference to the less-efficient **Node** class **osProcessId** method.

processes

Type: ProcessDict

The read-only **processes** property of the **Node** class is a read-only property that contains a reference to all processes currently executing in the node.

The key of the dictionary is the **userCode** property of the process.

If abnormal terminations have occurred in a node, duplicated processes can exist.

Caution Lock environmental object collections with extreme caution, as this can cause hold-ups when processes sign off and on and when nodes initiate and terminate; for example, you should *never* use the **foreach** instruction to iterate through an environmental object collection. Instead, create a transient clone of the collection and iterate through that.

system

Type: System

The **system** property of the **Node** class contains a read-only reference to the system object, as shown in the code fragment in the following example.

```
foreach lock in locksArray do
    listBoxQueue.addItem(lock.requestedBy.node.system.name.String);
endforeach;
```

userExitCode

Type: Integer

The **userExitCode** property of the **Node** class contains a value returned by your applications when a JADE program (for example, **jade.exe**, **jadapp**, **jadrap.exe**, **jaded**, and so on) exits. The default value is zero (**0**). For more details, see Appendix A, "Exit Values", in the *JADE Installation and Configuration Guide*.

Tip You can use this property, for example, to set a non-zero exit code that can then be checked in a batch file by using the **ERRORLEVEL** keyword to check for appropriate **userExitCode** values, as shown in the following example.

```
begin
    beginTransaction;
    node.userExitCode := 123;
    commitTransaction;
    terminate;
end;
```

The specified value is returned only if the JADE program would have normally returned zero (**0**); that is, if JADE wants to return a non-zero exit value, the JADE value takes precedence over your value specified in this property.

If the **StandardExitValues** parameter in the [FaultHandling] section of the JADE initialization file is set to **false**, any exit code value that you specify in this property is returned, within any limitations imposed by Microsoft Windows.

Conversely, if the **StandardExitValues** parameter is set to **true**, your user-supplied value must be in the range zero (**0**) through **127**, inclusive. If it is outside this range, it is reset to **63**. As values in the range **32** through **63**, inclusive, are for your use as exit codes, JADE code will not remap JADE error numbers into this range.

Note As the **userExitCode** property applies to the JADE node, any JADE application can set this value. Cooperation between applications wanting to set this attribute may therefore be required.

Node Methods

The methods defined in the **Node** class are summarized in the following table.

Method	Description
beginIndividualRequestsLogging	Manually starts sampling individual remote requests of all processes in the local node
beginSample	Manually opens a new sample file and begins accumulating local node statistics
clearMethodCache	Clears previously loaded methods from method cache
createExternalProcess	Initiates an external process from within JADE logic
downloadCount	Returns the number of processes currently performing an automatic download of software
endIndividualRequestsLogging	Manually terminates sampling of individual remote requests of all process in the local node
endSample	Manually terminates sampling of statistics in the local node and releases the current file
getAppServerGroupName	Returns a string containing the name of the **AppServerGroupName** parameter in the JADE initialization file
getCacheSizes	Retrieves the cache sizes of the node on which the method is executing
getCacheSizes64	Retrieves the cache sizes of the node on which the method is executing when running in a 64-bit JADE environment

Method	Description
getCharacterSize	Returns an integer value representing the character size of the node of the receiver object
getCommandLine	Returns a string containing the command line of the node of the receiver object
getComputerName	Returns a string containing the computer name of the receiving node object
getDefaultLCID	Returns the number of the locale with which the background process for the node was initiated
getEnvironmentVariable	Returns a string containing the value of the specified user or system environment variable on the node of the receiver object
getExecuteFlagValue	Returns a boolean value containing the effective value of a flag used in **executeWhen** instructions
getIniFileName	Returns a string containing the name and full path of the JADE initialization file
getJadeHomeDirectory	Returns a string containing the JADE HOME directory
getJadeInstallDirectory	Returns a string containing the directory in which the JADE binaries are installed
getJadeWorkDirectory	Returns a string containing the directory in which JADE work files are created
getLCIDFromCharacterSet	Returns a locale ID corresponding to the specified short name of a character set
getLineDelimiter	Returns a string containing the line delimiter of the node of the receiver object
getLocks	Populates the specified array with transient instances of the current locks for shared transient instances
getMutexCounts	Retrieves the number of contentions on mutexes (locking mechanism used to ensure thread safety when executing critical sections of code) used internally by JADE for the node
getNotes	Reserved for future use
getObjectCaches	Retrieves node sampling values relating to cache activity
getOSDetails	Returns comprehensive information about the operating system and machine architecture of the node of the receiver object
getOSPlatform	Returns the operating system of the receiver object
getProfileString	Retrieves a string from the specified section in an initialization file on the application server workstation when the application is running in JADE thin client mode
getProgramDataDirectory	Returns a string containing the program data directory
getQueuedLocks	Populates the specified array with transient instances of the lock requests waiting for shared transient object to be unlocked

Method	Description
getRequestStats	Returns node statistics relating to persistent database requests from the node of the receiver object
getRpcServerStatistics	Retrieves statistics relating to RPC activity between the database server node and the node of the receiver object
getTempPath	Returns a string containing the architecture-specific version of the directory in which temporary files are created on the node of the receiver object
getUserDataDirectory	Returns a string containing the user data directory
isApplicationServer	Specifies whether the method is executing on an application server node
isCacheCoherencyEnabled	Specifies whether the receiving node has cache coherency enabled
isReadOnlySchema	Specifies whether the node on which the method is executing is a read-only schema
isReadOnlySystemSchema	Specifies whether the node on which the method is executing is a read-only system schema
isServerNode	Specifies whether the node on which the method is executing is the server
isService	Specifies whether the executable that is currently running on the node of the receiver object is running as a service
logObjectCaches	Specifies the local node object cache statistics logged to the sample statistics file
logRequestStatistics	Specifies the request statistics logged for processes in the local node
logUserCommand	Invokes the **NodeSampleUserCommandCallBack** entry point in the user library
networkAddress	Returns the IP address of the network interface connection to the database server
nodeRole	Returns an integer value that represents the database role of the node of the receiver object
nodeType	Returns an integer value that indicates the role of the node with regard to processes and the Synchronized Database Service (SDS)
osProcessId	Returns the process identifier of the executable that is currently running on the node of the receiver object
processDump	Invokes a non-fatal process dump of the node specified by the receiver
setCacheSizes	Sets the cache size values on the node on which the method is executing and retrieves the current values after the operation
setCacheSizes64	Sets the cache size values on the node on which the method is executing and retrieves the current values after the operation when running in a 64-bit JADE environment
setExecuteFlagValue	Sets the effective value of a flag used in **executeWhen** instructions
setProfileString	Copies a string into the specified section of the JADE initialization file
wbemListClasses	Retrieves a list of the WBEM classes that can be queried for the node of the receiver object

Method	Description
wbemListInstanceNames	Retrieves the names of all instances of a specified WBEM class for the node of the receiver object
wbemQueryQualifiers	Retrieves the name, type, and scale factor for each attribute of a specified WBEM class
wbemRetrieveData	Retrieves instances and attribute values for a specified WBEM class for the node of the receiver object

beginIndividualRequestsLogging

Signature

```
beginIndividualRequestsLogging(samplingHandle:              Integer;
                               localRequests:               Boolean;
                               remoteRequests:              Boolean;
                               persistentCacheBuffers:      Boolean;
                               transientCacheBuffers:       Boolean;
                               remoteTransientCacheBuffers: Boolean;
                               userNumber:                  Integer;
                               userText:                    String);
```

The **beginIndividualRequestsLogging** method of the **Node** class starts sampling the individual requests or cache activities, or both, of all processes in the local node and invokes the **NodeSampleIndividualRequestCallBack** or **NodeSampleObjectBufferCallBack** entry point, or both of these entry points, in the user library specified in the **libraryName** parameter of the **beginSample** method.

The **NodeSampleIntervalCallBack** entry point is invoked once only before these entry points, with the **eventType** parameter in the entry point set to **1**.

The **beginIndividualRequestsLogging** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSample** method when sampling for the node started
localRequests	Logs individual requests to the database of the node
remoteRequests	Logs individual requests to remote nodes
persistentCacheBuffers	Logs activities in the persistent object cache
transientCacheBuffers	Logs activities in the transient object cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the sampling of the statistics that you require, set the appropriate Boolean parameters to **true**. The following code fragment shows an example of the **beginIndividualRequestsLogging** method and its parameters.

```
node.beginIndividualRequestsLogging(samplingHandle, false, true, false,
     false, false, 4, "Start logging of remote requests");
```

The JADE sampling libraries produce the following record types.

- Begin process record (type **6**), which is optional
- **BeginInterval** record (type **11**), containing your specified user number and text to the output file immediately, followed by one **IndividualRequest** record for each of the subsequent individual requests or one cache buffer activity record for each of the subsequent buffer cache activities, or both
- Individual local request records (record type **14**)
- Individual remote request records (record type **10**)
- Cache buffer activity records (record type **2**)

For more details about the individual remote requests that are sampled in record types **2**, **6**, **7**, **10**, **11**, and **14**, see Chapter 4 of the *JADE Object Manager Guide*.

beginSample

Signature
```
beginSample(libraryName:              String;
            initializationParameter: String): Integer;
```

The **beginSample** method of the **Node** class opens a new sample context for the node, begins the accumulation of sampling statistics on that node, and invokes the following entry points.

- **NodeSampleInfoCallBack**, passing it the **initializationParameter** string and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleProcessInfoCallBack**, invoked every time that a process begins and once for every existing process at the time sampling begins.

This method returns the sampling handle number used to identify the sampling context that is opened. All subsequent methods use this sampling context handle as the first parameter.

When the **beginSample** method is called in your application, request statistics are stored in transient memory for every process in the node until they are passed to the corresponding entry point in the user library specified in the **libraryName** parameter. The JADE-supplied library writes a begin process record (type **6**) to the statistics file.

If you are using the **filesmpl** or **tcpsmpl** JADE sampling library, you can set the **initializationParameter** parameter to **"<null>"** or to **""** so that sample values will not be output. For **filesmpl**, the values will not be written to a file. For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted.

For more details, see "Direct Node Sampling", in Chapter 4 of the *JADE Object Manager Guide*.

The following is an example of a method that manually samples node statistics.

```
testManualSamplingFullInterval();
vars
    coy            : Company;
    cust           : Customer;
    custDict       : CustomerByNameDict;
    samplingHandle : Integer;
begin
    samplingHandle := node.beginSample("filesmpl",
                                       "c:\temp\fullInterval%p, txt");
```

```
    node.beginIndividualRequestsLogging(samplingHandle, false,
                        true, true, true, false, 557, "fullInterval");
    coy := Company.firstInstance;
    foreach cust in coy.CustDict do
        write cust.name;
    endforeach;
    node.endIndividualRequestsLogging(samplingHandle, 557, "fullInterval");
    node.endSample(samplingHandle);
end;
```

clearMethodCache

Signature `clearMethodCache();`

The **clearMethodCache** method of the **Node** class clears previously loaded methods from method cache. Use this method is if you have called the **setExecuteFlagValue** method to change the value of the **executeWhen** flag that conditionally loads method code, as shown in the following code fragment.

```
node.setExecuteFlagValue("DebugTest", true);
node.clearMethodCache();
```

createExternalProcess

Signature

```
createExternalProcess(directory:  String;
                      command:    String;
                      args:       StringArray;
                      alias:      String;
                      thinClient: Boolean;
                      modal:      Boolean;
                      result:     Integer output): Integer;
```

The **createExternalProcess** method of the **Node** class initiates an external process from within JADE logic. The parameters of the **createExternalProcess** method are listed in the following table.

Parameter	Usage
directory	Specifies the directory that you require for the working directory when you run the external process application specified in the command parameter. The current directory is used if this parameter contains an empty string.
command	Specifies the fully qualified path of the application (that is, the external process) that you want to run. **Note** To ensure that the expected executable is started, specify the full path to the executable. If the path includes spaces, the value should be in double quotes.
args	Specifies the external process parameters, or arguments. Each string in the array is passed as a separate argument to the command. This can be **null** if there are no arguments. If an argument contains white space, you will need to protect it by using quote marks; for example: `args[2] := 'ini="c:\Program files\jade\jade.ini"';`
alias	Not currently implemented (that is, this parameter is ignored).

Parameter	Usage
thinClient	If running in a JADE thin client environment, specifies whether the external process is executed on a presentation client workstation or the application server. If this parameter is set to **true**, the process is initiated on the presentation client workstation, or thin client.
modal	When **true**, specifies the suspension of the JADE application until the external process terminates, or when **false**, specifies that the JADE application is to run in parallel with the process.
result	Returns the exit value from the external process. This has meaning only when the modal parameter is set to **true**.

The values that are returned by this method are listed in the following table.

Node Class Constant	Integer Value	Description
ExternalProcess_Failed	3	External process failed, due to an exception (**modal** parameter is set to **true**)
ExternalProcess_InitiateFailed	2	External process failed to initiate
ExternalProcess_InitiateOK	0	External process initiated successfully (**modal** parameter is set to **false**)
ExternalProcess_InvalidParam	1	Invalid parameter in the external process
ExternalProcess_Successful	4	External process was successful (**modal** parameter is set to **true**)

The following example shows the use of the **createExternalProcess** method.

```
vars
    command, alias : String;
    args           : StringArray;
    exitValue      : Integer;     // random value if modal = false
    result         : Integer;
begin
    command := "mycommand";
    alias   := command;
    create args transient;
    args[1] := "first";
    args[2] := '"white space"';
    result  := node.createExternalProcess(".", command, args, alias,
                                          false, false, exitValue);
    if result = node.ExternalProcess_InvalidParam then
       write "Something is wrong with node.createExternalProcess arguments";
    elseif result = node.ExternalProcess_InitiateFailed then
       write "Could not start " & command;
    elseif result = node.ExternalProcess_InitiateOK then
       write "Non-modal command '" & command & "' started successfully";
    elseif result = node.ExternalProcess_Failed then
       write "Modal command '" & command & "' started, but died under
                     abnormal conditions";
    elseif result = node.ExternalProcess_Successful then
       write "Modal command '" & command & "' started, and exited with " &
                     exitValue.String;
    endif;
```

```
epilog
    delete args;
end;
```

downloadCount

Signature `downloadCount(): Integer;`

The **downloadCount** method of the **Node** class returns the number of processes that are currently performing an automatic download of software, to enable you to monitor the automatic download process.

For more details, see "Upgrading Software on Presentation Clients", in Appendix B of the *JADE Thin Client Guide*.

endIndividualRequestsLogging

Signature
```
endIndividualRequestsLogging(samplingHandle: Integer;
                             userNumber:     Integer;
                             userText:       String);
```

The **endIndividualRequestsLogging** method of the **Node** class terminates the sampling of individual remote requests or cache activities started by the **beginIndividualRequestsLogging** method of the **Node** class.

The **NodeSamplIntervalCallBack** entry point is invoked with the **eventType** parameter set to **2**.

The **endIndividualRequestsLogging** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSample** method when sampling for the node started
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The following code fragment shows an example of the **endIndividualRequestsLogging** method and its parameters.

```
node.endIndividualRequestsLogging(samplingHandle, 4, "End logging of remote
                                  requests");
```

The JADE-supplied library writes an **endInterval** record (type **12**), containing your specified user number and text, which is written to the output file specified in the **initializationParameter** parameter of the **beginSample** method.

For more details, see Chapter 4 of the *JADE Object Manager Guide*.

endSample

Signature `endSample(samplingHandle: Integer);`

The **endSample** method of the **Node** class terminates the sampling of statistics on the local node for the context identified by the **samplingHandle** parameter (returned by the **beginSample** method when sampling for the node started) and invokes the following entry points.

- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **2**.

- **NodeSampleInfoCallBack**, which your user library should consider the last call for the node sampling context.

The JADE-supplied library closes and releases the current sampling file, which you can then analyze.

You can produce multiple files during a node lifetime, by using the **Node** class **beginSample** and **endSample** methods, but you cannot sample statistics simultaneously on the same node. For more details, see Chapter 4 of the *JADE Object Manager Guide*.

getAppServerGroupName

Signature `getAppServerGroupName(): String;`

The **getAppServerGroupName** method of the **Node** class containing the name of the **AppServerGroupName** parameter in the [JadeAppServer] section of the JADE initialization file.

If the **AppServerGroupName** parameter is not specified, this method returns an empty string.

For details about application groups, see "Thin Client Connection Balancing", in Chapter 3 of the *JADE Thin Client Guide*.

getCacheSizes

Signature
```
getCacheSizes(persistentCache:      Integer output;
              transientCache:       Integer output;
              remoteTransientCache: Integer output);
```

The **getCacheSizes** method of the **Node** class retrieves the persistent, transient, and remote transient cache values of the node on which the method is executing.

These values, which are in bytes, represent the maximum amount of memory that is allocated by the JADE Object Manager library for caching objects in the node. See also the **setCacheSizes** method.

getCacheSizes64

Signature
```
getCacheSizes64(persistentCache:      Integer64 output;
                transientCache:       Integer64 output;
                remoteTransientCache: Integer64 output);
```

The **Node** class **getCacheSizes64** method retrieves the persistent, transient, and remote transient cache values of the node on which the method is executing.

These values, which are in bytes, represent the maximum amount of memory that is allocated by the JADE Object Manager library for caching objects in the node. See also the **setCacheSizes64** method.

getCharacterSize

Signature `getCharacterSize(): Integer;`

The **getCharacterSize** method of the **Node** class returns an integer value that represents the size of the character for the JADE version and operating system (for the Unicode version) under which the node of the receiver object is running.

The values that can be returned are listed in the following table.

Returned Value	Description
1	ANSI version of JADE
2	Unicode version of JADE in a Windows operating system

getCommandLine

Signature `getCommandLine(): String;`

The **getCommandLine** method of the **Node** class returns a string containing the current command line of the node of the receiver object.

In JADE thin client mode, this method returns the command line file from the application server. (Use **process.getCommandLine** to return the current command line of the presentation client.)

The command line of the specified node instance is returned, which does not have to be the current node. If you require the command line of the current node, use the **node** environmental object (system variable).

The following example shows the use of the **getCommandLine** method.

```
vars
    cmdLine, myOption : String;
    int               : Integer;
begin
    cmdLine := node.getCommandLine;                 // get command line
    // look for my command line option ('myOption')
    int     := cmdLine.pos('myOption', 1);
    if int <> 0 then
        int := cmdLine.pos('=', int) + 1;           // look for '='
        // skip any blanks after the '='
        cmdLine.scanWhile(' ', int);
        // return input up to next blank
        myOption := cmdLine.scanUntil(' ', int);
        write myOption;
    endif;
end;
```

getComputerName

Signature `getComputerName(): String;`

The **getComputerName** method of the **Node** class returns the computer name of the receiving node object.

getDefaultLCID

Signature `getDefaultLCID(): Integer;`

The **getDefaultLCID** method of the **Node** class returns an integer for the locale ID (LCID) with which the background process for the node was initiated.

In the following example, a presentation client determines the LCID of its application server node to obtain information about the locale.

```
vars
    lcid: Integer;
    info: LocaleFullInfo;
begin
    create info transient;
    lcid := node.getDefaultLCID;
    currentSchema.getLocaleFullInfo(lcid, info);
    . . .
epilog
    delete info;
end;
```

getEnvironmentVariable

Signature `getEnvironmentVariable(name: String): String;`

The **getEnvironmentVariable** method of the **Node** class returns a string containing the value of the user or system environment variable specified in the **name** parameter of the node of the receiver object.

The value specified in the **name** parameter equates to a variable listed in the **Variable** column on the Environment Variables dialog (accessed by selecting the **Advanced** sheet on the System Properties dialog). The returned value equates to the corresponding value listed in the **Value** column on the Environment Variables dialog for that row. For example, **envvar := node.getEnvironmentVariable("TEMP");** could return **C:\WINNT\TEMP**.

For details about returning the architecture-specific version of the directory in which temporary files are placed, see the **Node** class **getTempPath** method.

getExecuteFlagValue

Signature `getExecuteFlagValue(name: String): Boolean;`

The **getExecuteFlagValue** method of the **Node** class returns a boolean value containing the current, effective value of a flag used in **executeWhen** instructions.

The effective value of the flag is read from the [JadeExecuteFlags] section of the JADE initialization file when the node is initialized, but can be changed by calling the **setExecuteFlagValue** method.

getIniFileName

Signature `getIniFileName(): String;`

The **getIniFileName** method of the **Node** class returns the full path and file name of the JADE initialization file; for example:

```
c:\jade\system\jade.ini
```

The name of the JADE initialization file is returned in the form that it was entered on the command line. If no initialization file name was specified, JADE looks for an initialization file with the name **jade.ini** in the default location and either finds the file or creates it.

The name and full path of that *default* initialization file is returned with forward slash characters (for example, **c:/jade/system/jade.ini**).

The JADE initialization file is returned on the specified node instance, which does not have to be the current node. If you require the JADE initialization file on the current node, use the **node** environmental object (system variable).

In JADE thin client mode, this method returns the initialization file from the application server. Use the **Application** class **getIniFileName** method or the **Process** class **getIniFileName** to obtain the file from the thin client.

Note If you create a shortcut that has the **newcopy** parameter set to **false** and you specify a different JADE initialization file from the one with which the node was started, the active JADE initialization file is the one that was specified when the node started up and *not* the one specified in the **newcopy=false** shortcut.

Calling the **getIniFileName** method in a new application enables you to get the name of the initialization file that was used when the node started up.

getJadeInstallDirectory

Signature `getJadeInstallDirectory(): String;`

The **getJadeInstallDirectory** method of the **Node** class returns a string containing the JADE installation directory, from which the JADE executable program is running; for example:

```
c:\jade\bin
```

getJadeHomeDirectory

Signature `getJadeHomeDirectory(): String;`

The **getJadeHomeDirectory** method of the **Node** class returns a string containing the JADE HOME directory, which is the parent directory of the JADE installation directory; for example:

```
c:\jade      \\ if the installation directory was c:\jade\bin
```

getJadeWorkDirectory

Signature `getJadeWorkDirectory(): String;`

The **getJadeWorkDirectory** method of the **Node** class returns a string containing the directory where work files are created by JADE. When you call the **getJadeWorkDirectory** method and the directory does not exist, JADE creates it based on the value of the **JadeWorkDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file.

By default, this directory is created at the same level as the JADE installation directory (that is, the directory in which the **jade.exe** executable program is located) and is named **temp**. For example, if the JADE installation directory is **c:\jade\bin**, the working directory would be **c:\jade\temp**.

The cache file for a thin client (which contains all forms and pictures sent by logic from the application server) is stored in the work directory, unless another location is specified by the **FormCacheFile** parameter in the [JadeThinClient] section. The thin client automatic download interlock file (**thinlock.fil**) is also created in the work directory.

getLCIDFromCharacterSet

Signature `getLCIDFromCharacterSet(charset: String): Integer;`

The **getLCIDFromCharacterSet** method of the **Node** class returns an integer for the locale ID (LCID) that corresponds to the character set specified by the value of the **charset** parameter. If the value of the **charset** parameter is not a valid value or if the locale is not installed, the method returns zero (**0**).

Web pages and HTTP headers should contain the name of the character set of the encoding being used. The Euro symbol (€) is encoded as 0x80 and 0xA4, depending on the character set being used. As the JADE Unicode to ANSI conversion routines use the locale ID instead of the character set, the **getLCIDFromCharacterSet** method enables you to determine the locale ID to use in those routines and to determine whether the locale is installed.

The following example obtains the locale for the **fr-FR** character set.

```
lcid := node.getLCIDFromCharacterSet("fr-FR");  // locale ID is 1036
```

getLineDelimiter

Signature `getLineDelimiter(): String;`

The **getLineDelimiter** method of the **Node** class returns a string containing the line delimiter of the node of the receiver object (that is, **CrLf**).

getLocks

Signature
```
getLocks(locks:      LockArray input;
         maxEntries: Integer);
```

The **getLocks** method of the **Node** class populates the array specified in the **locks** parameter with transient instances of the current locks for the shared transient objects in the node specified as the method receiver.

The parameters for the **getLocks** method are listed in the following table.

Parameter	Specifies the …
locks	Locks array that is to be populated with the lock instances
maxEntries	Maximum number of lock instances that are to be included in the array

The following example shows the use of the **getLocks** method.

```
showSharedTransientLocks();
vars
    lock : Lock;
    lockArray : LockArray;
    nodedict : NodeDict;
    n : Node;
begin
    create lockArray transient;
    create nodedict transient;
    system.nodes.copy(nodedict);
    foreach n in nodedict do
        write "Shared transient locks for node " & n.String;
        n.getLocks(lockArray, 100);
        foreach lock in lockArray do
            write 'Oid ' & lock.target.String;
            write 'Locked by ' & lock.requestedBy.String;
        endforeach;
        lockArray.purge;
    endforeach;
epilog
    delete nodedict;
```

```
    delete lockArray;
end;
```

The output from the **getLocks** method shown in the previous example is as follows.

```
Shared transient locks for node Node/186.1
Shared transient locks for node Node/186.2
Oid Animal/51248.1
Locked by Process/187.04
```

getMutexCounts

Signature `getMutexCounts(jdo: JadeDynamicObject input; includeZeroContentions: Boolean);`

The **getMutexCounts** method of the **Node** class retrieves the number of contentions on mutexes used internally by JADE for the particular node identified as the method receiver. A *mutex* is a locking mechanism used to ensure thread safety when executing critical sections of code.

The contention counts are returned as **Integer64** properties of the **JadeDynamicObject** instance specified by the **jdo** parameter. The name of each property represents the internal mutex name, and the value represents the number of times that mutex has been contended (that is, the number of times execution of a thread has been temporarily suspended because another thread was executing in a section of code protected by the mutex).

The contention counts are cumulative from the time the specified node is initiated.

The **includeZeroContentions** parameter indicates whether mutexes that have not yet encountered any contentions should be included in the information returned. If set to **false**, only information for those mutexes that have had at least one contention are added to the dynamic object. If this parameter is set to **true**, information about all current mutexes is added.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. The **getMutexCounts** method clears any existing properties from the **JadeDynamicObject** instance each time it is called.

The number of mutexes reported and the order that the properties are added to the dynamic object can vary from call to call, because mutexes can be dynamically created and deleted.

Note If a mutex is deleted then recreated between **getMutexCounts** calls, the contention count can appear to reduce in value. Any application attempting to calculate contention count differences should take this into account.

The mutex contention information is primarily for internal use. High mutex contention counts can indicate bottlenecks that are impacting overall system performance.

The following example shows the use of the **getMutexCounts** method.

```
tryMutexCounts();
vars
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    node.getMutexCounts(jdo, false);
    write jdo.display;
epilog
    delete jdo;
end;
```

The output from the **getMutexCounts** method shown in the previous example is as follows.

```
---MutexStatistics(111)---
InterpreterOutp = 13
DrawTextLock = 17
PDB BuffChgLock = 1
PersistentCache = 102
PersistentDelet = 7
ClientTransient = 9
tblMgtLock = 2
```

getNotes

Signature

```
getNotes(notes:      NotificationArray input;
         transients: Boolean;
         maxEntries: Integer);
```

The **getNotes** method of the **Node** class is not yet implemented. It is reserved for future use.

getObjectCaches

Signature

```
getObjectCaches(dynObj:    JadeDynamicObject input;
                cacheType: Integer);
```

The **getObjectCaches** method of the **Node** class retrieves statistics relating to cache activity for the node specified as the method receiver.

The cache statistics values are returned as properties of a **JadeDynamicObject** object.

The cumulative counter values are not reset during the lifetime of the database server node, and you need to compare values from one execution of the **getObjectCaches** method with previous values to work out the differences.

The cumulative values are held as 64-bit unsigned integers, which are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

You can use the **getObjectCaches** method regardless of whether node sampling is enabled.

The **cacheType** parameter specifies whether information is retrieved from the persistent, transient, or remote transient cache. The retrieved values are listed in the following table.

Value	Description
1	Persistent cache
2	Transient cache
3	Remote transient cache (applicable only on server nodes)

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls. For a list and explanations about the properties that are returned by this method, see "Node::*getObjectCaches* Method", in Chapter 4 of the *JADE Object Manager Guide*.

If the dynamic object passed to the method already contains properties but they do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. The method is most efficient when the properties match those to be returned. The following example shows the use of the **getObjectCaches** method.

```
showCacheStatistics();
vars
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    node.getObjectCaches(jdo, 1 /*Persistent*/);
    write jdo.display;
epilog
    delete jdo;
end;
```

The output from the **getObjectCaches** method shown in the previous example is as follows.

```
---CacheStatistics(103)---
clockTicks = 103739652
nodeCPUTime = 7625000
nodeTicks = 452891
cacheType = 1
hits = 310723
misses = 29737
topOfLRUHits = 0
createdBuffers = 29745
cleanSwappedBuffers = 19297
dirtySwappedBuffers = 0
resizedBuffers = 1367
maximumBufferSize = 5000000
totalNumberOfBuffers = 10448
availableBufferSize = 159
maximumOverdraftBufferSize = 2500000
overdraftBufferSize = 0
deadBuffers = 0
totalOperations = 426611
currentOperations = 301982
currentBuffers = 10448
deletedBuffers = 19297
copiedBuffers = 0
newBuffers = 18
fetches = 29727
duplicateFetches = 4662
totalSwaps = 19297
totalOpsWhenSwapped = 124629
minOpsWhenSwapped = 1
maxOpsWhenSwapped = 289
totalAgeWhenSwapped = 2570606545
minAgeWhenSwapped = 104864
maxAgeWhenSwapped = 366995
lruTraversals = 2
totalLruTraversalTicks = 237599
latestLruTraversalTicks = 115197
totalCacheCoherencyNotifications = 0
cacheCoherencyNotificationHits = 0
```

```
cacheCoherencyUpdatedObjects = 0
cacheCoherencyObjectHits = 0
cacheCoherencyObjectMisses = 0
cacheCoherencyRangeRequests = 0
nodeLockRemoveRequestsSent = 0
nodeLockRemoveRequestsRcvd = 0
nodeLockSwapOutRequestsSent = 0
```

getOSDetails

Signature `getOSDetails(jdo: JadeDynamicObject input);`

The **getOSDetails** method of the **Node** class populates a **JadeDynamicObject** object with information about the operating system and architecture of the receiver node.

This method enables you to determine the various usages of JADE for a specific environment; for example, the type of binaries required for thin client downloads (for example, **x64-msoft-win64-ansi**).

The properties that are returned in the dynamic object specified in the **jdo** parameter are listed in the following table.

Property	Type	Description
version	String	Specific version of the operating system.
architecture	Integer	Internal byte ordering and alignment information relevant to JADE release. It is used by the **setByteOrderLocal** and **setByteOrderRemote** methods of the **Character**, **Date**, **Decimal**, **Integer**, **Integer64**, **Real**, **Time**, and **TimeStamp** primitive types. The architecture can be one of the values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

Property	Type	Description
platformId	Integer	Operating system of the server node of the receiver object. The operating system returned by this method can be one of the values listed in the following table.

Node Class Constant	Description
OSWindowsEnterprise	Microsoft Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008
OSWindowsHome	Microsoft Windows 98 (not a supported operating system)
OSWindowsMobile	Microsoft Windows CE (not a supported operating system)

buildArchitecture	String	Details about the platform and build type for which the binaries where built (for example, **x64-msoft-win64-ansi**). This can be used to determine the type of binaries required for thin client downloads.
currentBuildArchitectureList	String	Complete list of current **buildArchitecture** strings, separated by semicolons.
fullBuildArchitectureList	String	Complete list of past and current **buildArchitecture** strings, separated by semicolons.
isBigEndian	Boolean	Indicates if CPU for the node is running big-endian (PowerPC can switch from big-endian to little-endian, and the reverse).
characterSize	Integer	**1** for ANSI, **2** for Unicode.
addressWidth	Integer	**32** indicates 32-bit executing binaries, **64** indicates 64-bit executing binaries.
osAddressWidth	Integer	**32** indicates a 32-bit operating system, **64** indicates a 64-bit operating system.
osVersionEnum	Integer	Internal unique number representing the operating system and hardware combination.
osVersionString	String	Description of the operating system in a readable format.

The first three properties (**version**, **architecture**, and **platformId**) are the same as the values returned by the **getOSPlatform** method.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. The following example shows the use of the **getOSDetails** method.

```
vars
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    node.getOSDetails(jdo);
    write jdo.display;
epilog
    delete jdo;
end;
```

The output from the **getOSDetails** method shown in the previous example is as follows.

```
---GetOSDetails(300)---children = JadeDynamicObjectArray/4194888.1.4194886.2.1 : 0
name = "GetOSDetails"
parent = *** <null> object reference ***
type = 300
version = 10.0
architecture = 3
platformId = 17
buildArchitecture = x64-msoft-win64-ansi
fullBuildArchitectureList = i686-msoft-win32-ansi;ppc-ibm-aix433-ansi;i686-suse-
sles9-ansi;i686-redhat-rh9-ansi;i686-redhat-rhel3-ansi;i686-msoft-win32-
unicode;i686-redhat-rhel3-unicode;i686-suse-sles9-unicode;armv4-msoft-wince42-
unicode;i686-msoft-win32_6x-unicode;armv4-msoft-wince42_6x-unicode;armv4i-msoft-
wince50_6x-unicode;armv4i-msoft-wince50-unicode;i686-msoft-x86emu-unicode;i686-
suse-sles10-ansi;i686-suse-sles10-unicode;i686-redhat-rhel5-ansi;i686-redhat-rhel5-
unicode;x64-msoft-win64-ansi;x64-msoft-win64-unicode;x86_64-suse-sles10-ansi;x86_
64-suse-sles10-unicode;x86_64-redhat-rhel5-ansi;x86_64-redhat-rhel5-unicode;armv4i-
msoft-wm60-unicode;i686-msoft-win32_VS2005-ansi;i686-msoft-win32_VS2005-unicode
currentBuildArchitectureList = i686-msoft-win32-ansi;i686-msoft-win32-
unicode;armv4i-msoft-wince50-unicode;i686-msoft-x86emu-unicode;x64-msoft-win64-
ansi;x64-msoft-win64-unicode;armv4i-msoft-wm60-unicode
isBigEndian = false
characterSize = 1
addressWidth = 64
osAddressWidth = 64
osVersionEnum = 80
osVersionString = Windows 10 Enterprise, 64-bit Edition
```

getOSPlatform

Signature `getOSPlatform(version: String output; architecture: Integer output): Integer;`

The **getOSPlatform** method of the **Node** class returns an integer value that indicates the operating system of the receiver object. The operating system returned by this method can be one of the values listed in the following table.

Constant	Operating system is …
OSWindowsEnterprise	Microsoft Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008
OSWindowsHome	Microsoft Windows 98 (not a supported operating system)
OSWindowsMobile	Microsoft Windows CE (not a supported operating system)

The **version** parameter specifies the specific version of the operating system.

The following example uses the **OSWindows** class constant, which is a bit mask that enables you to identify a family of operating systems.

```
vars
    platform     : Integer;
    version      : String;
    architecture : Integer;
begin
```

```
    platform := node.getOSPlatform(version, architecture);
    if platform.bitAnd(Node.OSWindows) <> 0 then
        // operating system is Windows family (Windows 10, Windows 8,
        // Windows 7, Windows Server 2016, Windows Server 2012,
        // or Windows Server 2008)
        if platform = Node.OSWindowsHome then
            // version is an older version of Windows (unsupported)
            return 'Windows (unsupported) ' & version;
        endif;
        if platform = Node.OSWindowsEnterprise then
            // version is Windows 10, Windows 8, Windows 7, Windows
            // Server 2016, Windows Server 2012, or Windows Server 2008
            return 'Windows ' & version;
        endif;
        if platform = Node.OSWindowsMobile then
            // version is Windows CE
            return 'Windows CE (unsupported) ' & version;
        endif;
    endif;
    return '* Unknown platform: '& platform.String & ' version: ' & version;
end;
```

The **architecture** parameter indicates internal byte ordering and alignment information relevant to this release of JADE. It is used by the **setByteOrderLocal** and **setByteOrderRemote** methods of the **Character**, **Date**, **Decimal**, **Integer**, **Integer64**, **Real**, **Time**, and **TimeStamp** primitive types.

The architecture can be one of the values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

In JADE thin client mode, this method returns the operating platform of the workstation that is running the JADE logic; that is, the application server. (To return the operating system of the presentation client, use the **getOSPlatform** method of the **Process** class.)

getProfileString

Signature

```
getProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 default:  String): String;
```

The **getProfileString** method of the **Node** class retrieves a parameter (key name) string from the specified section of the JADE initialization file on the application server workstation when the application is running in JADE thin client mode.

The key name string is returned on the specified node instance, which does not have to be the current node. If you require the key name on the current node, use the **node** environmental object (system variable).

If the application is not running in JADE thin client mode, this method functions like the **Application** class **getProfileStringAppServer** method or **process**.**getProfileString**; that is, it returns the specified profile string from the workstation in which the application or process is running.

The **setProfileString** method copies the string into the specified section of an initialization file on the node.

Use the **Application** class **getProfileString** method or **Process** class **getProfileString** method to obtain the file from the application server.

The parameters for the **getProfileString** method are listed in the following table.

Parameter	Specifies the…
fileName	Initialization file. If you set this parameter to **windows**, the **win.ini** file on the application server workstation is used. If it does not contain a full path to the file, Windows searches for the file in the Windows directory on the application server.
section	Initialization file section containing the key (parameter) name.
keyName	Name of the key (parameter) whose associated string is to be retrieved.
default	Default value for the specified key if the key cannot be found in the initialization file.

You can return all initialization file sections or all parameters in a section, by using the **JadeProfileString** category global constants listed in the following table.

Global Constant	Specified in the…	Returns all…
ProfileAllKeys	**keyName** parameter	Key (parameter) strings in the initialization file section, separated by spaces
ProfileAllSections	**section** parameter	Initialization file sections, separated by spaces

You can use this method to retrieve a string from a two-level section name (prefixed with a unique identifier) within a JADE initialization file shared by multiple programs on the same application server host. For details, see "Two-Level Section Names" under "Format of the JADE Initialization File", in the *JADE Initialization File Reference*.

getProgramDataDirectory

Signature `getProgramDataDirectory(): String;`

The **getProgramDataDirectory** method of the **Node** class returns a string containing the path of the program data directory.

The program data directory is used to share files among the users of the executables; for example, the **jommsg.log** file or shared dictionary spelling files that are updated.

If JADE is not installed under the **\Program Files** directory, the path of the JADE HOME directory is returned. If JADE is installed under the **\Program Files** directory, the value that is returned by the **getProgramDataDirectory** method depends on the value of the **ProgramDataDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file.

If the directory does not exist, JADE creates it. The values of the **ProgramDataDirectory** parameter and the corresponding values returned by the **getProgramDataDirectory** method are shown in the following table.

ProgramDataDirectory Value	Return Value
<default>	For a Windows release earlier than Windows 7, Windows Server 2016, or Windows Server 2008, the value returned is the same as for **<homedir>**. For Windows 10, Windows 8, Windows 7, Windows Server 2012, or Windows Server 2008, the value returned is the same as for **<programdata>**.
<homedir>	The path of the **JADE HOME** directory.
<programdata>	The path of the **JADE HOME** directory with the **\Program Files** portion replaced with the programmatically obtained path of the common application data directory. For example, a presentation client installed in **\Program Files\Jade Software\parsys** returns **\Documents and Settings\All Users\Application Data\Jade Software\parsys** on a Windows operating system earlier than Windows 7, or Windows Server 2008, or **\ProgramData\Jade Software\parsys** on Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008.
Directory name	Directory name.

getQueuedLocks

Signature
```
getQueuedLocks(locks:      LockArray input;
               maxEntries: Integer);
```

The **getQueuedLocks** method of the **Node** class populates the array specified in the **locks** parameter with transient instances of the lock requests that are waiting for shared transient objects in the node specified as the method receiver to be unlocked by the processes that currently have them locked.

The parameters for the **getQueuedLocks** method are listed in the following table.

Parameter	Specifies the …
locks	Locks array that is to be populated with the lock request instances
maxEntries	Maximum number of lock instances that are to be included in the array

The calling process is responsible for creating and deleting the **LockArray** instance used with this method, as well as deleting the **Lock** instances inserted into the array.

The following example shows the use of the **getQueuedLocks** method.

```
showQueuedSharedTransientLocks();
vars
    lock : Lock;
    lockArray : LockArray;
    nodedict : NodeDict;
    n : Node;
begin
    create lockArray transient;
    create nodedict transient;
    system.nodes.copy(nodedict);
    foreach n in nodedict do
```

```
            write 'Queued shared transient locks for node ' & n.String;
            n.getQueuedLocks(lockArray, 100);
            foreach lock in lockArray do
                write 'Oid ' & lock.target.String;
                write 'Locked by ' & lock.lockedBy.String;
                write 'Requested by ' & lock.requestedBy.String;
            endforeach;
            lockArray.purge;
        endforeach;
    epilog
        delete nodedict;
        delete lockArray;
    end;
```

The output from the **getQueuedLocks** method shown in the previous example is as follows.

```
Queued shared transient locks for node Node/186.1
Queued shared transient locks for node Node/186.2
Oid Animal/51248.1
Locked by Process/187.5
Requested by Process/187.6
```

getRequestStats

Signature `getRequestStats(jdo: JadeDynamicObject input);`

The **getRequestStats** method of the **Node** class returns node statistics relating to persistent database requests from the receiving node that is the method receiver.

The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

The node statistics are held on the database server node and relate to persistent object requests received from the node specified as the method receiver. For details about the properties returned in the dynamic object, see "Node::*getRequestStats* Method" in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. The method is most efficient when the properties match those to be returned.

The cumulative counter values are not reset during the lifetime of the database server node, and you may need to compare values from one execution of the **getRequestStats** method with previous values to work out the differences.

The cumulative values are held as 64-bit unsigned integers, which are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getRequestStats** method.

```
showNodeRequestStats();
vars
   jdo : JadeDynamicObject;
begin
```

```
   create jdo transient;
   node.getRequestStats(jdo);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getRequestStats** method shown in the previous example is as follows.

```
---NodeStatistics(104)---
committedTransactions = 43
abortedTransactions = 0
getObjects = 10173
createObjects = 363
deleteObjects = 136
updateObjects = 526
lockObjects = 15588
unlockObjects = 6797
beginNotifications = 537
endNotifications = 52
serverMethodExecutions = 0
causeEvents = 60
```

getRpcServerStatistics

Signature `getRpcServerStatistics(jdo: JadeDynamicObject input; detailed: Boolean);`

The **getRpcServerStatistics** method of the **Node** class RPC statistics relating to activity between the database server node and the client node represented by the **Node** instance used as the method receiver.

The values returned represent information about the connection to the specified node and totals for requests received and replies sent to it. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter. The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

The **detailed** parameter specifies whether the values returned should include individual totals for each request type. For details about the returned values, see "Node::*getRpcServerStatistics* Method", in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. The method is most efficient when the properties match those to be returned.

The cumulative counter values are not reset during the lifetime of the database server node, and you may need to compare values from one execution of the **getRpcServerStatistics** method with previous values to work out the differences.

The cumulative values are held as 64-bit unsigned integers, which are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getRpcServerStatistics** method.

```
showRpcNodeStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   node.getRpcServerStatistics(jdo, false);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getRpcServerStatistics** method shown in the previous example is as follows.

```
---RPCServerStatistics(106)---
timeStarted = 27 April 2007, 12:31:43
connectionType = 1
lastInboundRequest = 27 April 2007, 14:38:30
requestsFromClients = 24155
repliesToClients = 24154
requestPacketsFromClients = 24155
replyPacketsToClients = 24154
requestBytesFromClients = 3719096
replyBytesToClients = 9861812
requestsToClients = 38
repliesFromClients = 38
requestPacketsToClients = 38
replyPacketsFromClients = 38
requestBytesToClients = 40611
replyBytesFromClients = 20962
notificationPacketsToClients = 3
notificationBytesToClients = 1122
```

getTempPath

Signature `getTempPath(): String;`

The **getTempPath** method of the **Node** class returns a string containing the architecture-specific version of the directory in which temporary files are created on the node of the receiver object. For example, this method returns **TEMP** or **TMP**, as appropriate.

The temporary path of the specified node instance is returned, which does not have to be the current node. If you require the temporary path of the current node, use the **node** environmental object (system variable).

For details about returning the value of a specified environment variable, see the **Node** class **getEnvironmentVariable** method.

getUserDataDirectory

Signature `getUserDataDirectory(): String;`

The **getUserDataDirectory** method of the **Node** class returns a string containing the path of the user data directory. The user data directory is used for files that are specific to each user of the JADE executables; for example, if a presentation client installation occurs on a Windows machine running Citrix or Terminal Services and all users run the same thin client binaries, any data created on the client file system should be stored under this directory (that is, in separate directories for each user).

If JADE is not installed under the **\Program Files** directory, the location of the JADE HOME directory is returned.

If JADE is installed under the **\Program Files** directory, the value that is returned depends on the value of the **UserDataDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file. If the directory does not exist, JADE creates it.

The values of the **UserDataDirectory** parameter and the corresponding values returned by the **getUserDataDirectory** method are shown in the following table.

UserDataDirectory Value	Return Value
<default>	The path of the **JADE HOME** directory with the **\Program Files** portion replaced with the programmatically obtained path for the specific user application private data directory. For example, a presentation client installed into **\Program Files\Jade Software\parsys** and executed by user **wilbur** returns **\Users\wilbur\AppData\Local\Jade Software\parsys**.
<homedir>	The path of the **JADE HOME** directory.
<userdata>	The same as for **<default>**.
Directory name	Directory name.

isApplicationServer

Signature `isApplicationServer(): Boolean;`

The **isApplicationServer** method of the **Node** class returns **true** if the method is executing on an application server node (that is, the application is running in JADE thin client mode).

isCacheCoherencyEnabled

Signature `isCacheCoherencyEnabled(): Boolean;`

The **isCacheCoherencyEnabled** method of the **Node** class returns **true** if the receiving node has cache coherency enabled.

For details, see the **AutomaticCacheCoherency** parameter in the [JadeClient] section and the **AutomaticCacheCoherency** or **AutomaticCacheCoherencyDefault** parameter in the [JadeServer] section of the JADE initialization file.

isReadOnlySchema

Signature `isReadOnlySchema(): Boolean;`

The **isReadOnlySchema** method of the **Node** class returns **true** if the JADE schema in the node on which the method is executing is a read-only schema, specified by using the **ReadOnlySchema** parameter in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file.

For details about the **ReadOnlySchema** parameter, see "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*.

isReadOnlySystemSchema

Signature `isReadOnlySystemSchema(): Boolean;`

The **isReadOnlySystemSchema** method of the **Node** class returns **true** if the JADE schema in the node on which the method is executing is a read-only system schema, specified by using the **ReadOnlySystemSchema** parameter in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file.

For details about the **ReadOnlySystemSchema** parameter, see "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*.

isServerNode

Signature `isServerNode(): Boolean;`

The **isServerNode** method of the **Node** class returns **true** if the node on which the method is executing is the server node. This method returns **false** if the node on which the method is executing is running as a client in a multiuser JADE configuration.

isService

Signature `isService(): Boolean;`

The **isService** method of the **Node** class returns **true** if the executable that is currently running on the node of the receiver object is running as a service or it returns **false** if the executable is *not* running as a service.

logObjectCaches

Signature
```
logObjectCaches(samplingHandle:              Integer;
                persistentCacheStats:        Boolean;
                persistentCacheBuffers:      Boolean;
                transientCacheStats:         Boolean;
                transientCacheBuffers:       Boolean;
                remoteTransientCacheStats:   Boolean;
                remoteTransientCacheBuffers: Boolean;
                userNumber:                  Integer;
                userText:                    String);
```

The **logObjectCaches** method of the **Node** class specifies the local node object cache statistics that are logged by invoking the **NodeSampleCacheInfoCallBack** or **NodeSampleObjectBuffer** entry point, or both of these entry points, in the user library.

The JADE-supplied library logs the statistics to the file specified in the **initializationParameter** parameter of the **beginSample** method and writes the following statistics to your output file.

- Cache header record (type **1**) for cache statistics
- Cache buffer records (type **2**) for individual object buffers

The **logObjectCaches** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSample** method when sampling for the node started
persistentCacheStats	Logs statistics of the persistent objects cache
persistentCacheBuffers	Logs statistics of the persistent object cache buffers
transientCacheStats	Logs statistics of the transient objects cache
transientCacheBuffers	Logs statistics of the transient object cache buffers
remoteTransientCacheStats	Logs statistics of the remote transient objects cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache buffers
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the cache statistics that you require, set the appropriate Boolean cache parameters to **true**.

The following code fragment shows an example of the **logObjectCaches** method and its parameters.

```
node.logObjectCaches(samplingHandle, true, true, false, false, false, false,
                50, "After the load data operation");
```

All buffers containing non-shared transient objects are listed when node sampling snapshots are requested.

For more details, see "Statistics File Format", in Chapter 4 of the *JADE Object Manager Guide*.

logRequestStatistics

Signature

```
logRequestStatistics(samplingHandle: Integer;
                     local:          Boolean;
                     remote:         Boolean;
                     userNumber:     Integer;
                     userText:       String);
```

The **logRequestStatistics** method of the **Node** class specifies the request statistics that are logged for all processes in the node that are logged by invoking the **NodeSampleRequestStatisticsCallBack** entry point in the user library.

The JADE-supplied library automatically writes the following statistics.

- Local request statistics record (type **8**)
- Remote request statistics record (type **9**)

The **logRequestStatistics** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSample** method when sampling for the node started
local	Logs statistics of all requests invoked on the local node
remote	Logs statistics of all requests from the local node to remote nodes
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the request statistics that you require, set the appropriate Boolean cache parameters to **true**. The user number and text values specified in the **userNumber** and **userText** parameters are written in the corresponding records.

The following code fragment shows an example of the **logRequestStatistics** method and its parameters.

```
node.logRequestStatistics(samplingHandle, true, true, 23, "Before
                          method m1");
```

For more details, see "Statistics File Format", in Chapter 4 of the *JADE Object Manager Guide*.

logUserCommand

Signature

```
logUserCommand(samplingHandle: Integer;
               command:        String;
               userNumber:     Integer;
               userText:       String);
```

The **logUserCommand** method of the **Node** class invokes the **NodeSampleUserCommandCallBack** entry point in the user library, passing the **command** parameter to it.

The JADE-supplied library automatically writes the user command (type **13**).

The **logUserCommand** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSample** method when sampling for the node started
command	Action specific to your user library (for example, the JADE-supplied library uses this command for filtering and setting the **SamplingExceptionEvent**)
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

For more details, see "JADE Sampling Libraries", "Statistics File Format", and "Sampling Exception Handling", in Chapter 4 of the *JADE Object Manager Guide*.

networkAddress

Signature `networkAddress(): String;`

The **networkAddress** method of the **Node** class returns a string whose contents depend on the type of transport used for the connection to the database server.

When the transport is TCP/IP, the string contains the IP address used by the client for the connection to the database server; for example, **127.0.0.1** or **::1**.

When the transport is **JadeLocal**, the returned string is empty.

When the transport is **HPSM**, the returned string contains **"proc*NNNN*"**, where the ***NNNN*** value is the decimal number of the process at the other end of the connection.

nodeRole

Signature `nodeRole(): Integer;`

The **nodeRole** method of the **Node** class returns an integer value that represents the role of the node with regard to processes and the Synchronized Database Service (SDS). The role can be one of the values listed in the following table.

Node Class Constant	Integer Value	Description
Role_Replay	2	Replay node role
Role_Standard	1	Standard node role
Role_Unknown	0	Unknown node role

An SDS secondary server has two node objects, as follows.

- A standard node object to which processes initiated on the secondary server are attached
- A replay node object to which pseudo-processes representing processes on the primary server are attached

The **nodeRole** method allows processes initiated on SDS secondary servers to be distinguished from pseudo-processes automatically created to represent processes on the primary server, by using the **process.node.nodeRole** method.

nodeType

Signature `nodeType(): Integer;`

The **nodeType** method of the **Node** class returns a **Node** class constant integer value that represents the type of the node object. The values that can be returned are listed in the following table.

Node Class Constant	Description
Type_Undefined	Undefined
Type_DatabaseServer	Database server (**jadrap** or **jadserv**)
Type_ApplicationServer	Application server (**jadapp** or **jadappb** in multiuser mode)
Type_ApplicationServerAndDatabaseServer	Application server and database server (**jadapp** or **jadappb** in single user mode)

Node Class Constant	Description
Type_StandardClient	Standard client node (**jade** in multiuser mode; not as a thin client)
Type_StandardClientAndDatabaseServer	Standard client node and database server (**jade** in single user mode)
Type_NonGuiClient	Non-GUI (**jadclient**) node
Type_NonGuiClientAndDatabaseServer	Non-GUI (**jadclient**) node and database server
Type_DatabaseAdmin	Database administration (**jdbadmin**) node
Type_DatabaseAdminAndDatabaseServer	Database administration (**jdbadmin**) node and database server

Non-GUI nodes include user-written executables that use the JADE Object Manger API (C++) and the JADE .NET API (C#).

osProcessId

Signature `osProcessId(): Integer;`

The **osProcessId** method of the **Node** class returns the process identifier (or process id) of the executable that is currently running on the node of the receiver object. (See also the **osID** property.)

The process id is the number before the dash character (**-**) in the third column in a **jommsg.log** file. For example, in the **2002/01/18 07:10:28 00618-6a4 PDB: Database closed successfully** record in a JADE message log file, **00618** is the hexadecimal process id.

This method returns the process identifier from the node on which the fat client executes when running in standard (fat) client mode or it returns the value from the application server when the method is executed from a presentation client running in JADE thin client mode.

processDump

Signature `processDump();`

The **processDump** method of the **Node** class invokes a non-fatal process dump of the node specified by the receiver.

setCacheSizes

Signature
```
setCacheSizes(persistentCache:      Integer io;
              transientCache:       Integer io;
              remoteTransientCache: Integer io);
```

The **setCacheSizes** method of the **Node** class changes the sizes of the persistent, transient, and remote transient cache on the node on which the method is executing to be set to the specified values.

The cache size cannot be set lower than the minimum for that type of cache or higher than two-thirds of the physical memory size. In addition, the cache size sometimes cannot be reduced because of current usage of objects on it.

If the cache size cannot be set to the requested value, it is increased or reduced as much as possible at that time. No exception is raised.

The parameter values are then updated with the actual new cache sizes.

The cache size on 32-bit systems cannot exceed 4G bytes.

Note The values are set for the current JADE session only.

When you next initiate JADE, the values in the **ObjectCacheSizeLimit**, **TransientCacheSizeLimit**, and **RemoteTransientCacheSizeLimit** parameters in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file are those that are used for the persistent, transient, and remote transient cache sizes, respectively.

For details about cache sizes, see the appropriate parameters in "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*. See also the **getCacheSizes** method.

setCacheSizes64

Signature
```
setCacheSizes64(persistentCache:      Integer64 io;
                transientCache:       Integer64 io;
                remoteTransientCache: Integer64 io);
```

The **setCacheSizes64** method of the **Node** class changes the sizes of the persistent, transient, and remote transient cache on the node on which the method is executing to be set to the specified values.

The cache size cannot be set lower than the minimum for that type of cache or higher than two-thirds of the physical memory size. In addition, the cache size sometimes cannot be reduced because of current usage of objects on it. If the cache size cannot be set to the requested value, it is increased or reduced as much as possible at that time. No exception is raised.

The parameter values are then updated with the actual new cache sizes.

The cache size on 32-bit systems cannot exceed 4G bytes.

Note The values are set for the current JADE session only.

When you next initiate JADE, the values in the **ObjectCacheSizeLimit**, **TransientCacheSizeLimit**, and **RemoteTransientCacheSizeLimit** parameters in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file are those that are used for the persistent, transient, and remote transient cache sizes, respectively.

For details about cache sizes, see the appropriate parameters in "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*. See also the **getCacheSizes64** method.

setExecuteFlagValue

Signature
```
setExecuteFlagValue(name:  String
                    value: Boolean): Boolean;
```

The **setExecuteFlagValue** method of the **Node** class sets the effective value of a flag used in **executeWhen** instructions.

The flag is a **Boolean** global constant. However, the defined value of the global constant is not used for an **executeWhen** instruction. Instead, the effective value of the global constant is read from the [JadeExecuteFlags] section of the JADE initialization file when the node is initialized.

The following code fragment shows the use of the **setExecuteFlagValue** method with the **clearMethodCache** method, which is required to cause methods to be reloaded with changed flag values.

```
node.setExecuteFlagValue("DebugTest", true);
node.clearMethodCache();
```

setProfileString

Signature

```
setProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 string:   String): Boolean;
```

The **setProfileString** method of the **Node** class copies a parameter (key name) string specified in the **section** parameter into the section of an initialization file on the application server.

The key name string is set on the specified node, which does not have to be the current node. If you want to set the key name on the current node, use the **node** environmental object (system variable).

This method returns **true** if it succeeds in storing the specified string. Conversely, if the value of the **section** or **keyName** parameter is null (**""**) or empty, this method returns **false**, to indicate that the JADE initialization file has not been updated.

Use the respective **ProfileRemoveSection** or **ProfileRemoveKey** global constant in the **JadeProfileString** category to delete a section or key, rather than passing a null or empty string in the appropriate parameter of this method.

To retrieve a stored string, use the **getProfileString** method.

The parameters for the **setProfileString** method are listed in the following table.

Parameter	Description
fileName	Specifies the initialization file. If you set this parameter to **windows**, the **win.ini** file is used. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory.
section	Specifies the initialization file section containing the key (parameter) name.
keyName	Specifies the name of the key (parameter) whose associated string is to be stored.
string	Specifies the string that is to be written to the file.

In JADE thin client mode, this method sets the initialization file string in the specified initialization file on the application server.

If the application is not running in JADE thin client mode, this method functions like the **Application** class **setProfileStringAppServer** method or **process**.**getProfileString**; that is, it sets the specified profile string on the workstation in which the application or process is running.

The following example shows the use of this method to remove an entire [mySection] section and the **WindowPos** parameter in the [InternalAS.JadeAppServer] section from the JADE initialization file.

```
begin
    node.setProfileString(node.getIniFileName, "mySection",
                          ProfileRemoveSection, "");
    // If the user has moved the window, reset it to the default values
    node.setProfileString(node.getIniFileName, "JadeAppServer", "WindowPos",
```

```
                                ProfileRemoveKey);
    end;
```

wbemListClasses

Signature `wbemListClasses(hsa: HugeStringArray input);`

The **wbemListClasses** method of the **Node** class retrieves a list of the Web-Based Enterprise Management (WBEM) classes that can be queried for the host machine on which the node specified by the receiver object is running. This is a subset of the full WBEM classes available, as JADE allows only a subset of classes to be queried. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information.

The method inserts strings containing the allowed class names into the huge string array specified by the **hsa** parameter.

The **wbemListClasses** method always empties the array before inserting the class names. The caller is responsible for creating and deleting this array.

The strings that are inserted into the huge string array parameter are fully qualified WBEM class names that can be used directly as class names for the other WBEM methods provided by the **Node** class.

The following example shows the use of the **wbemListClasses** method.

```
showWbemClasses()
vars
    hsa: HugeStringArray;
    str : String;
begin
    create hsa transient;
    node.wbemListClasses(hsa);
    foreach str in hsa do
        write "WBEM class name : " & str;
    endforeach;
epilog
    delete hsa;
end;
```

An example of the output from this method is as follows:

```
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfDisk_LogicalDisk
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfOS_Cache
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfOS_System
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_Tcpip_NetworkInterface
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfNet_Server
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfOS_Processor
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_PerfOS_Processor
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_PerfDisk_LogicalDisk
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
```

```
PerfFormattedData.Win32_PerfFormattedData_PerfOS_Cache
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_PerfOS_System
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_Tcpip_NetworkInterface
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_PerfOS_Memory
WBEM class name : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfFormattedData.Win32_PerfFormattedData_PerfNet_Server
```

wbemListInstanceNames

Signature `wbemListInstanceNames(className: String; instNameArray: HugeStringArray input);`

The **wbemListInstanceNames** method of the **Node** class retrieves the names of all instances of the Web-Based Enterprise Management (WBEM) class specified by the **className** parameter for the host machine in which the node of the receiver object is running.

The class name must be a fully qualified WBEM class name.

The instance names are inserted as strings into the huge string array specified by the **instNameArray** parameter. This method always empties the array before inserting the instance names. The caller is responsible for creating and deleting this array.

JADE allows only a subset of the available WBEM classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information. You can use the **Node** class **wbemListClasses** method to retrieve the fully qualified WBEM class names that can be used.

An exception is raised if a name that is not allowed or recognized is used.

The following example shows the use of the **wbemListInstanceNames** method.

```
showWbemInstances();
vars
    hsa : HugeStringArray;
    cls : String;
    inst : String;
begin
    create hsa transient;
    node.wbemListClasses(hsa);
    if hsa.size > 0 then
        cls := hsa[1];
        hsa.purge;
        write "WBEM class : " & cls;
        node.wbemListInstanceNames(cls, hsa);
        foreach inst in hsa do
            write "Instance : " & inst;
        endforeach;
    endif;
epilog
    delete hsa;
end;
```

The output from the **wbemListInstanceNames** method shown in the previous example is as follows.

```
WBEM class : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfDisk_LogicalDisk
Instance : C:
Instance : E:
Instance : _Total
```

wbemQueryQualifiers

Signature

```
wbemQueryQualifiers(className:      String;
                    attributeNames: StringArray input;
                    counterTypes:   IntegerArray input;
                    scaleFactors:   IntegerArray input);
```

The **wbemQueryQualifiers** method of the **Node** class retrieves the names, type, and scale factor for each attribute of Web-Based Enterprise Management (WBEM) class specified by the **className** parameter. This allows attribute values returned by the **wbemRetrieveData** method defined in the **Node** class to be correctly interpreted.

The qualifier information is placed into three matched arrays. Information for the first attribute is placed into the first member of each of the three arrays, information for the second attribute is placed into the second member, and so on.

The caller is responsible for creating and deleting the three arrays. This method always empties these arrays before inserting the qualifier information.

The class name must be a fully qualified WBEM class name.

JADE allows only a subset of the available WBEM classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information. The **wbemListClasses** method of the **Node** class can be used to retrieve the fully qualified WBEM class names that can be used.

An exception is raised if a name that is not allowed or recognized is used.

The string array specified by the **attributeNames** parameter contains the name of each attribute for the specified class. These match the names of the attributes that the **wbemRetrieveData** method creates in the **JadeDynamicObject** it uses.

The integer array specified by the **counterTypes** parameter contains performance counter type values for the attributes. The values are those defined by Microsoft and documented in the MSDN (Microsoft Developer Network) literature.

The integer array specified by the **scaleFactors** parameter contains the default scale factor to be applied to the attribute values. This is a power of 10 that can be used to estimate the likely range of the value.

The meaning of each counter type value and the correct way to extract meaningful information from the attribute values is described in the MSDN literature. Searching using **WMI Performance Counter Types** should locate the relevant information.

The following example shows the use of the **wbemQueryQualifiers** method.

```
showWbemQualifiers();
vars
    hsa : HugeStringArray;
    ctrNames : StringArray;
    ctrTypes : IntegerArray;
    ctrScaleFactors : IntegerArray;
    cls : String;
```

```
    i : Integer;
begin
    create hsa transient;
    create ctrNames transient;
    create ctrTypes transient;
    create ctrScaleFactors transient;
    node.wbemListClasses(hsa);
    if hsa.size > 0 then
        cls := hsa[1];
        hsa.purge;
        write "WBEM class : " & cls;
        node.wbemQueryQualifiers(cls, ctrNames, ctrTypes, ctrScaleFactors);
        foreach i in 1 to ctrNames.size do
            write "Attribute: " & ctrNames[i] &
                  " type: " & ctrTypes[i].String &
                  " scale factor: " & ctrScaleFactors[i].String;
        endforeach;
    endif;
epilog
    delete hsa;
    delete ctrNames;
    delete ctrTypes;
    delete ctrScaleFactors;
end;
```

The output from the **wbemQueryQualifiers** method shown in the previous example is as follows.

```
WBEM class : Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_
PerfRawData.Win32_PerfRawData_PerfDisk_LogicalDisk
Attribute: AvgDiskBytesPerRead type: 1073874176 scale factor: -2
Attribute: AvgDiskBytesPerRead_Base type: 1073939458 scale factor: 0
Attribute: AvgDiskBytesPerTransfer type: 1073874176 scale factor: -2
Attribute: AvgDiskBytesPerTransfer_Base type: 1073939458 scale factor: 0
Attribute: AvgDiskBytesPerWrite type: 1073874176 scale factor: -2
Attribute: AvgDiskBytesPerWrite_Base type: 1073939458 scale factor: 0
Attribute: AvgDiskQueueLength type: 5571840 scale factor: 2
Attribute: AvgDiskReadQueueLength type: 5571840 scale factor: 2
Attribute: AvgDisksecPerRead type: 805438464 scale factor: 3
Attribute: AvgDisksecPerRead_Base type: 1073939458 scale factor: 0
Attribute: AvgDisksecPerTransfer type: 805438464 scale factor: 3
Attribute: AvgDisksecPerTransfer_Base type: 1073939458 scale factor: 0
Attribute: AvgDisksecPerWrite type: 805438464 scale factor: 3
Attribute: AvgDisksecPerWrite_Base type: 1073939458 scale factor: 0
Attribute: AvgDiskWriteQueueLength type: 5571840 scale factor: 2
Attribute: Caption type: 0 scale factor: 0
Attribute: CurrentDiskQueueLength type: 65536 scale factor: 1
Attribute: Description type: 0 scale factor: 0
Attribute: DiskBytesPersec type: 272696576 scale factor: -4
Attribute: DiskReadBytesPersec type: 272696576 scale factor: -4
Attribute: DiskReadsPersec type: 272696320 scale factor: 0
Attribute: DiskTransfersPersec type: 272696320 scale factor: 0
Attribute: DiskWriteBytesPersec type: 272696576 scale factor: -4
Attribute: DiskWritesPersec type: 272696320 scale factor: 0
Attribute: FreeMegabytes type: 65536 scale factor: 0
Attribute: Frequency_Object type: 0 scale factor: 0
```

```
Attribute: Frequency_PerfTime type: 0 scale factor: 0
Attribute: Frequency_Sys100NS type: 0 scale factor: 0
Attribute: Name type: 0 scale factor: 0
Attribute: PercentDiskReadTime type: 542573824 scale factor: 0
Attribute: PercentDiskReadTime_Base type: 1073939712 scale factor: 0
Attribute: PercentDiskTime type: 542573824 scale factor: 0
Attribute: PercentDiskTime_Base type: 1073939712 scale factor: 0
Attribute: PercentDiskWriteTime type: 542573824 scale factor: 0
Attribute: PercentDiskWriteTime_Base type: 1073939712 scale factor: 0
Attribute: PercentFreeSpace type: 537003008 scale factor: 0
Attribute: PercentFreeSpace_Base type: 1073939459 scale factor: 0
Attribute: PercentIdleTime type: 542573824 scale factor: 0
Attribute: PercentIdleTime_Base type: 1073939712 scale factor: 0
Attribute: SplitIOPerSec type: 272696320 scale factor: 0
Attribute: Timestamp_Object type: 0 scale factor: 0
Attribute: Timestamp_PerfTime type: 0 scale factor: 0
Attribute: Timestamp_Sys100NS type: 0 scale factor: 0
```

wbemRetrieveData

Signature

```
wbemRetrieveData(className:     String;
                 instNameArray: HugeStringArray;
                 jdoArray:      JadeDynamicObjectArray input);
```

The **wbemRetrieveData** method of the **Node** class retrieves Web-Based Enterprise Management (WBEM) instances and attribute values for a specified WBEM class. The values are retrieved from the machine in which the node of the receiver object is running.

JADE allows only a subset of classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information.

A **JadeDynamicObject** instance is created for each WBEM instance that is retrieved and added to the instance of the **JadeDynamicObjectArray** specified by the **jdoArray** input parameter. If the array is transient, the **JadeDynamicObject** instances are transient (not shared). If the array is persistent, the instances are persistent.

The caller is responsible for creating and deleting the **JadeDynamicObjectArray** instance, and for deleting any **JadeDynamicObject** instances that are added to it.

The **instNameArray** parameter is used to select the set of WBEM instances that are retrieved. If used, the array should contain a set of strings representing the names of the WBEM instances to be retrieved. Only WBEM instances that have names that match entries in the array are returned. If the value of the **instNameArray** parameter is null, all instances for the class specified by the **className** parameter are returned.

As this method does not clear or purge the **JadeDynamicObjectArray** before inserting the **JadeDynamicObject** instances, if it is called multiple times without first calling the **purge** or **clear** methods, previously added entries will remain in the array.

Each **JadeDynamicObject** that is created contains attributes representing each attribute of the corresponding WBEM instance. The name of each attribute matches the WBEM class attribute name. The attribute value is one of the following types, depending on the corresponding WBEM attribute type:

- **Integer64**
- **String**
- **Real**

- **Decimal**
- **Boolean**

JADE converts WBEM attributes that are arrays into individual attributes with the array index inserted at the end of each attribute name. Although unlikely, applications should be prepared for the possibility that cumulative **Integer64** values will overflow to negative values. The maximum value before they overflow to a negative value is **2^63 - 1** (approximately 8 Exabytes).

There are some attributes that are not returned by JADE. These are mainly attributes that pertain to WBEM class and superclass names.

There is a limit of approximately 48K bytes to the size of the WBEM data that can be retrieved from remote nodes. If this limit is exceeded, exception 1141 is raised, in which case you should use the **instNameArray** parameter to restrict the number of WBEM class instances that are retrieved.

If a name that is not allowed or recognized is used, exception 1136 is raised.

The following example shows the use of the **wbemRetrieveData** method.

```
showWbemCPUInformation();
vars
   wbemClassName : String;
   jdoArray : JadeDynamicObjectArray;
   jdo : JadeDynamicObject;
begin
   wbemClassName := "Root.CIMV2.CIM_StatisticalInformation.Win32_Perf."
                  & "Win32_PerfFormattedData.Win32_PerfFormattedData_"
                  & "PerfOS_Processor";
   create jdoArray transient;
   node.wbemRetrieveData(wbemClassName, null, jdoArray);
   foreach jdo in jdoArray do
      write jdo.display;
   endforeach;
epilog
   jdoArray.purge;
   delete jdoArray;
end;
```

The following example shows the use of the **instNameArray** parameter to restrict the number of WBEM class instances retrieved by the **wbemRetrieveData** method.

```
showWbemLogicalDiskInformation();
vars
   wbemClassName : String;
   jdoArray : JadeDynamicObjectArray;
   hsa : HugeStringArray;
   jdo : JadeDynamicObject;
begin
   wbemClassName := "Root.CIMV2.CIM_StatisticalInformation.Win32_Perf."
                  & "Win32_PerfRawData.Win32_PerfRawData_PerfDisk_
                  & "LogicalDisk";
   create hsa transient;
   hsa[1] := "C:";
   hsa[2] := "E:";
   create jdoArray transient;
   node.wbemRetrieveData(wbemClassName, hsa, jdoArray);
```

```
   foreach jdo in jdoArray do
      write jdo.display;
   endforeach;
epilog
   delete hsa;
   jdoArray.purge;
   delete jdoArray;
end;
```

The output from the **wbemRetrieveData** method shown in the previous example is as follows.

```
Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_PerfRawData.Win32_
PerfRawData_PerfDisk_LogicalDisk(116)---
CIMPath = \\wilbur2a\root\cimv2:Win32_PerfRawData_PerfDisk_LogicalDisk.Name="C:"
CIMServer = WILBUR2A
CIMClass = Win32_PerfRawData_PerfDisk_LogicalDisk
AvgDiskBytesPerRead = 10004757504
AvgDiskBytesPerRead_Base = 450205
AvgDiskBytesPerTransfer = 27031866880
AvgDiskBytesPerTransfer_Base = 1234382
AvgDiskBytesPerWrite = 17027109376
AvgDiskBytesPerWrite_Base = 784177
AvgDiskQueueLength = 2183210108
AvgDiskReadQueueLength = 2379370099
AvgDisksecPerRead = 3604485205
AvgDisksecPerRead_Base = 450205
AvgDisksecPerTransfer = 1117796622
AvgDisksecPerTransfer_Base = 1234382
AvgDisksecPerWrite = 1808278713
AvgDisksecPerWrite_Base = 784177
AvgDiskWriteQueueLength = -196159991
Caption = NULL
CurrentDiskQueueLength = 0
Description = NULL
DiskBytesPersec = 27031866880
DiskReadBytesPersec = 10004757504
DiskReadsPersec = 450205
DiskTransfersPersec = 1234382
DiskWriteBytesPersec = 17027109376
DiskWritesPersec = 784177
FreeMegabytes = 15309
Frequency_Object = 0
Frequency_PerfTime = 3192100000
Frequency_Sys100NS = 10000000
Name = C:
PercentDiskReadTime = 2379370099
PercentDiskReadTime_Base = 128217570284699662
PercentDiskTime = 2183210108
PercentDiskTime_Base = 128217570284699662
PercentDiskWriteTime = -196159991
PercentDiskWriteTime_Base = 128217570284699662
PercentFreeSpace = 15309
PercentFreeSpace_Base = 72763
PercentIdleTime = -742782462
PercentIdleTime_Base = 128217570284699662
```

```
SplitIOPerSec = 69672
Timestamp_Object = 0
Timestamp_PerfTime = 1651626206541880
Timestamp_Sys100NS = 128217570284699662
---Root.CIMV2.CIM_StatisticalInformation.Win32_Perf.Win32_PerfRawData.Win32_
PerfRawData_PerfDisk_LogicalDisk(116)---
CIMPath = \\wilbur2a\root\cimv2:Win32_PerfRawData_PerfDisk_LogicalDisk.Name="E:"
CIMServer = WILBUR2A
CIMClass = Win32_PerfRawData_PerfDisk_LogicalDisk
AvgDiskBytesPerRead = 9827585536
AvgDiskBytesPerRead_Base = 290889
AvgDiskBytesPerTransfer = 15723511808
AvgDiskBytesPerTransfer_Base = 397212
AvgDiskBytesPerWrite = 5895926272
AvgDiskBytesPerWrite_Base = 106323
AvgDiskQueueLength = 3881830496
AvgDiskReadQueueLength = 2274370873
AvgDisksecPerRead = 152453346
AvgDisksecPerRead_Base = 290889
AvgDisksecPerTransfer = 2168531380
AvgDisksecPerTransfer_Base = 397212
AvgDisksecPerWrite = 2016078033
AvgDisksecPerWrite_Base = 106323
AvgDiskWriteQueueLength = 1607459623
Caption = NULL
CurrentDiskQueueLength = 0
Description = NULL
DiskBytesPersec = 15723511808
DiskReadBytesPersec = 9827585536
DiskReadsPersec = 290889
DiskTransfersPersec = 397212
DiskWriteBytesPersec = 5895926272
DiskWritesPersec = 106323
FreeMegabytes = 103216
Frequency_Object = 0
Frequency_PerfTime = 3192100000
Frequency_Sys100NS = 10000000
Name = E:
PercentDiskReadTime = 2274370873
PercentDiskReadTime_Base = 128217570284699662
PercentDiskTime = 3881830496
PercentDiskTime_Base = 128217570284699662
PercentDiskWriteTime = 1607459623
PercentDiskWriteTime_Base = 128217570284699662
PercentFreeSpace = 103216
PercentFreeSpace_Base = 190779
PercentIdleTime = 1794241932
PercentIdleTime_Base = 128217570284699662
SplitIOPerSec = 6177
Timestamp_Object = 0
Timestamp_PerfTime = 1651626206541880
Timestamp_Sys100NS = 128217570284699662
```

NormalException Class

The **NormalException** class is the superclass of all non-fatal exceptions. You may occasionally want to define exceptions other than those automatically captured by the system. In this case, create a subclass of the **NormalException** class in order to add new properties and methods specific to your own exception protocol or to override system methods such as the **showDialog** method.

Inherits From: Exception

Inherited By: ConnectionException, FileException, JadeMessagingException, JadeSOAPException, JadeXMLException, ODBCException, SystemException, user-defined exception classes, UserInterfaceException, WebSocketException

The method in the following example arms the exception handler so that the **exceptionHandler** method is called when an exception of the **NormalException** class is encountered and is passed the exception object as a parameter.

```
method1();
begin
    on NormalException do exceptionHandler(exception);
    self.method2;
    status.caption := "Resuming execution after exception throwing
                       method invocation";
end;
```

The method in the following example causes a *1035 - String Too Long* exception, by assigning a four-character string to a variable defined as being a three-character string.

```
method2();
vars
    str : String[3];
begin
    str := "Long string value";
end;
```

The method in the following example arms the exception handler so that the **exceptionHandler** method is called when an exception of the **UserException** class is raised and is passed the exception object as a parameter.

```
method3();
begin
    on UserException do exceptionHandler(exception);
    self.method4;
end;
```

The method in the following example creates an object of the **UserException** class and defines the properties for this object. The exception is then raised.

```
method4();
vars
    ex : UserException;
begin
    create ex;
    ex.errorCode   := 64000;
    ex.continuable := true;
    ex.resumable   := true;
    raise ex;
```

```
    status.caption := "Resuming execution after raising of exception";
end;
```

The following is an example of an **exceptionHandler** method in a **UserException** subclass of the **NormalException** class.

```
exceptionHandler(ex: NormalException): Integer;
vars
    returnCode : Integer;
begin
    // Exception handling method specified when arming the handler, and
    // called when the appropriate exceptions are raised.  If the error
    // code of the exception is 1035, the exception is identified as
    // being the String Too Long exception and is handled appropriately.
    if ex.errorCode = 1035 then
        returnCode := app.msgBox("String too long.  Resume execution after
                                  method?", "String too long", 52);
        if returnCode = MsgBox_Return_Yes then
            // The Ex_Resume_Next return value passes control back to
            // the method that armed the exception handler (in this
            // case, method1) and resumes execution after the
            // invocation of the method that raised the exception.
            return Ex_Resume_Next;
        else
            // The Ex_Abort_Action return value causes all currently
            // executing methods to be aborted.  In this case, the
            // application reverts to execution after the invocation
            // of the method that raised the idle state, and awaits
            // further user input.
            status.caption := "Aborting all currently executing methods";
            return Ex_Abort_Action;
        endif;
    // If the error code of the exception is 64000, the exception is
    // identified as the user-defined exception that was assigned this
    // code, and is handled appropriately.
    elseif ex.errorCode = 64000 then
        returnCode := app.msgBox("User-defined exception.  Continue method
                                  execution?", "User-defined exception", 52);
        if returnCode = MsgBox_Return_Yes then
            // The Ex_Continue return value passes control back to the
            // method that raised the exception handler (in this case,
            // method4) and resumes execution after raising the exception.
            return Ex_Continue;
        else
            // The Ex_Resume_Next return value passes control back to the
            // method that armed the exception handler (in this case,
            // method1) and resumes execution after the invocation of
            // the method that raised the exception.
            status.caption := "Resuming execution after exception throwing
                               method invocation";
            return Ex_Resume_Next;
        endif;
    endif;
end;
```

Notification Class

Instances of the **Notification** class are used to describe the notifications registered by the system. JADE notifications may have a differing execution order when intermixed with **Window** events in JADE thin client mode. This difference arises because the notifications occur on the application server rather than the presentation client.

Notifications are usually interlaced with any **Window** events that may occur. In thin client mode, the notification occurs when the application server thread processing the presentation client operations becomes idle. However, the presentation client may also be idle and send event notifications such as form activations, focus changes, and so on, at the same time. This asynchronous operation may result in a slightly different execution order for these events from that experienced in JADE when it is not running in thin client mode.

For details about the properties and methods defined in the **Notification** class, see "Notification Properties" and "Notification Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

Notification Properties

The properties defined in the **Notification** class are summarized in the following table.

Property	Description
elapsedTime	Contains the time that the notification request has been in place
eventType	Contains the event type of the notification request
featureNumber	Contains the interface number and method number that allows identification of the interface method that was mapped by the subscriber
isInterface	Specifies whether the notification was registered by an interface notification method
requestedBy	Contains the process that submitted the notification request
requestTime	Contains the date and time of the notification request
responseType	Contains the response type of the notification request
serialNumber	Contains the serial number that is internally assigned by JADE
typeNumber	Contains a number that allows identification of the associated interface
userTag	Contains the user tag of the notification request

elapsedTime

Type: Time

The read-only **elapsedTime** property of the **Notification** class is set to the time that the notification request has been in place.

eventType

Type: Integer

The read-only **eventType** property of the **Notification** class is set to the **eventType** parameter of the notification request.

The **eventType** property specifies the type of event for which the notification is requested. The global constants in the **SystemEvents** category for the types of event that can be requested are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

featureNumber

Type: Integer

If the notification was registered by the **Object** class **beginClassNotificationForIF**, **beginClassesNotificationForIF**, or **beginNotificationForIF** method, the read-only **featureNumber** property of the **Notification** class contains the identifying number of the interface method that was mapped by the subscriber to receive an event notification.

If the notification was registered by the **Object** class **beginClassNotification**, **beginClassesNotification**, or **beginNotification** method, this property contains zero (**0**).

isInterface

Type: Boolean

The read-only **isInterface** property of the **Notification** class contains **true** if the notification was registered by the **Object** class **beginClassNotificationForIF**, **beginClassesNotificationForIF**, or **beginNotificationForIF** method.

If the notification was registered by the **Object** class **beginClassNotification**, **beginClassesNotification**, or **beginNotification** method, this property contains **false** and the **featureNumber** and **typeNumber** properties contain zero (**0**).

requestedBy

Type: Process

The **requestedBy** property of the **Notification** class is set to the process that submitted the notification request.

requestTime

Type: TimeStamp

The read-only **requestTime** property of the **Notification** class is set to date and time of the notification request.

responseType

Type: Integer

The read-only **responseType** property of the **Notification** class is set to the **responseType** parameter of the notification request.

The **responseType** parameter specifies the frequency with which an event notification is to be sent.

The **NotificationResponses** category global constants for the types of response that can be sent are listed in the following table.

Global Constant	Integer Value	Sends a notification…
Response_Cancel	1	When the class receives a matching event and then cancels the notification
Response_Continuous	0	Whenever the class receives a matching event
Response_Suspend	2	When the class receives a matching event and then suspends notification until the user refreshes the local copy of the class

serialNumber

Type: Integer

The **serialNumber** read-only property of the **Notification** class is assigned internally by JADE.

typeNumber

Type: Integer

If the notification was registered by the **Object** class **beginClassNotificationForIF**, **beginClassesNotificationForIF**, or **beginNotificationForIF** method, the read-only **typeNumber** property of the **Notification** class contains the identifying number of the associated interface.

If the notification was registered by the **Object** class **beginClassNotification**, **beginClassesNotification**, or **beginNotification** method, this property contains zero (**0**).

userTag

Type: Integer

The **userTag** read-only property of the **Notification** class is set to the **eventTag** parameter of the notification request.

The **eventTag** parameter specifies an integer value (for example, an index into an array) that is returned with each notification.

Notification Methods

The methods defined in the **Notification** class are summarized in the following table.

Method	Description
subscriber	Returns the object to which the notification is to be delivered
target	Returns the object that is the target of the notification request

subscriber

Signature `subscriber(): Object;`

The **subscriber** method of the **Notification** class returns a reference to the object to which the notification is delivered.

The following example shows the use of the **subscriber** method.

```
vars
    notification      : Notification;
    notificationArray : NotificationArray;
begin
    create notificationArray transient;
    system.getNotes(notificationArray, true, 100);
        foreach notification in notificationArray do
            // access the notification entry properties
            write notification.target.String;
            // now check that the subscriber class is valid for this user
            if app.isValidObject(notification.subscriber) then
                write notification.subscriber.String;
            endif;
        endforeach;
epilog
    notificationArray.purge;
    delete notificationArray;
end;
```

target

Signature `target(): Object;`

The **target** method of the **Notification** class returns a reference to the object that is the target of a notification request.

For an example of the use of the **target** method, see the **subscriber** method.

NotificationArray Class

The **NotificationArray** class is the transient class that encapsulates behavior required to access **Notification** objects in an array.

The notifications are referenced by their position in the collection.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a notification array.

Inherits From: ObjectArray

Inherited By: (None)

NotificationException Class

The **NotificationException** class is the transient class that defines behavior for exceptions that occur as a result of notification events when the subscriber cannot be found. For details about the method defined in the **NotificationException** class, see "NotificationException Method", in the following subsection.

Inherits From: SystemException

Inherited By: (None)

NotificationException Method

The method defined in the **NotificationException** class is summarized in the following table.

Method	Description
notificationTarget	Returns the reference to the target for the notification whose subscriber was not found

notificationTarget

Signature `notificationTarget(): Object;`

The **notificationTarget** method of the **NotificationException** class returns a reference to the target for the notification whose subscriber was not found.

The **Exception** class **errorObject** method returns a reference to this subscriber.

Note You should use this subscriber reference only to examine the object id (oid). Do not attempt to reference the object itself, as it has just been determined that it could not be found.

The code fragment in the following example shows the use of the **Object** class **getOidStringForObject** method to determine the oid of the object that was not found.

```
write getOidStringForObject(errorObject);
```

NumberFormat Class

The **NumberFormat** class is used to store Windows locale numeric information.

You cannot modify system-created instances of the **NumberFormat** class (that is, instances created and maintained by JADE to store locale information and user-defined formats) from your JADE code.

JADE automatically creates a transient instance of **NumberFormat** for each application that you can read by using **app.currentLocaleInfo.numericInfo**. This instance contains numeric information for the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

For details about returning a string containing the receiver in the supplied number format, see the **userNumberFormat** method in the **Integer**, **Real**, or **Decimal** primitive type.

NumberFormat instances are also used to store user-defined numeric formats that can be passed to the various primitive type user format methods. You can maintain these formats only by using the appropriate Formats menu command, accessed from the Format Browser.

For details about the constants, properties, and method defined in the **NumberFormat** class, see "NumberFormat Class Constants", "NumberFormat Properties", and "NumberFormat Method", in the following subsections.

Inherits From: LocaleFormat

Inherited By: CurrencyFormat

NumberFormat Class Constants

The constants provided by the **NumberFormat** class are listed in the following table.

Constant	Integer Value	Example
NegNumBrackets	0	(10.25)
NegNumLeadingSign	1	-10.25
NegNumLeadingSignSpace	2	- 10.25 (note the space after -)
NegNumTrailingSign	3	10.25-
NegNumTrailingSpaceSign	4	10.25 - (note the space before -)

NumberFormat Properties

The properties defined in the **NumberFormat** class are summarized in the following table.

Property	Description
decimalPlaces	Contains the number of digits to the right of the decimal separator
decimalSeparator	Contains the character used to separate the integer and fractional parts of numbers
groupings	Contains the size for each group of digits to the left of the decimal

Property	Description
negativeFormat	Contains the negative number format
negativeSign	Contains the string used to represent the negative sign
positiveSign	Contains the string used to represent the positive sign
showLeadingZeros	Specifies if zero (**0**) is displayed in front of a number that is less than 1
thousandSeparator	Contains the character used to separate groups of digits left of the decimal separator

decimalPlaces

Type: Integer

The **decimalPlaces** property of the **NumberFormat** class contains the number of digits to the right of the decimal point in numbers.

decimalSeparator

Type: String[20]

The **decimalSeparator** property of the **NumberFormat** class contains the character used to separate the integer part from the fractional part of a number.

groupings

Type: String[80]

The **groupings** property of the **NumberFormat** class contains the size for each group of digits to the left of the decimal. An explicit size is specified for each group, separated by semicolons.

If the last value is zero (**0**), the preceding value is repeated; for example, to group numbers by three digits, **3;0** is specified.

negativeFormat

Type: Integer

The **negativeFormat** property of the **NumberFormat** class contains the negative number format.

The **NumberFormat** class constants that represent the format for negative numbers are listed in the following table.

NumberFormat Class Constant	Integer Value	Example
NegNumBrackets	0	(10.25)
NegNumLeadingSign	1	-10.25
NegNumLeadingSignSpace	2	- 10.25 (note the space after -)
NegNumTrailingSign	3	10.25-
NegNumTrailingSpaceSign	4	10.25 - (note the space before -)

negativeSign

Type: String[20]

The **negativeSign** property of the **NumberFormat** class contains the string value for the negative sign; for example, **-**.

positiveSign

Type: String[20]

The **positiveSign** property of the **NumberFormat** class contains the string value for the positive sign; for example, **+**.

showLeadingZeros

Type: Boolean

The **showLeadingZeros** property of the **NumberFormat** class is set to **true** if a leading zero is to be displayed in front of a number that is less than 1; for example, **0.7**.

thousandSeparator

Type: String[20]

The **thousandSeparator** property of the **NumberFormat** class contains the character used to separate groups of digits to the left of the decimal separator.

NumberFormat Method

The method defined in the **NumberFormat** class is summarized in the following table.

Property	Description
defineNumberFormat	Defines the characteristics of a number format

defineNumberFormat

Signature

```
defineNumberFormat(numberOfDecimalPlaces: Integer;
                   decimalSep:            String;
                   thousandSep:           String;
                   negFormat:             Integer;
                   showLeadingZero:       Boolean) updating;
```

The **defineNumberFormat** method of the **NumberFormat** class enables you to dynamically define the characteristics of a number format. Set the **numberOfDecimalPlaces** parameter to the number of decimal places that you want displayed, in the range **0** through **9**. A value of zero (**0**) is assumed if you specify a value less than **0**. Conversely, a value of **9** is assumed if you specify a value greater than **9**.

The **decimalSep** and **thousandSep** parameters enable you to specify a string of up to three characters that is to separate decimals from the rest of the number and to separate thousands, respectively. If the strings contain any numeric characters, these numeric characters are removed. If the strings are longer than three characters, they are truncated to three characters.

If you do not specify one of the **NumberFormat** class constants listed in the following table, **NumberFormat**.**NegNumLeadingSign** is assumed.

NumberFormat Class Constant	Integer Value	Example
NegNumBrackets	0	(10.25)
NegNumLeadingSign	1	-10.25
NegNumLeadingSignSpace	2	- 10.25 (note the space after -)
NegNumTrailingSign	3	10.25-
NegNumTrailingSpaceSign	4	10.25 - (note the space before -)

Set the **showLeadingZero** parameter to **true** if you want to display a leading zero (**0**) for numbers in the range **1** through **-1**. Alternatively, set the parameter to **false** if you do not want to display a leading zero. (For details about returning a string containing the receiver in the supplied number format, see the **userNumberFormat** method in the **Integer**, **Real**, or **Decimal** primitive type.)

Object Class

The **Object** class is the base class (superclass) for all system-defined and user-defined classes. The **Object** class defines default behavior for all other classes in the schema.

For details about the methods defined in the **Object** class, see "Object Methods", in the following subsection. Refer also to "Global Constants Reference", in Appendix A of the *JADE Encyclopaedia of Primitive Types*, for a reference to the JADE global constants.

Object Methods

The methods defined in the **Object** class are summarized in the following table.

Method	Description
autoPartitionIndex	Returns the partition index of the database file partition in which the receiver is stored on creation
beginClassNotification	Registers the receiver to be notified when a nominated event occurs on instances of a class and its subclasses
beginClassNotificationForIF	Registers the receiver mapped to the **userNotification** and **sysNotification** methods of the **theInterface** parameter to be notified when a nominated event occurs on instances of a class and its subclasses, rather than to those of the subscriber
beginClassesNotification	Registers the receiver to be notified when a nominated event occurs on instances of a class and optionally its subclasses, without any additional searches for subschema copies in the current schema
beginClassesNotificationForIF	Registers the receiver mapped to the **userNotification** and **sysNotification** methods of the **theInterface** parameter to be notified when a nominated event occurs on instances of a class and optionally its subclasses, rather than to those of the subscriber
beginNotification	Registers the receiver to be notified when a nominated event occurs on a specified object (or all objects) of a class or its subclasses
beginNotificationForIF	Registers the receiver mapped to the **userNotification** and **sysNotification** methods of the **theInterface** parameter to be notified when a nominated event occurs on instances of an object, rather than to those of the subscriber
beginTimer	Arms a timer on the receiver, and registers the receiver for timer notification
beginTimerForIF	Arms a timer for methods mapped to the **timerEvent** method of the **theInterface** parameter, rather than to that of the subscriber
causeEvent	Causes a user event
changeObjectVolatility	Changes the volatility state of the specified persistent object
class	Returns the class of the receiver
cloneSelf	Creates a new instance of the same type as the receiver but does not invoke constructors
cloneSelfAs	Creates a new instance of the specified class, copying attributes defined in a "common ancestor" class, but does not invoke constructors

Method	Description
copySelf	Creates a new instance of the same type as the receiver, invoking constructor methods if defined
copySelfAs	Creates an instance of the specified class, invoking constructor methods if defined, and copying attributes defined in a "common ancestor" class
creationTime	Returns the date and time as a timestamp that the receiver was created
deletePropertyValue	Sets the value of the property specified by the name parameter to null
display	Returns a string containing a description of the receiver
edition	Returns the edition of the receiver as an integer value
endClassNotification	Ends a notification registered using a **beginClassNotification** method for the corresponding parameters
endClassNotificationForIF	Ends a notification registered using a **beginClassNotificationForIF** method for the corresponding parameters
endClassesNotification	Ends a notification registered using a **beginClassesNotification** method for the corresponding parameters
endClassesNotificationForIF	Ends a notification registered using a **beginClassesNotificationForIF** method for the corresponding parameters
endNotification	Ends a notification registered using a **beginNotification** method for the corresponding parameters
endNotificationForIF	Ends a notification registered using a **beginNotificationForIF** method for the corresponding parameters
endNotificationForSubscriber	Terminates all notifications registered by a specified subscriber for the corresponding parameters
endTimer	Terminates a timer that was initiated by using the **beginTimer** method for the corresponding parameters
endTimerForIF	Terminates a timer that was initiated by using the **beginTimerForIF** method for the corresponding parameters
exclusiveLock	Attempts to acquire an exclusive lock on the specified object
getClassForObject	Returns a reference to the class of the object identifier (oid) specified in the **obj** parameter
getClassNumberForObject	Returns the class number of the specified object
getInstanceIdForObject	Returns the instance identifier of the specified object as a **Decimal** value
getInstanceIdForObject64	Returns the instance identifier of the specified object as an **Integer64** value
getLockCallStack	Returns the lock call stack for a specified locked object
getLockStatus	Gets the status of the specified lock for the current process
getModifiedBy	Returns a string containing the user name of the user who modified the receiver
getName	Returns a string containing the class of the receiver
getObjectStringForObject	Returns a string representing the specified object as an oid-like string based on class numbers followed by an optional lifetime indication

Method	Description
getObjectVolatility	Returns the volatility state of the specified object
getOidString	Returns the object id (oid) of the receiver in a string format
getOidStringForObject	Returns the object id (oid) in a string format for the specified object
getOwnerForObject	Returns the owner (parent) for the specified collection
getPropertyValue	Returns the value of the property specified by the name parameter
getTimerStatus	Returns the status of a specified timer if it is currently active
getTimerStatusForIF	Returns the status of a specified timer that was initiated using the **beginTimerForIF** method, if it is currently active
getUpdateTranID	Returns the transaction identifier of the transaction that created or last updated the receiver
hasMembers	Returns **true** if the specified persistent, exclusive collection has any members
inspect	Opens an Inspector form for the receiver object
inspectModal	Opens a modal Inspector form for the receiver object
invokeIOMethod	Sends the specified target method containing a variable list of parameters to the receiver, after switching to the specified **targetContext** execution context
invokeMethod	Sends the specified target method containing a variable list of parameters to the receiver, after switching to the specified **targetContext** execution context
isImportedObject	Returns **true** if the specified object is an instance of an imported class
isKindOf	Returns **true** if the receiver is the kind of the specified class
isLockedByMe	Returns **true** if the executing process is the owner of the lock on the specified object
isObjectFrozen	Returns **true** if the volatility state of the specified object is frozen (that is, it is not updated)
isObjectNonSharedTransient	Returns **true** if the specified object is a non-shared transient instance
isObjectPersistent	Returns **true** if the specified object is a persistent instance
isObjectSharedTransient	Returns **true** if the specified object is a shared transient instance
isObjectStable	Returns **true** if the volatility state of the specified object is stable (that is, it is not updated frequently)
isObjectTransient	Returns **true** if the specified object is a shared or a non-shared transient instance
isObjectVolatile	Returns **true** if the volatility state of the specified object is volatile (that is, it is often updated)
isSharedTransient	Returns **true** if the receiver is a shared transient object
isSystemObject	Returns **true** if the receiver is an instance of a system class
isTransient	Returns **true** if the receiver is a transient object
jadeReportWriterCheck	Returns **true** when reimplemented in user subclasses for object instance security of JADE Report Writer reports

Method	Description
jadeReportWriterDisplay	Returns the text of the combo box entry for each object returned by the **Application** class **jadeReportWriterParamObjects** method
latestEdition	Returns the latest edition of the receiver
lock	Attempts to acquires the specified type of lock for a specific object
makeObjectFrozen	Changes the volatility state of the specified object to frozen
makeObjectStable	Changes the volatility state of the specified object to stable
makeObjectVolatile	Changes the volatility state of the specified object to volatile
moveToPartition	Moves the receiver to the specified partition
reserveLock	Attempts to acquire a reserve lock on the specified object
respondsTo	Returns **true** if the receiver's class or its superclasses implement the specified JADE interface
resynch	Marks the receiver as obsolete
resynchObject	Marks the specified object as obsolete
sdeCauseEvent	Causes an inter-system event notification in a Synchronized Database Environment (SDE)
sdsCauseEvent	Causes an inter-system event notification in a Synchronized Database Service (SDS)
sendMsg	Sends the specified message (a valid method) to the receiver and executes the specified method or condition
sendMsgWithIOParams	Sends the specified message (a valid method or condition) with a variable parameter list to the receiver and executes the specified method or condition
sendMsgWithParams	Sends the specified message (a valid method or condition) with a variable parameter list to the receiver and executes the specified method or condition
sendTypeMsg	Sends the specified message (a valid type method) to the receiver and executes the specified method
sendTypeMsgWithIOParams	Sends the specified message (a valid type method) with a variable parameter list to the receiver and executes the specified method
sendTypeMsgWithParams	Sends the specified message (a valid type method) with a variable parameter list to the receiver and executes the specified method
setPartitionID	Specifies the absolute partition ID in which to locate the receiver
setPartitionIndex	Specifies the partition in which to locate the receiver
setPropertyValue	Sets the property of the receiver to the specified value
sharedLock	Attempts to acquire a shared lock on the specified object
sysNotification	Called by the system when a subscribed system event occurs
timerEvent	Invoked by the system when the timer period expires
tryGetPropertyValue	Returns the value of the specified property, if it exists; otherwise returns **false**
tryLock	Attempts to acquire a lock of the specified type

Method	Description
unlock	Removes the current lock from the specified object
updateLock	Attempts to acquire an update lock on the specified object
updateObjectEdition	Increments the edition of the specified object by one (**1**)
userNotification	Called by the system when a subscribed user event occurs
version	Returns the version of the object

autoPartitionIndex

Signature `autoPartitionIndex(): Integer partitionMethod;`

The **autoPartitionIndex** method of the **Object** class returns the partition index of the database file partition in which an object is stored on creation. The method is automatically called when an instance of a partitioned class is created.

A partition index value of zero (**0**) refers to the latest partition created, a partition index value of one (**1**) to the second latest, and so on.

The **autoPartitionIndex** method returns the value zero, so objects are created by default in the latest partition. However, you can reimplement the method in user classes to override the default behavior. For more details, see "partitionMethod Option", in Chapter 1 of the *JADE Developer's Reference*.

Note The **autoPartitionIndex** method cannot be used if the database file for that object is encrypted, as the database cannot invoke the **autoPartitionIndex** method using an encrypted buffer. If this occurs, exception 3009 (*File encrypted and partition unspecified*) is raised.

If the file is encrypted, use the **Object** class **setPartitionID** or **setPartitionIndex** method to explicitly set the partition in the created object.

beginClassNotification

Signature
```
beginClassNotification(theClass:     Class;
                       transients:   Boolean;
                       eventType:    Integer;
                       responseType: Integer;
                       eventTag:     Integer);
```

The **beginClassNotification** method of the **Object** class registers the receiver to be notified when a nominated event occurs on instances of a class and its subclasses.

The object that invokes the **beginClassNotification** method is referred to as the *subscriber*. An object that subscribes to a class notification is notified when the nominated event occurs for *any* instance of the specified class or its subclasses defined in the schema of the specified class. The class instances that are subscribed to are referred to as *notification targets*.

For notifications on persistent instances, you can subscribe to the following types of event:

- System events, which include object creation, deletion, or change. System events are notified automatically by the system when the nominated persistent objects are committed (using the **commitTransaction** instruction). Each of the target objects is automatically resynchronized in the client cache.
- User events, which are specified by selecting an **eventType** parameter in the range **User_Base_Event**

through **User_Max_Event**. User events are notified when the **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method of the **Object** class is invoked on a target instance.

For notifications on transient or shared transient instances, you can only subscribe to user events.

A process that uses the **beginClassNotification** method to subscribe to user event notifications for transient instances will receive notifications for all shared transient instances and for those non-shared transient instances that it has created (that is, the process will not receive notifications for non-shared transient instances that have been created by other processes).

Non-GUI objects (that is, objects that are not instances of a **Window** subclass) respond to system notifications by implementing the **sysNotification** method. GUI objects (instances of a **Window** subclass) respond to system notifications by implementing their **sysNotify** event method.

If a form or control has an attached window, a requested user notification is directed to the **userNotify** event and a requested system notification to the **sysNotify** event.

The subscription to a notification registered by the **beginClassNotification** method is terminated by the **endClassNotification** method. You can also use the **endNotificationForSubscriber** method to terminate all notifications for a specific subscriber.

Caution If the application will be run in JADE thin client mode, subscribe to notifications and timers with care. When an event occurs, each registered subscriber is notified on the application server. In thin client mode, the first notification or timer that is sent causes a further message to be sent to the presentation client to post a message into the Windows message queue of the client. (This is necessary so that the subsequent execution of the notification logic is synchronized with what is taking place on the presentation client.)

When that Windows message is processed by the presentation client, another message is sent to the application server to initiate the processing of the first 10 queued notifications or timers for that client. If there are more than 10 notifications, these actions are repeated until all queued events are processed.

Notifications and timers could therefore have a considerable impact on network traffic.

For an example of the **beginClassNotification** method, see "Example of Beginning Notifications", under the **Object** class **beginNotification** method.

The **beginClassNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Specifies …
theClass	The class for which the notification is to be invoked
transients	If the user notification is invoked for events occurring in transient instances
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value that is required for each notification

theClass

Use the **theClass** parameter of the **beginClassNotification** method to specify the class for which the notification is to be invoked.

transients

Use the **transients** parameter of the **beginClassNotification** method to specify if the user notification is to be invoked for events that occur to transient instances (**true**) or persistent instances (**false**) of the class.

Note You can subscribe to system notifications only for persistent objects; that is, the **transients** parameter must be **false**.

eventType

Use the **eventType** parameter of the **beginClassNotification** method to specify the type of event for which the notification is requested.

The global constants in the **SystemEvents** category for the types of event that can be subscribed to are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

responseType

Use the **responseType** parameter of the **beginClassNotification** method to specify the frequency with which an event notification is notified.

The valid values for the **responseType** parameter, represented by global constants in the **NotificationResponses** category, are listed in the following table.

Global Constant	Integer Value	Sends a notification…
Response_Cancel	1	When an instance of the class receives a matching event and then cancels the notification
Response_Continuous	0	Whenever an instance of the class receives a matching event
Response_Suspend	2	When an instance of the class receives a matching event and then suspends notification until the user refreshes the local copy of the instance of the class

eventTag

Use the **eventTag** parameter of the **beginClassNotification** method to identify a specific notification when you have multiple subscriptions for the same event type.

beginClassNotificationForIF

Signature
```
beginClassNotificationForIF(theClass:     Class;
                            transients:   Boolean;
                            eventType:    Integer;
                            responseType: Integer;
                            eventTag:     Integer;
                            theInterface: JadeInterface);
```

The **beginClassNotificationForIF** method of the **Object** class is a variation of the **beginClassNotification** method.

The **beginClassNotificationForIF** method allows notification events to be sent to methods mapped to the **userNotification** and **sysNotification** methods of the **theInterface** parameter when a nominated event occurs on instances of a class and its subclasses, rather than to those of the subscriber.

The subscriber must be an instance of a **Class** that includes methods that map to (implement) the specified **sysNotification** or **userNotification** method of the interface. The parameters specified in the method signatures must match the standard **userNotification** or **sysNotification** method, as follows.

```
sysNotification(eventType: Integer; theObject: Object; eventTag: Integer);

userNotification(eventType: Integer; theObject: Object; eventTag: Integer;
        userInfo: Any);
```

If the method signature does not match the standard **userNotification** or **sysNotification** method, an exception is raised when the **beginClassNotificationForIF** method is executed. The request for a notification registered by the **beginClassNotificationForIF** method is terminated by the **endClassNotificationForIF** method. You can also use the **endNotificationForSubscriber** method to terminate all previous notifications for a specified subscriber.

The **beginClassNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies…
theClass	The interface implementation class for which the notification is to be invoked
transients	If the user notification is invoked for events occurring in transient instances
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value that is required for each notification
theInterface	The interface implemented by the specified class and its subclasses

With the exception of the **theInterface** parameter, described in the following subsection, see the appropriate subsections of the **beginClassNotification** method for details about the other parameters.

theInterface

Use the **theInterface** parameter of the **beginClassNotificationForIF** class to specify the interface that defines the appropriate **userNotification** or **sysNotification** method.

If the value of the **eventType** parameter is a system event, events are sent to the method that maps to the **sysNotification** method of the interface. If this parameter does not specify a system event, events are sent to the method that maps to the **userNotification** method of the interface.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver class and its subclasses must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

beginClassesNotification

Signature

```
beginClassesNotification(theClass:          Class;
                         includeSubclasses: Boolean;
                         transients:        Boolean;
                         eventType:         Integer;
                         responseType:      Integer;
                         eventTag:          Integer);
```

The **beginClassesNotification** method of the **Object** class registers the receiver to be notified when a nominated event occurs on instances of a class and optionally its subclasses.

The object that invokes the **beginClassesNotification** method is referred to as the *subscriber*.

The **beginClassesNotification** method does not attempt to re-interpret the value of the **theClass** parameter, so that the call does not look for a subschema copy class in the current schema with which to register this call and it optionally allows only the class without any of its subclasses to be registered for the notification. For example, a **beginClassNotification(MemberKeyDictionary, false, Any_System_Event, Response_Continuous, 1)** call looks for a subschema copy class in the current schema to register, and a **beginClassesNotification** (**MemberKeyDictionary, false, false, Any_System_Event, Response_Continuous, 1)** call registers the root **MemberKeyDictionary** class in the **RootSchema**.

If you want to specify your **MemberKeyDictionary** subschema copy class and allow both the class *and* its subclasses to be registered for the notification, call **beginClassesNotification** as follows.

```
beginClassesNotification(currentSchema.getClass('MemberKeyDictionary'),
                         true, false, Any_System_Event, Response_Continuous, 1);
```

An object that subscribes to a class notification is notified when the nominated event occurs for *any* instance of the specified class or its subclasses (when the **includeSubclasses** parameter is set to **true**). The class instances that are subscribed to are referred to as *notification targets*.

For notifications on persistent instances, you can subscribe to the following types of event:

- System events, which include object creation, deletion, or change. System events are notified automatically by the system when the nominated persistent objects are committed (using the **commitTransaction** instruction). Each of the target objects is automatically resynchronized in the client cache.
- User events, which are specified by selecting an **eventType** parameter in the range **User_Base_Event** through **User_Max_Event**. User events are notified when the **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method of the **Object** class is invoked on a target instance.

For notifications on transient instances, you can only subscribe to user events.

A process that uses the **beginClassesNotification** method to subscribe to user event notifications for transient instances will receive notifications for all shared transient instances and for those non-shared transient instances that it has created (that is, the process will not receive notifications for non-shared transient instances that have been created by other processes).

Non-GUI objects (that is, objects that are not instances of a **Window** subclass) respond to system notifications by implementing the **sysNotification** method. GUI objects (instances of a **Window** subclass) respond to system notifications by implementing their **sysNotify** event method. If a form or control has an attached window, a requested user notification is directed to the **userNotify** event and a requested system notification to the **sysNotify** event.

The subscription to a notification registered by the **beginClassesNotification** method is terminated by the **endClassesNotification** method. You can also use the **endNotificationForSubscriber** method to terminate all notifications for a specific subscriber.

Caution If the application will be run in JADE thin client mode, subscribe to notifications and timers with care. When an event occurs, each registered subscriber is notified on the application server.

In thin client mode, the first notification or timer that is sent causes a further message to be sent to the presentation client to post a message into the Windows message queue of the client. (This is necessary so that the subsequent execution of the notification logic is synchronized with what is taking place on the presentation client.)

When that Windows message is processed by the presentation client, another message is sent to the application server to initiate the processing of the first 10 queued notifications or timers for that client. If there are more than 10 notifications, these actions are repeated until all queued events are processed. Notifications and timers could therefore have a considerable impact on network traffic.

The **beginClassesNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Specifies …
theClass	The class for which the notification is to be invoked
includeSubclasses	Whether subclasses are included in or excluded from the notification registration
transients	If the user notification is invoked for events occurring in transient instances
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value that is required for each notification

theClass

Use the **theClass** parameter of the **beginClassesNotification** method to specify the class for which the notification is to be invoked.

Note This method does not attempt to re-interpret the value of the **theClass** parameter, so that the call does not look for a subschema copy class in the current schema with which to register this call. Call the **beginClassNotification** method if you want to look for a subschema copy class in the current schema or specify the subschema copy class itself.

includeSubclasses

Use the **includes** parameter of the **beginClassesNotification** method to specify whether subclasses are to be included in (when set to **true**) or excluded from (when set to **false**) the notification registration.

Note When registering a **beginClassesNotification** with the **includeSubclasses** parameter set to **true**, the only subclasses that are included are those that exist at that time. If the user creates (subclasses) dynamic classes after the registration, these classes do not receive notifications. This also applies if the dynamic classes exist but they are then deleted and subsequently recreated.

The **endClassesNotification** method raises an exception if a dynamic subclass has been deleted after the call to the **beginClassesNotification** method.

transients

Use the **transients** parameter of the **beginClassesNotification** method to specify if the user notification is to be invoked for events that occur to transient instances (**true**) or persistent instances (**false**) of the class.

Note You can subscribe to system notifications only for persistent objects; that is, the **transients** parameter must be **false**.

eventType

Use the **eventType** parameter of the **beginClassesNotification** method to specify the type of event for which the notification is requested. The global constants in the **SystemEvents** category for the types of event that can be subscribed to are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

responseType

Use the **responseType** parameter of the **beginClassesNotification** method to specify the frequency with which an event notification is notified. The valid values for the **responseType** parameter, represented by global constants in the **NotificationResponses** category, are listed in the following table.

Global Constant	Integer Value	Sends a notification…
Response_Cancel	1	When an instance of the class receives a matching event and then cancels the notification
Response_Continuous	0	Whenever an instance of the class receives a matching event
Response_Suspend	2	When an instance of the class receives a matching event and then suspends notification until the user refreshes the local copy of the instance of the class

eventTag

Use the **eventTag** parameter of the **beginClassesNotification** method to identify a specific notification when you have multiple subscriptions for the same event type.

beginClassesNotificationForIF

Signature

```
beginClassesNotificationForIF(theClass:          Class;
                              includeSubclasses: Boolean;
                              transients:        Boolean;
                              eventType:         Integer;
                              responseType:      Integer;
                              eventTag:          Integer;
                              theInterface:      JadeInterface);
```

The **beginClassesNotificationForIF** method of the **Object** class is a variation of the **beginClassesNotification** method.

The **beginClassesNotificationForIF** method allows notification events to be sent to methods mapped to the **userNotification** and **sysNotification** methods of the **theInterface** parameter when a nominated event occurs on instances of a class and optionally its subclasses, rather than to those of the subscriber.

The subscriber must be an instance of a **Class** that includes methods that map to (implement) the specified **sysNotification** or **userNotification** method of the interface.

The parameters specified in the method signatures must match the standard **userNotification** or **sysNotification** method, as follows.

```
sysNotification(eventType: Integer; theObject: Object; eventTag: Integer);

userNotification(eventType: Integer; theObject: Object; eventTag: Integer;
        userInfo: Any);
```

If the method signature does not match the standard **userNotification** or **sysNotification** method, an exception is raised when the **beginClassesNotificationForIF** method is executed.

The subscription to a notification registered by the **beginClassesNotificationForIF** method is terminated by the **endClassesNotificationForIF** method. You can also use the **endNotificationForSubscriber** method to terminate all notifications for a specific subscriber.

The **beginClassesNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies …
theClass	The class for which the notification is to be invoked
includeSubclasses	Whether subclasses are included in or excluded from the notification registration
transients	If the user notification is invoked for events occurring in transient instances
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value that is required for each notification
theInterface	The interface implemented by the specified class and optionally its subclasses

With the exception of the **theInterface** parameter, described in the following subsection, for details about the other parameters, see the appropriate subsections of the **beginClassesNotification** method.

theInterface

Use the **theInterface** parameter of the **beginClassesNotificationForIF** class to specify the interface that defines the appropriate **userNotification** or **sysNotification** method.

If the value of the **eventType** parameter is a system event, events are sent to the method that maps to the **sysNotification** method of the interface. If this parameter does not specify a system event, events are sent to the method that maps to the **userNotification** method of the interface.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver class and optionally its subclasses must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

beginNotification

Signature

```
beginNotification(theObj:       Object;
                  eventType:    Integer;
                  responseType: Integer;
                  eventTag:     Integer);
```

The **beginNotification** method of the **Object** class registers the receiver to be notified when a nominated event occurs on a specified object.

The object that invokes the **beginNotification** method is referred to as the *subscriber*.

An object that subscribes to a notification is notified when the specified event occurs for the notification target. If a form or control has an attached window, a requested user notification is directed to the **userNotify** method and a requested system notification to the **sysNotify** method.

Non-GUI objects respond to system and user notifications by implementing the **sysNotification** and **userNotification** methods, respectively. GUI objects (instances of a **Window** subclass) respond to system and user notifications by implementing their **sysNotify** and **userNotify** event methods, respectively. For notification on a persistent instance, you can subscribe to the following types of event:

- System events, which include object creation, deletion, or change. System events are notified automatically by the system when the nominated persistent objects are committed (using the **commitTransaction** instruction).

  Each of the target objects is automatically resynchronized in the client cache.

- User events, which are specified by selecting an **eventType** parameter in the range **User_Base_Event** through **User_Max_Event**. User events are notified when the **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method of the **Object** class or the **causeClassEvent** method of the **Class** class is invoked on a target instance.

For notifications on a transient or shared transient instance, you can subscribe to user events only.

The request for a notification registered by the **beginNotification** method is terminated by the **endNotification** method. You can also use the **endNotificationForSubscriber** method to terminate all previous notifications for a specified subscriber.

Caution If the application will be run in JADE thin client mode, subscribe to notifications and timers with care. When an event occurs, each registered subscriber is notified on the application server.

In thin client mode, the first notification or timer that is sent causes a further message to be sent to the presentation client to post a message into the Windows message queue of the client. (This is necessary so that the subsequent execution of the notification logic is synchronized with what is taking place on the presentation client.) When that Windows message is processed by the presentation client, another message is sent to the application server to initiate the processing of the first 10 queued notifications or timers for that client.

If there are more than 10 notifications, these actions are repeated until all queued events are processed. Notifications and timers could therefore have a considerable impact on network traffic.

Non-immediate events caused on transient objects are not discarded when a persistent transaction is aborted. For example, if the receiver of a **causeEvent** is a shared transient instance, any notifications are held when the transaction is aborted and delivered when the next transaction commits.

For an example of the **beginNotification** method, see "Example of Beginning Notifications", later in this topic.

The **beginNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Specifies …
theObj	The object for which the notification is to be invoked
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value used to identify a notification subscription that will be passed back to the notification method

theObj

Use the **theObj** parameter of the **beginNotification** method to specify the object for which the notification is to be invoked. If this object is transient, only user notifications can be received.

eventType

Use the **eventType** parameter of the **beginNotification** method to specify the type of event for which the notification is requested.

The global constants in the **SystemEvents** category that can be subscribed to are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

responseType

Use the **responseType** parameter of the **beginNotification** method to specify the frequency with which the subscribed event was notified.

The valid values for the **responseType** parameter, represented by global constants in the **NotificationResponses** category, are listed in the following table.

Global Constant	Integer Value	Sends a notification…
Response_Cancel	1	When the object receives a matching event and then cancels the notification
Response_Continuous	0	Whenever the object receives a matching event
Response_Suspend	2	When the object receives a matching event and then suspends notification until the user refreshes the local copy of the object

eventTag

Use the **eventTag** parameter of the **beginNotification** method to specify an integer value passed to the notification callback method when the event is notified.

This tag can be used to identify (or tag) each individual notification subscription.

Example of Beginning Notifications

The following example shows the use of the **beginClassNotification** method and the **beginNotification** method when loading a form.

```
load() updating;
vars
    a1     : A;
    b1     : B;
    c1, c2 : C;
begin
    // Creates instances of classes A, B, and C which will be the target
    // instances of the notifications.
    beginTransaction;
       create a1;
       create b1;
       create c1;
       create c2;
    commitTransaction;
    // For each of these notifications, the false parameter specifies that
    // the notification will only occur if the instance is persistent, the
    // Response_Continuous parameter specifies that a notification will be
    // sent whenever an event occurs and the final eventTag parameter is an
    // integer value that is returned with each notification.
    // Registers the receiver (in this case, the form) to be notified when a
    // system event (create, update, or delete) occurs on an instance of
    // class A.  When the notification is received, the sysNotify event of
    // the form will be executed.
    beginClassNotification(A, false, Any_System_Event, Response_Continuous,
                           1);
    // Registers the receiver to be notified when a user event with an
    // eventType of 16 (User_Base_Event) occurs on an instance of class B.
    // When the notification is received, the userNotify event of the form
    // will be executed.
    beginClassNotification(B, false, User_Base_Event, Response_Continuous,
                           2);
```

```
        // Registers the receiver to be notified when a system event occurs on
        // the instance c1 (that is, when c1 is created, updated or deleted.
        // When the notification is received, the sysNotify event of the form
        // will be executed.
        beginNotification(c1, Any_System_Event, Response_Continuous, 3);
        // Registers the receiver to be notified when a user event with an
        // eventType of 17 occurs on the instance c2.  When the notification is
        // received, the userNotify event of the form will be executed.
        beginNotification(c2, 17, Response_Continuous, 4);
    end;
```

beginNotificationForIF

Signature

```
beginNotificationForIF(theObj:       Object;
                       eventType:    Integer;
                       responseType: Integer;
                       eventTag:     Integer;
                       theInterface: JadeInterface);
```

The **beginNotificationForIF** method of the **Object** class is a variation of the **beginNotification** method. The **beginNotificationForIF** method allows notification events to be sent to the **userNotification** and **sysNotification** methods mapped to the **theInterface** parameter when a nominated event occurs on instances of an object, rather than to those of the subscriber.

The subscriber must be an instance of a **Class** that includes methods that map to (implement) the specified **sysNotification** or **userNotification** method of the interface.

The parameters specified in the method signatures must match the standard **userNotification** or **sysNotification** method, as follows.

```
sysNotification(eventType: Integer; theObject: Object; eventTag: Integer);

userNotification(eventType: Integer; theObject: Object; eventTag: Integer;
        userInfo: Any);
```

If the method signature does not match the standard **userNotification** or **sysNotification** method, an exception is raised when the **beginNotificationForIF** method is executed.

The request for a notification registered by the **beginNotificationForIF** method is terminated by the **endNotificationForIF** method. You can also use the **endNotificationForSubscriber** method to terminate all previous notifications for a specified subscriber.

The **beginNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies …
theObj	The object for which the notification is to be invoked
eventType	The type of event for which the notification is requested
responseType	The frequency with which an event notification is sent
eventTag	An integer value used to identify a notification subscription that will be passed back to the notification method
theInterface	The interface implemented by the specified object

With the exception of the **theInterface** parameter, described in the following subsection, for details about the other parameters, see the appropriate subsections of the **beginNotification** method.

theInterface

Use the **theInterface** parameter of the **beginNotificationForIF** method to specify the interface that defines the appropriate **userNotification** or **sysNotification** method.

If the value of the **eventType** parameter is a system event, events are sent to the method that maps to the **sysNotification** method of the interface. If this parameter does not specify a system event, events are sent to the method that maps to the **userNotification** method of the interface.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver class must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

beginTimer

Signature

```
beginTimer(delay:    Integer;
           option:   Integer;
           eventTag: Integer);
```

The **beginTimer** method of the **Object** class arms a timer on the receiver and registers the receiver for timer notification.

When the specified timer delay (or period) expires, the system calls the **timerEvent** method for the object that registered the notification. If a negative value for is specified for the **delay** parameter, the minimum timer granularity of one (**1**msec is used.

A specific object can register multiple timers of different durations. The **eventTag** parameter can then be used by the **timerEvent** method to determine which timer has expired.

The parameters for the **beginTimer** method are listed in the following table.

Parameter	Description
delay	Integer value (in milliseconds) for the timer delay
option	**TimerDurations** category global constant **Timer_Continuous** (the **timerEvent** occurs continuously until it is disabled by the **endTimer** method) or **Timer_OneShot** (the **timerEvent** occurs once only)
eventTag	User-specified literal or constant that can be used to identify a specific timer event

Notes In JADE thin client mode, use of timers whose logic interacts with the presentation client side of the thin client processing may cause a processing loop if the interval between timer calls is less than the time taken to process each request. This could arise over a slower-speed line where the transmission time to the presentation client becomes significant.

When you develop an application that could run in JADE thin client mode, use timers with care. When a timer event occurs, it notifies the application server, which then echoes the event to all attached presentation clients; that is, the application server sends the notification to each presentation client, which then sends a response to the application server. This can have a considerable impact on network traffic.

Timers are deactivated when the process that armed them terminates. Similarly, notifications are unsubscribed when the process that subscribed to them terminates. As timer events are not transported between nodes, a timer armed in a server method will not invoke the **timerEvent** callback on the client node.

The following example shows the activation of a timer.

```
optionActivateTimer_click(menuItem: MenuItem input) updating;
begin
    if self.timeInterval <> 0 then
        if optionActivateTimer.caption = "Deactivate Timer" then
            self.endTimer(0);
            optionActivateTimer.caption := "Activate Timer";
        else
            beginTimer(self.timeInterval * 1000, Timer_Continuous, 0);
            optionActivateTimer.caption := "Deactivate Timer";
        endif;
    endif;
end;
```

The following code fragment checks if the timer is active, and if so, stops the timer and restarts it with a new value.

```
if optionActivateTimer.caption = "Deactivate Timer" then
    self.endTimer(0);
    optionActivateTimer.caption := "Activate Timer";
    if self.timeInterval <> 0 then
        beginTimer(self.timeInterval * 1000, Timer_Continuous, 0);
        optionActivateTimer.caption := "Deactivate Timer";
    endif;
endif;
```

Use the **getTimerStatus** method to return the status of the timer specified in the **eventTag** parameter.

beginTimerForIF

Signature

```
beginTimerForIF(delay:        Integer;
                option:       Integer;
                eventTag:     Integer;
                theInterface: JadeInterface);
```

The **beginTimerForIF** method of the **Object** class is a variation of the **beginTimer** method.

The **beginTimerForIF** method arms a timer on the receiver and registers the receiver for a timer notification.

When the specified timer delay (or period) expires, the system calls the **timerEvent** method mapped to the interface that registered the notification, rather than to that of the subscriber.

The object registering the notification (that is, the subscriber) must be an instance of a **Class** that includes a method that maps to the specified **timerEvent** method of the **theInterface** parameter value. The parameter specified in the method signatures must match the standard **timerEvent** method, as follows.

```
timerEvent(eventTag: Integer);
```

The parameters for the **beginTimerForIF** method are listed in the following table.

Parameter	Description
delay	Integer value (in milliseconds) for the timer delay
option	**TimerDurations** category global constant **Timer_Continuous** (the **timerEvent** occurs continuously until it is disabled by the **endTimerForIF** method) or **Timer_OneShot** (the **timerEvent** occurs once only)

Parameter	Description
eventTag	User-specified literal or constant that can be used to identify a specific timer event
theInterface	The interface implemented by the subscriber

Use the **getTimerStatusForIF** method to return the status of the interface timer specified in the **eventTag** parameter.

The interface must have a defined **timerEvent** method and the receiver class must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

causeEvent

Signature

```
causeEvent(eventType: Integer;
           immediate: Boolean;
           userInfo:  Any);
```

The **causeEvent** method of the **Object** class triggers a user event. Any objects that have registered a **beginNotification** for that object or its class receive a corresponding event message.

A process that uses the **causeEvent** method to cause notifications for transient instances will cause system event and user event notifications *only* for shared transient instances and for non-shared transient instances that it has created (that is, the process will not cause notifications for non-shared transient instances that have been created by other processes).

The parameters for the **causeEvent** method are listed in the following table.

Parameter	Description
eventType	Integer in the range **User_Base_Event** through **User_Max_Event** that represents the event being caused.
immediate	Boolean value specifying the timing of the event; **false** indicates that notifications occur at the end of transaction and **true** indicates that the notification is sent immediately. If the client is not within a begin/commit transaction cycle and this parameter is set to **false**, the notification waits for the next commit on that client.
userInfo	A value of any primitive type value (for example, a **String** or an **Integer**) or object reference that is passed to the **userNotify** event handlers when the event is notified. Although you should not use a transient object reference across nodes, you can use a shared transient object reference between applications on the same node. Notifications containing binary and string (**Binary**, **String**, **StringUtf8**) data of up to 48K bytes can be sent across the network. For applications running within the server node, the limit for notifications containing binary or string data is 2G bytes. Note, however, that this applies only to single user and server applications. In multiuser applications, persistent notifications are sent via the database server, even if the receiving process is on the same node as the sender. In notification cause events, exception 1267 (*Notification info object too big*) is raised if the binary or string **userInfo** data exceeds the applicable value.

The following table lists the **UserEvents** category global constants for notification events.

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

You can define your own constants to represent event types in the range **User_Base_Event** through **Max_Integer**.

In a Synchronized Database Environment (SDE), when the **AuditCauseEvents** parameter in the [SyncDbService] section of the JADE initialization file is set to **true**, events caused on a primary database using **Object** class :**causeEvent** method with a persistent target and the **immediate** parameter value of **false** outside of a database transaction are not audited for replay on secondary databases because the events are not part of a transaction.

changeObjectVolatility

Signature

```
changeObjectVolatility(object:      Object;
                       volatility:  Integer;
                       conditional: Boolean);
```

The **changeObjectVolatility** method of the **Object** class changes the volatility state of the persistent object specified in the **object** parameter. (You can change the volatility state only of persistent objects. All transient objects are considered volatile.) For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

Use the **volatility** parameter to specify the volatility state that you require, represented by one of the following global constants in the **ObjectVolatility** category.

Global Constant	Integer Value	Description
Volatility_Frozen	#04	Object is frozen (that is, it is not updated)
Volatility_Stable	#08	Object is stable (that is, it is updated infrequently)
Volatility_Volatile	#00	Object is volatile (that is, it is updated often)

As object volatility state is conditional by default, a frozen object can be updated only by first changing its volatility to **Volatility_Stable** or **Volatility_Volatile**.

Use the **conditional** parameter to specify whether the change is conditional or unconditional. Set the value of this parameter to:

- **false** if the change is unconditional; that is, the change takes place even if an attempt is made to change the volatility of a frozen object that is being used by any other process.
- **true** if the change is conditional; that is, the change takes place only if the object is not in use by another process. In a multiuser system where production mode is set, it is not possible to determine whether an object is in use by another process. In that case, an exception (*1068 - Feature not available in this release*) is raised.

See the **makeObjectStable** or **makeObjectVolatile** method for an equivalent way to conditionally change the volatility of an object.

class

Signature `class(): Class;`

The **class** method of the **Object** class returns a reference to the class of the receiver object.

Note If you want to return a reference to the class of a specified object identifier (oid) even if this object is no longer valid, call the **Object** class **getClassForObject** method; for example, in exception handlers that may need to deal with object references that are no longer valid.

cloneSelf

Signature `cloneSelf(bTransient: Boolean): SelfType;`

The **cloneSelf** method of the **Object** class creates a new instance of the same type as the receiver and copies the attributes of the receiver (including the contents of primitive arrays).

This method does not invoke constructors.

Note References and **MemoryAddress** attributes are not copied and are initialized to **null** in the cloned object.

See also the **Object** class **copySelf**, and **copySelfAs** methods, which invoke constructors, and the **cloneSelfAs** method.

cloneSelfAs

Signature
```
cloneSelfAs(asClass:    Class;
            bTransient: Boolean): Object;
```

The **cloneSelfAs** method of the **Object** class creates a new instance of the class specified in the **asClass** parameter and copies any attributes of the receiver (including the contents of primitive arrays) that are common to both the receiver and target class definitions; that is, those attributes defined in a "common ancestor" class. This method does not invoke constructors.

Note References and **MemoryAddress** attributes are not copied and are initialized to **null** in the cloned object.

See also the **Object** class **copySelf** and **copySelfAs** methods, which invoke constructors, and the **cloneSelf** method.

copySelf

Signature `copySelf(transient: Boolean): SelfType;`

The **copySelf** method of the **Object** class creates a new instance of the same type as the receiver, invoking constructor methods if defined, and copies the attributes of the receiver (including the contents of primitive arrays).

Note References and **MemoryAddress** attributes are not copied and are initialized to **null** in the copied object.

See also the **Object** class **copySelfAs** method and the **cloneSelf** and **cloneSelfAs** methods, which do not invoke constructors.

copySelfAs

Signature `copySelfAs(asClass: Class; transient: Boolean): Object;`

The **copySelfAs** method of the **Object** class creates a new instance of the class specified in the **asClass** parameter, invoking constructor methods if defined, and copies any attributes of the receiver (including the contents of primitive arrays) that are common to both the receiver and target class definitions; that is, those attributes defined in a "common ancestor" class.

Note References and **MemoryAddress** attributes are not copied and are initialized to **null** in the copied object.

See also the **Object** class **copySelf** method and the **cloneSelf** and **cloneSelfAs** methods, which do not invoke constructors.

creationTime

Signature `creationTime(): TimeStamp;`

The **creationTime** method of the **Object** class returns the date and time at which the receiver was created as a timestamp. The object creation time is stored as a Coordinated Universal Time (UTC) value.

When it is accessed it is converted to a local time using the current time zone bias of the executing process.

Note The time will be out by one hour if daylight saving (DST) is in force and the object was created during a standard time (STD), or the reverse.

deletePropertyValue

Signature `deletePropertyValue(name: String): Any allowBrowserExecution;`

The **deletePropertyValue** method of the **Object** class sets the value of the property specified by the **name** parameter to null if the property is a static property.

If the property is a dynamic property, the value of the specified dynamic property is removed from the cluster, making the property un-instantiated for that object instance.

If the property name is invalid, an exception is raised.

display

Signature `display(): String;`

The **display** method of the **Object** class returns a string containing a description of the receiver. In this default implementation of **display**, the description consists of the name of the class of the receiver.

A subclass can reimplement the **display** method to provide a more-informative description appropriate to that class.

Note Most subclasses of the **Object** class in **RootSchema** reimplement the **display** method to return a description of that class of object.

edition

Signature `edition(): Integer;`

The **edition** method of the **Object** class returns the edition of the receiver as an integer value. Each object has an edition of **1** when it is created.

The edition is incremented each time that the object is updated in a specified transaction but only once for each transaction, and is used for buffer (cache) synchronization.

For more details about object editions, see "Using Object Editions" under "Unlocking Objects", in Chapter 6 of the *JADE Developer's Reference*.

endClassNotification

Signature
```
endClassNotification(theClass:   Class;
                     transients: Boolean;
                     eventType:  Integer);
```

The **endClassNotification** method of the **Object** class terminates a subscription registered by using the **beginClassNotification** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginClassNotification** method.

The **endClassNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Specifies ...
theClass	The class for which the end notification is to be invoked
transients	If the user end notification is invoked for events occurring in transient instances
eventType	The type of event for which the end notification is requested

theClass

Use the **theClass** parameter of the **endClassNotification** method to specify the class for which class notifications of the specified type are to be ended.

transients

Use the **transients** parameter of the **endClassNotification** method to specify if the user end notification is to be invoked for events that occur to transient instances (**true**) or persistent instances (**false**) of the class.

Note System notifications apply only to persistent objects.

eventType

Use the **eventType** parameter of the **endClassNotification** method to specify the type of event for which the notification subscription is to be terminated.

The valid values for system event types, represented by global constants in the **SystemEvents** category, are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

User events are in the range **User_Base_Event** through **User_Max_Event**.

endClassNotificationForIF

Signature

```
endClassNotificationForIF(theClass:     Class;
                          transients:   Boolean;
                          eventType:    Integer;
                          theInterface: JadeInterface);
```

The **endClassNotificationForIF** method of the **Object** class is a variation of the **endClassNotification** method.

The **endClassNotificationForIF** method terminates a notification to an interface method implemented by the specified class and subclasses that was previously registered by using the **beginClassNotificationForIF** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginClassNotificationForIF** method.

The **endClassNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies ...
theClass	The class for which the end notification is to be invoked
transients	If the user end notification is invoked for events occurring in transient instances
eventType	The type of event for which the end notification is requested
theInterface	The interface implemented by the subscriber class and subclasses

With the exception of the **theInterface** parameter, details about the other parameters can be found in the appropriate subsections of the **endClassNotification** method.

theInterface

Use the **theInterface** parameter of the **endClassNotificationForIF** method to specify the interface implemented by the class specified in the **theClass** parameter and its subclasses.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver class must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

endClassesNotification

Signature
```
endClassesNotification(theClass:          Class;
                       includeSubclasses: Boolean;
                       transients:        Boolean;
                       eventType:         Integer);
```

The **endClassesNotification** method of the **Object** class terminates a subscription registered by using the **beginClassesNotification** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginClassesNotification** method.

The **endClassesNotification** method parameters, described in the following subsections, are listed in the following table.

Parameter	Specifies …
theClass	The class for which the end notification is to be invoked
includeSubclasses	Whether subclasses are included in or excluded from the notification termination
transients	If the user end notification is invoked for events occurring in transient instances
eventType	The type of event for which the end notification is requested

theClass

Use the **theClass** parameter of the **endClassesNotification** method to specify the class for which class notifications of the specified type are to be ended.

Note This method does not attempt to re-interpret the value of the **theClass** parameter, so that the call does not look for a subschema copy class in the current schema whose notification subscription it is to terminate.

includeSubclasses

Use the **includeSubclasses** parameter of the **endClassesNotification** method to specify whether subclasses are to be included in (when set to **true**) or excluded from (when set to **false**) the termination of the notification subscription.

Note The **endClassesNotification** method raises an exception if a dynamic subclass has been deleted after the call to the **beginClassesNotification** method when the **includeSubclasses** parameter is set to **true**.

transients

Use the **transients** parameter of the **endClassesNotification** method to specify if the user end notification is to be invoked for events that occur to transient instances (**true**) or persistent instances (**false**) of the class.

Note System notifications apply only to persistent objects.

eventType

Use the **eventType** parameter of the **endClassesNotification** method to specify the type of event for which the notification subscription is to be terminated.

The valid values for system event types, represented by global constants in the **SystemEvents** category, are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

User events are in the range **User_Base_Event** through **User_Max_Event**.

endClassesNotificationForIF

Signature

```
endClassesNotificationForIF(theClass:          Class;
                            includeSubclasses: Boolean;
                            transients:        Boolean;
                            eventType:         Integer;
                            theInterface:      JadeInterface);
```

The **endClassesNotificationForIF** method of the **Object** class is a variation of the **endClassesNotification** method.

The **endClassesNotificationForIF** method terminates a notification to an interface method implemented by the specified class and optionally subclasses that was previously registered by using the **beginClassesNotificationForIF** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginClassesNotificationForIF** method.

The **endClassesNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies …
theClass	The class for which the end notification is to be invoked
includeSubclasses	Whether subclasses are included in or excluded from the notification termination
transients	If the user end notification is invoked for events occurring in transient instances
eventType	The type of event for which the end notification is requested
theInterface	The interface implemented by the subscriber class and optionally subclasses

With the exception of the **theInterface** parameter, details about the other parameters can be found in the appropriate subsections of the **endClassesNotification** method.

theInterface

Use the **theInterface** parameter of the **endClassesNotificationForIF** method to specify the interface implemented by the class specified in the **theClass** parameter and optionally its subclasses.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver class must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

endNotification

Signature `endNotification(theObj: Object; eventType: Integer);`

The **endNotification** method of the **Object** class terminates a previous **beginNotification** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginNotification** method.

The **endNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Specifies ...
theObj	The notification target object for which the notifications are to be ended
eventType	The type of event for which the notifications are to be ended

theObj

Use the **theObj** parameter of the **endNotification** method to specify the object for which the end notification subscription is to be terminated.

eventType

The **theObj** and the **eventType** parameters of the **endNotification** method together identify the specific notification that is to be terminated.

The global constants for the system event types, represented by global constants in the **SystemEvents** category, are listed in the following table.

Global Constant	Integer Value	Description
Any_System_Event	0	Object has been created, deleted, or updated
Object_Create_Event	4	Object has been created
Object_Delete_Event	6	Object has been deleted
Object_Update_Event	3	Object has been updated

endNotificationForIF

Signature `endNotificationForIF(theObj: Object; eventType: Integer theInterface: JadeInterface);`

The **endNotificationForIF** method of the **Object** class is a variation of the **endNotification** method. The **endNotificationForIF** method terminates a notification to an interface method implemented by the specified object instance previously registered by using the **beginNotificationForIF** method for the corresponding parameters.

Note The **eventType** parameters must be the same as the **eventType** parameters specified in the **beginNotificationForIF** method.

The **endNotificationForIF** method parameters are summarized in the following table.

Parameter	Specifies …
theObj	The notification target object for which the notifications are to be ended
eventType	The type of event for which the notifications are to be ended
theInterface	The interface implemented by the subscriber object instance

With the exception of the **theInterface** parameter, details about the other parameters can be found in the appropriate subsections of the **endNotification** method.

theInterface

Use the **theInterface** parameter of the **endNotificationForIF** method to specify the interface implemented by the object instance specified in the **theObj** parameter.

The interface must have a defined **userNotification** or **sysNotification** method and the receiver object must implement the corresponding method in the interface. (For details, see "Implementing an Interface", in Chapter 14 of the *JADE Development Environment User's Guide*.)

endNotificationForSubscriber

Signature `endNotificationForSubscriber(subscriber: Object);`

The **endNotificationForSubscriber** method of the **Object** class terminates all previous:

- **beginNotification**, **beginClassNotification**, and **beginClassesNotification** requests for the specified subscriber
- **beginNotificationForIF**, **beginClassNotificationForIF**, and **beginClassesNotificationForIF** requests for the specified subscriber

The **subscriber** parameter specifies the subscriber whose registered notifications or implemented interface notifications are to be terminated.

The following example (from the Erewhon Investments example schema supplied on the JADE release medium) shows the use of the **endNotificationForSubscriber** method when clearing the contents of a list box and disabling notifications for the objects that were in the list.

```
clear() updating;
begin
    // End all notifications
    endNotificationForSubscriber(self);
    // The list box is being cleared so clear our selected objects list
    zSelectedObjects.clear;
    if showUpdates and zCollectionOid <> null then
        // Turn notifications back on for the collection itself
        beginNotification(zCollectionOid.asOid.Collection, Any_System_Event,
                          Response_Continuous, NotifyCollectionUpdate);
    endif;
    inheritMethod;
end;
```

endTimer

Signature `endTimer(eventTag: Integer);`

The **endTimer** method of the **Object** class terminates a timer that was started and registered by using the **beginTimer** method for the corresponding parameter. Use the **eventTag** parameter to distinguish between a number of different timers. The following example shows the use of the **endTimer** method.

```
dblClick() updating;
begin
    if self.timerInProgress then
        self.endTimer(0);
    else
        self.beginTimer(2000, 0, 0);
    endif;
    self.timerInProgress := not self.timerInProgress;
end;
```

endTimerForIF

Signature
```
endTimerForIF(eventTag:     Integer;
              theInterface: JadeInterface);
```

The **endTimerForIF** method of the **Object** class is a variation of the **endTimer** method.

The **endTimerForIF** method terminates a timer that was started and registered by using the **beginTimerForIF** method for the corresponding parameters.

Use the **eventTag** parameter to distinguish between a number of different timers and the **theInterface** parameter to specify the interface in which the **timerEvent** method implemented in the subscriber is defined.

exclusiveLock

Signature `exclusiveLock(lockTarget: Object);`

The **exclusiveLock** method of the **Object** class attempts to acquire an exclusive lock on the object specified in the **lockTarget** parameter. If another process has a conflicting lock, the process waits until the lock is released. The object is exclusively locked for the duration of the transaction.

For details about exclusive locks, see "Locking Objects", in Chapter 6 of the *JADE Developer's Reference*.

getClassForObject

Signature `getClassForObject(obj: Object): Class;`

The **getClassForObject** method of the **Object** class returns a reference to the class of the object identifier (oid) specified in the **obj** parameter, even if this object is no longer valid.

If the object specified in the **obj** parameter is valid, the **getClassForObject** method returns the same reference as a call to the **Object** class **class** method (that is, **obj.class**).

Note This method is useful in exception handlers that may need to deal with object references that are no longer valid.

getClassNumberForObject

Signature `getClassNumberForObject(obj: Object): Integer;`

The **getClassNumberForObject** method of the **Object** class returns the number of the class specified in the obj parameter, even if this object is no longer valid.

getInstanceIdForObject

Signature `getInstanceIdForObject(obj: Object): Decimal;`

The **getInstanceIdForObject** method of the **Object** class returns the instance identifier of the object specified in the **obj** parameter, even if this object is no longer valid.

Note The decimal value returned by this method avoids problems caused by negative numbers for large values.

getInstanceIdForObject64

Signature `getInstanceIdForObject64(obj: Object): Integer64;`

The **getInstanceIdForObject64** method of the **Object** class returns the instance identifier of the object specified in the **obj** parameter, even if this object is no longer valid.

Note If the instance identifier is larger than $2^{\wedge 63}$, the **getInstanceIdForObject** method, which returns a decimal, must be used.

getLockCallStack

Signature
```
getLockCallStack(target:    Object;
                 callStack: ProcessStackArray input): Boolean;
```

The **getLockCallStack** method of the **Object** class returns the lock call stack for a locked object.

The **target** parameter specifies a currently locked object. The **callStack** parameter is populated with the call stack of the process at the time the object was locked. If the object is not currently locked, an empty call stack is returned.

Notes This method returns a lock call stack only when JADE has been set to record the current call stack for the process that locked the object.

To dynamically set whether JADE records the lock call stack when a process locks an object, use the **Process** class **setSaveLockCallStack** method.

To enable automatic recording of the lock call stack for all client processes, use the **DefaultProcessSaveLockCallStack** parameter in the [JadeClient] section of the JADE initialization file. To enable automatic recording of the lock call stack for all database server processes, use the **DefaultProcessSaveLockCallStack** parameter in the [JadeServer] section of the JADE initialization file.

getLockStatus

Signature
```
getLockStatus(target:       Object;
              lockType:     Integer output;
              lockDuration: Integer output;
              lockedBy:     Process output);
```

The **getLockStatus** method of the **Object** class returns the lock type and the lock duration of the current process locks for the object specified in the **target** parameter. The **lockedBy** parameter contains the current process.

This method returns only the lock status of an object locked by the current process.

getModifiedBy

Signature
```
getModifiedBy(): String;
```

The **getModifiedBy** method of the **Object** class returns a string containing the user name of the user who modified the receiver.

Note Not all entities have this information. Where this information is not available, a null value (**""**) is returned.

getName

Signature
```
getName(): String;
```

The **getName** method of the **Object** class returns a string containing the class of the receiver. In this default implementation of the **getName** method, the description consists of the name of the class of the receiver.

A subclass can reimplement the **getName** method to provide a more-informative description appropriate to that class.

getObjectStringForObject

Signature
```
getObjectStringForObject(obj: Object): String;
```

The **getObjectStringForObject** method of the **Object** class returns a string representing the object specified in the **obj** parameter.

This method is the inverse of the **String** primitive type **asObject** method.

The returned string consists of the oid-like string based on class numbers, followed by an optional lifetime indication.

The form of the oid-like string can be one of the following.

- *class-number.instId*
- *class-number.instId.parent-class-number*
- *class-number.instId.parent-class-number.subLevel.subId*

The optional lifetime can be **'(t)'**, to indicate a transient object, or **'(s)'**, to indicate a shared transient object. If the optional lifetime is absent, it indicates a persistent object.

The code fragments in the following examples show what is returned after each of the assignments to **o**.

```
// return persistent instance of class number 16401
o := '16401.1'.asObject;
// return '16401.1' for a persistent instance
write getObjectStringForObject(o);
// return transient instance of class number 16401
o := '16401.1 (t)'.asObject;
// return '16401.1 (t)' for a transient instance
write getObjectStringForObject(o);
// return shared transient instance of class number 16401
o := '16401.1 (s)'.asObject;
// return '16401.1 (s)' for a shared transient instance
write getObjectStringForObject(o);
```

For details about returning the object id (oid) in a string format for the specified object, see the **Object** class **getOidStringForObject** method.

getObjectVolatility

Signature `getObjectVolatility(object: Object): Integer;`

The **getObjectVolatility** method of the **Object** class returns the volatility state of the persistent object specified in the **object** parameter. For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

Use the **getObjectVolatility** method to determine the volatility state of a persistent object. (All transient objects are considered volatile.)

This method returns the volatility state of the object, represented by one of the following global constants in the **ObjectVolatility** category.

Global Constant	Integer Value	Description
Volatility_Frozen	#04	Object is frozen (that is, it is not updated)
Volatility_Stable	#08	Object is stable (that is, it is updated infrequently)
Volatility_Volatile	#00	Object is volatile (that is, it is updated often)

getOidString

Signature `getOidString(): String;`

The **getOidString** method of the **Object** class returns the object identifier (oid) of the receiver in a string format.

The formats of the object id for shared and exclusive references are listed in the following table.

Type of Reference	Format	Example
Shared	*class-id.instance-id*	"305.1208"
Exclusive	*class-id.instance-id.parent-class-id.sublevel.sub-id*	"66.101.305.2.1"

The following example shows the use of the **getOidString** method.

```
getInstanceId(): String;
vars
    oid    : String;
```

```
    instId : String;
    pos    : Integer;
begin
    oid := self.getOidString;
    pos := oid.pos(".",1) + 1;
    instId := oid.scanUntil(".", pos);
    return instId;
end;
```

Tip When you already have the object, calling **self.getOidStringForObject(self)** is significantly faster than calling the **getOidString** method.

getOidStringForObject

Signature `getOidStringForObject(obj: Object): String;`

The **getOidStringForObject** method of the **Object** class returns a string format of the object identifier (oid) specified in the **obj** parameter.

This method is the inverse of the **String** primitive type **asOid** method.

The formats of the object id for shared and exclusive references are listed in the following table.

Type of Reference	Format	Example
Shared	*class-id.instance-id*	"305.1208"
Exclusive	*class-id.instance-id.parent-class-id.sublevel.sub-id*	"66.101.305.2.1"

For details about returning a string of a specified object as an oid-like string based on class numbers and a following optional lifetime indication, see the **Object** class **getObjectStringForObject** method.

getOwnerForObject

Signature `getOwnerForObject(object: Object): Object;`

The **getOwnerForObject** method of the **Object** class returns a reference to the object that is the owner (parent) of the collection specified by the **object** parameter.

This method returns **null** if the **object** parameter is not an exclusive collection.

getPropertyValue

Signature `getPropertyValue(name: String): Any allowBrowserExecution;`

The **getPropertyValue** method of the **Object** class returns the value of the property specified in the **name** parameter if the property is a static property.

If the property is a dynamic property that has been initialized with a value, this value is returned. If it has not been initialized, the null value for the property type is returned.

If the **name** parameter does not correspond to a static or a dynamic property, an exception is raised.

The return result can be assigned to a variable of type **Any** or it can be converted to a specific primitive type or class if the type is known. If the property name is invalid, an exception is raised.

getTimerStatus

Signature
```
getTimerStatus(eventTag:      Integer;
               option:        Integer output;
               timeRemaining: Integer output): Boolean;
```

The **getTimerStatus** method of the **Object** class returns the status of a specified timer for the corresponding parameters.

The **eventTag** parameter is the user-specified literal or constant value that was passed to the **beginTimer** method to identify a specific timer event.

If the specified timer is active, this method returns **true** and updates the usage output parameter values listed in the following table.

Parameter	Description
option	The value that was specified in the **option** parameter of the **beginTimer** method (that is, the **TimerDurations** category **Timer_Continuous** or **Timer_OneShot** global constant)
timeRemaining	Number of milliseconds remaining until the specified timer expires

If the specified timer is not active, this method returns **false**. As each timer registration is unique to the process that armed the timer, the **getTimerStatus** method returns only the status of timers armed by the calling process.

Note More than one process can arm a timer on a persistent or a shared transient object with the same **eventTag** parameter value, in which case each process has its own independent timer registered on the object.

getTimerStatusForIF

Signature
```
getTimerStatusForIF(eventTag:      Integer;
                    option:        Integer output;
                    timeRemaining: Integer output;
                    interface:     JadeInterface): Boolean;
```

The **getTimerStatusForIF** method of the **Object** class is a variation of the **getTimerStatus** method for the corresponding parameters.

The **getTimerStatusForIF** method determines if a specific timer event started using the **beginTimerForIF** method is currently active.

The timer is identified by the values of the **eventTag** and **interface** parameters.

The **eventTag** parameter is the user-specified literal or constant value that was passed to the **beginTimerForIF** method to identify a specific timer event.

If the specified timer is active, this method returns **true** and updates the usage output parameter values listed in the following table.

Parameter	Description
option	The value that was specified in the **option** parameter of the **beginTimerForIF** method (that is, the **TimerDurations** category **Timer_Continuous** or **Timer_OneShot** global constant)
timeRemaining	Number of milliseconds remaining until the specified timer expires

If the specified timer is not active, this method returns **false**. As each timer registration is unique to the process that armed the timer, the **getTimerStatusForIF** method returns only the status of timers armed by the calling process.

Note More than one process can arm a timer on a persistent or a shared transient object with the same **eventTag** parameter value, in which case each process has its own independent timer registered on the object.

getUpdateTranID

Signature `getUpdateTranID(): Integer64;`

The **getUpdateTranID** method of the **Object** class returns the transaction identifier (TranID) of the transaction the created or last updated the receiver object. The update transaction identifier of an object corresponds to the value returned by **getTransactionId64** of the **Process** class for the transaction that originally created or updated the object.

The value returned by the **getUpdateTranID** method is persisted with the object.

hasMembers

Signature `hasMembers(coll: Collection): Boolean;`

The **hasMembers** condition method of the **Object** class returns **true** if the collection specified in the **coll** parameter has any members or it returns **false** if the collection is empty.

Note If the **coll** parameter specifies an exclusive collection, the method initially accesses the parent object of the collection (without locking) and if the collection has not been populated or instantiated (using the **Collection** class **instantiate** method), the method returns **false** without attempting to access or lock the collection.

inspect

Signature `inspect();`

The **inspect** method of the **Object** class opens an Inspector form for the receiver object. The Inspector form enables you to view properties of an object.

An exception is raised if this method is invoked from a server method or a non-GUI application.

inspectModal

Signature `inspectModal();`

The **inspectModal** method of the **Object** class opens a modal JADE Inspector form for the receiver object so that properties of the object can be viewed.

When additional forms are opened by double-clicking an object in a displayed Inspector form, the opened forms are displayed as a child of the initial modal Inspector form. These forms always sit on top of the initial modal form, and you can access all of the Inspector forms.

Closing the initial modal form also closes all of its inspector child forms.

An exception is raised if this method is invoked from a server method or a non-GUI application.

invokeIOMethod

Signature `invokeIOMethod(targetContext: ApplicationContext;
targetMethod: Method;
paramList: ParamListType io): Any;`

The **invokeIOMethod** method of the **Object** class sends the specified target method containing a variable list of parameters to the receiver, after switching to the specified **targetContext** execution context.

The return type of the **invokeIOMethod** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

After the method has finished, the execution context switches back to the current context. For details about using this method to call user methods from packages, see "Calling User Methods from Packages", in Chapter 8 of the *JADE Developer's Reference*.

The **targetMethod** parameter must be a valid method, which is executed when the **invokeIOMethod** method is called.

Use the **paramList** parameter to specify a variable list of parameters of any type that are passed to the method or condition specified in the **targetMethod** parameter when it is executed.

Note If the number or type of the actual parameters passed to a method by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result may occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list. However, the **Method** class **isCallCompatibleWith** method enables you to validate the number and type of parameters.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

As the application context used by **invokeIOMethod** is transient, it can switch to a context only within the same process. The mechanism is *not* designed to call a method running in another process in the node or in another node. In addition, as the context is transient, any connection between a context and a method to be invoked must be set up again if an application is stopped and then restarted.

If you want to save events to be called persistently so that methods would still be called if the application stops and restarts (for example, in a scheduler application), you would have to re-supply a context when the application restarts and events are loaded. The target method and object could be persistent but the context is not.

Although the callback mechanism is designed with packages in mind, you can also use it to allow a method to be invoked from within the same context. If the context in the **invokeIOMethod** call is **null**, the current context (that is, **appContext**) is used. This therefore enables you to invoke a specific saved method (for example, **myClass**::**myMethod**) rather than calling the **Object** class **sendMsg** method, which allows you to provide only the name of the method to which the message is sent. Within a package, the package writer may want to check that the method supplied by the user of the package is appropriate.

The **Method** class **isCallCompatibleWith** method checks that the target method supplied by the package user cannot be invoked only on the specified target object but that it has a signature that is compatible with that expected by the package. The **Method** class **isCallCompatibleWith** method has the following signature.

```
isCallCompatibleWith(targetObject: Object;
                     exampleMethod: Method): Boolean;
```

The method in the following example shows an example of the **invokeIOMethod** when the timer fires and inspects all events at the start of the queue and calls all those whose time has passed.

```
causeDueEvents();
vars
    se : ScheduledEvent;
begin
    foreach se in allScheduledEvents do
        if se.whenToStart > app.actualTime.time then
            return;
        endif;
        // Call users method, supplying expected start time as a parameter
        if se.targetObject <> null and se.targetMethod <> null then
            se.targetObject.invokeMethod(se.targetContext, se.targetMethod,
                                         se.whenToStart);
            se.myScheduler := null;
            delete se;
        endif;
    endforeach;
end;
```

invokeMethod

Signature

```
invokeMethod(targetContext: ApplicationContext;
             targetMethod:  Method;
             paramList:     ParamListType): Any;
```

The **invokeMethod** method of the **Object** class sends the specified target method containing a variable list of parameters to the receiver, after switching to the specified **targetContext** execution context.

The return type of the **invokeMethod** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

After the method has finished, the execution context switches back to the current context. For details about using this method to call user methods from packages, see "Calling User Methods from Packages", in Chapter 8 of the *JADE Developer's Reference*.

The **targetMethod** parameter must be a valid method, which is executed when the **invokeMethod** method is called. Use the **paramList** parameter to specify a variable list of parameters of any type that are passed to the method or condition specified in the **targetMethod** parameter when it is executed.

Notes If the number or type of the actual parameters passed to a method by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result can occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list. However, the **Method** class **isCallCompatibleWith** method enables you to validate the number and type of parameters.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

Use the **invokeIOMethod** method if the method represented by the **targetMethod** parameter takes **io** or **output** parameters.

As the application context used by **invokeMethod** is transient, it can switch to a context only within the same process. The mechanism is *not* designed to call a method running in another process in the node or in another node. In addition, as the context is transient, any connection between a context and a method to be invoked must be set up again if an application is stopped and then restarted.

If you want to save events to be called persistently so that methods would still be called if the application stops and restarts (for example, in a scheduler application), you would have to re-supply a context when the application restarts and events are loaded. The target method and object could be persistent but the context is not.

Although the callback mechanism is designed with packages in mind, you can also use it to allow a method to be invoked from within the same context. If the context in the **invokeMethod** call is **null**, the current context (that is, **appContext**) is used. This therefore enables you to invoke a specific saved method (for example, **myClass**::**myMethod**) rather than calling the **Object** class **sendMsg** method, which allows you to provide only the name of the method to which the message is sent.

Within a package, the package writer may want to check that the method supplied by the user of the package is appropriate.

The **Method**::**isCallCompatibleWith** method checks that the target method supplied by the package user cannot be invoked only on the specified target object but that it has a signature that is compatible with that expected by the package. The **Method** class **isCallCompatibleWith** method has the following signature.

```
isCallCompatibleWith(targetObject:  Object;
                     exampleMethod: Method): Boolean;
```

The method in the following example shows an example of the **invokeMethod** when the timer fires and inspects all events at the start of the queue and calls all those whose time has passed.

```
causeDueEvents();
vars
    se : ScheduledEvent;
begin
    foreach se in allScheduledEvents do
        if se.whenToStart > app.actualTime.time then
            return;
        endif;
        // Call users method, supplying expected start time as a parameter
        if se.targetObject <> null and se.targetMethod <> null then
            se.targetObject.invokeMethod(se.targetContext, se.targetMethod,
                                         se.whenToStart);
            se.myScheduler := null;
            delete se;
        endif;
    endforeach;
end;
```

isImportedObject

Signature `isImportedObject(obj: Object): Boolean;`

The **isImportedObject** method of the **Object** class returns **true** if the object specified in the **obj** parameter is an instance of an imported class or it returns **false** if it is not an instance of an imported class.

isKindOf

Signature `isKindOf(classObject: Class): Boolean;`

The **isKindOf** condition method of the **Object** class returns **true** if the receiver is an instance of the class specified in the **classObject** parameter or any of its subclasses, or it returns **false** if the receiver is not an instance.

Note An error 4 (*Object not found*) is not raised if the instance id of the receiver is invalid. In earlier releases the **hasInstance** method of the **Class** class was implemented to allow the type of an invalid object to be determined. This method is now defined only for upward compatibility. Similarly, error 1090 (*Attempted access via null object reference*) is not raised if the receiver is null.

isLockedByMe

Signature `isLockedByMe(target: Object): Boolean;`

The **isLockedByMe** method of the **Object** class returns **true** if the current process has the target object locked.

isObjectFrozen

Signature `isObjectFrozen(object: Object): Boolean;`

The **isObjectFrozen** method of the **Object** class returns **true** if the volatility state of the object specified in the **object** parameter is frozen (that is, cannot be updated).

A frozen object can be updated only by first changing its volatility to **Volatility_Stable** or **Volatility_Volatile**.

isObjectNonSharedTransient

Signature `isObjectNonSharedTransient(obj: Object): Boolean;`

The **isObjectNonSharedTransient** method of the **Object** class returns **true** if the object specified in the **obj** parameter is a non-shared transient instance or it returns **false** if it is not, even if this object is no longer valid.

isObjectPersistent

Signature `isObjectPersistent(obj: Object): Boolean;`

The **isObjectPersistent** method of the **Object** class returns **true** if the object specified in the **obj** parameter is a persistent instance or it returns **false** if it is not.

isObjectSharedTransient

Signature `isObjectSharedTransient(obj: Object): Boolean;`

The **isObjectSharedTransient** method of the **Object** class returns **true** if the object specified in the **obj** parameter is a shared transient instance or it returns **false** if it is not.

isObjectStable

Signature `isObjectStable(object: Object): Boolean;`

The **isObjectStable** method of the **Object** class returns **true** if the volatility state of the object specified in the **object** parameter is stable (that is, it is not updated frequently).

isObjectTransient

Signature `isObjectTransient(obj: Object): Boolean;`

The **isObjectTransient** method of the **Object** class returns **true** if the object specified in the **obj** parameter is a shared or a non-shared transient instance, or it returns **false** if it is not.

isObjectVolatile

Signature `isObjectVolatile(object: Object): Boolean;`

The **isObjectVolatile** method of the **Object** class returns **true** if the volatility state of the object specified in the object parameter is volatile (that is, it is often updated, and is locked and unlocked in the usual way).

isSharedTransient

Signature `isSharedTransient(): Boolean;`

The **isSharedTransient** method of the **Object** class returns **true** if the receiver is a shared transient object. This method returns **false** if the receiver is a non-shared transient object or a persistent object.

isSystemObject

Signature `isSystemObject(): Boolean;`

The **isSystemObject** method of the **Object** class returns **true** if the receiver is an instance of a system class.

isTransient

Signature `isTransient(): Boolean;`

The **isTransient** method of the **Object** class returns **true** if the receiver is a transient object. This method returns **false** if the receiver is a persistent object.

jadeReportWriterCheck

Signature `jadeReportWriterCheck(userObject: Object): Boolean;`

The **jadeReportWriterCheck** method of the **Object** class returns **true** by default.

To implement instance-based security for JADE Report Writer reports, reimplement this method in the appropriate user classes to check property values of an object against the current user access, to determine if the user has visibility to that object during the query phase of the JADE Report Writer process.

If a report references an object from the selected object, this method is called for the referenced object but the result is ignored.

Use the **userObject** parameter to specify details of the current user for checking in your reimplementation of this method. This enables you to use this object instance instead of checking for a transient instance of the **app** system variable or an equivalent, which may not be the correct instance when the report query is run in a separate process or on a server node.

This method returns **true** if the user has visibility or it returns **false** if the user does not have access to the object.

For more details about implementing security in JADE Report Writer reports, see "JadeReportWriterManager Class", "JadeReportWriterSecurity Class", and the **setSecurityObject** method of the **JadeReportWriterManager** class, earlier in this chapter.

jadeReportWriterDisplay

Signature `jadeReportWriterDisplay(): String;`

The **jadeReportWriterDisplay** method of the **Object** class returns the text of the combo box entry for each object returned by the **Application** class **jadeReportWriterParamObjects** method. This method is called automatically by the JADE Report Writer Designer application for each parameter object and it returns a null value (**""**) by default.

Reimplement this method for each class for which a report parameter is defined.

latestEdition

Signature `latestEdition(): Integer;`

The **latestEdition** method of the **Object** class returns the most-recently committed edition of the receiver as an integer value. For example, if you are using edition 4 of an object and the object has been updated twice by other users or methods so that it now has an edition of **6**, edition 6 is the edition that is returned with the **latestEdition** method.

If the receiver of the **latestEdition** method is being updated by the same process, the updated edition is returned. Other processes will instead have the most-recently committed edition returned.

Example of the Use of the latestEdition Method

When the **CustomerMaint** form for a customer is opened on a client node, the logic populates the data in the form controls as required and then stores the current edition of the object for further reference so that the object is not locked the entire time the client node has the **CustomerMaint** form open.

When the user clicks the update button, the logic can then compare the latest edition on the server against the edition stored when the form was opened. If it does not match the edition at the time the **CustomerMaint** form was populated, it can warn the user or take the appropriate action specified for the application.

lock

Signature
```
lock(lockTarget:   Object;
     lockType:     Integer;
     lockDuration: Integer;
     timeout:      Integer);
```

The **lock** method of the **Object** class acquires the type of lock specified in the **lockType** parameter for the object specified in the **lockTarget** parameter.

The duration and time of the lock are specified by the **lockDuration** and **timeout** parameters, respectively. (The **timeout** parameter specifies the number of milliseconds for the timeout.)

The following table lists the lock type, lock duration, and timeout global constant values.

Global Constant	Integer Value	Category
Exclusive_Lock	3	Locks

Global Constant	Integer Value	Category
Reserve_Lock	2	Locks
Share_Lock	1	Locks
Update_Lock	4	Locks
Persistent_Duration	2	LockDurations
Session_Duration	1	LockDurations
Transaction_Duration	0	LockDurations
LockTimeout_Immediate	-1	LockTimeouts
LockTimeout_Infinite	Max_Integer (#7FFFFFFF)	LockTimeouts
LockTimeout_Process_Defined	-2 (use the process-defined default)	LockTimeouts
LockTimeout_Server_Defined	0 (use the server-defined default)	LockTimeouts

The following example shows the use of the **lock** method.

```
createMe(newname, newadd1, newadd2, newadd3: String; newcash: Decimal)
         updating;
begin
    exclusiveLock(self);
    on LockException do alreadyInUse(exception);
    lock(self, Exclusive_Lock, Transaction_Duration, 100);
    self.name     := newname;
    self.address1 := newadd1;
    self.address2 := newadd2;
    self.address3 := newadd3;
    if newcash = 0 then
        self.cash := 100000;
    else
        self.cash := newcash;
    endif;
    self.myMarket := app.myMarket;
end;
```

makeObjectFrozen

Signature `makeObjectFrozen(object: Object);`

The **makeObjectFrozen** method of the **Object** class conditionally changes the volatility state of the persistent object specified in the **object** parameter to frozen. Alternatively, you can call the **changeObjectVolatility** method to change the volatility state of an object. For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

A frozen object can be updated only by first changing its volatility to stable or to volatile (that is, by calling the **makeObjectStable** or **makeObjectVolatile** method).

makeObjectStable

Signature `makeObjectStable(object: Object);`

The **makeObjectStable** method of the **Object** class changes the volatility state of the persistent object specified in the **object** parameter to stable. Alternatively, you can call the **changeObjectVolatility** method to change the volatility state of an object. For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

A frozen object can be updated only by first changing its volatility to stable or to volatile (that is, by calling the **makeObjectStable** or **makeObjectVolatile** method).

When attempting to change the volatility of a frozen object, an exception (*1068 - Feature not available in this release*) is raised if the object is in use by another process. In a multiuser application where production mode is set, it is not possible to determine whether an object is in use by another process. In that case, the exception is always raised; that is, the **makeObjectStable** or **makeObjectVolatile** method always raises an exception in a multiuser system with production mode set.

makeObjectVolatile

Signature `makeObjectVolatile(object: Object);`

The **makeObjectVolatile** method of the **Object** class changes the volatility state of the persistent object specified in the **object** parameter to volatile. Alternatively, you can call the **changeObjectVolatility** method to change the volatility state of an object. For details, see "Cache Concurrency", in Chapter 6 of the *JADE Developer's Reference*.

A frozen object can be updated only by first changing its volatility to stable or to volatile (that is, by calling the **makeObjectStable** or **makeObjectVolatile** method).

When attempting to change the volatility of a frozen object, an exception (*1068 - Feature not available in this release*) is raised if the object is in use by another process. In a multiuser application where production mode is set, it is not possible to determine whether an object is in use by another process. In that case, the exception is always raised; that is, the **makeObjectStable** or **makeObjectVolatile** method always raises an exception in a multiuser system with production mode set.

moveToPartition

Signature `moveToPartition(destPartitionID: Integer64) updating;`

The **moveToPartition** method of the **Object** class moves the receiver and its subobjects to the partition with the partition identifier specified by the **destPartitionID** parameter. The destination partition must be present, not frozen, and version-compatible with the receiver. The subobjects moved with an object include exclusive collections, **JadeBytes** properties, blob and slob properties.

Notes This method must be executed in transaction state.

Before the move is executed, the receiver and any exclusive collection properties are exclusively locked to prevent any further changes until the transaction has committed or aborted.

Exclusive instances of **JadeBytes** are maintained in a UDR file that is peered with the data partition containing the parent object. Moving **JadeBytes** objects with the **singleFile** property set to **true** requires a file system rename operation if the partitions are located on the same physical device, otherwise the single file instance is moved to the directory containing the destination partition.

An exception is raised if the specified partition identifier is out of range or the destination partition is frozen or offline.

reserveLock

Signature `reserveLock(lockTarget: Object);`

The **reserveLock** method of the **Object** class attempts to acquire a reserve lock on the object specified in the **lockTarget** parameter.

If the object already has a reserve lock or exclusive lock, the process waits until the lock is released for default timeout. (For details, see "ServerTimeout" under "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*.)

A reserve lock enables you to lock an object that you intend to update, when you want to minimize the time that it is locked with an exclusive lock. For more details, see "Locking Objects", in Chapter 6 of the *JADE Developer's Reference*.

respondsTo

Signature `respondsTo(jadeInterface: JadeInterface): Boolean;`

The **respondsTo** method of the **Object** class returns **true** if the receiver's class or its superclasses implement the JADE interface specified in the **jadeInterface** parameter.

resynch

Signature `resynch();`

The **resynch** method of the **Object** class marks the receiver objects as obsolete. This causes the latest edition of the object to be fetched from the server the next time the object is required.

For details about resynchronizing an object that is already in local cache, see the **resynchObject** method. For more details about object editions, see "Using Object Editions" under "Unlocking Objects", in Chapter 6 of the *JADE Developer's Reference*.

Note When automatic cache coherency is enabled (by setting the **AutomaticCacheCoherency** parameter in the [JadeClient] section of the JADE initialization file to **true**), calling the **Object** class **resynch** method has no effect.

With automatic cache coherency, an object updated on another node is automatically reloaded in cache, even when it is the receiver of a method currently being executed.

---

resynchObject

Signature `resynchObject(object: Object);`

The **resynchObject** method of the **Object** class enables you to mark as obsolete the transient replica of the object specified in the object parameter.

This causes the latest edition of the specified object to be fetched from the server the next time that object is required.

Notes It is preferable to use another object that already exists in cache to resynchronize the target object (for example, **resynchObject(myObj);** uses the current receiver of your method). When a resynchronization is performed on an object that is currently the receiver of an executing method, the operation is performed after the executing method has finished.

The object is not copied from the server if the obsolete buffer in your local cache on the client is the same edition as that on the server, but the buffer is marked as no longer obsolete when you next reference the object.

When automatic cache coherency is enabled (by setting the **AutomaticCacheCoherency** parameter in the [JadeClient] section of the JADE initialization file to **true**), calling the **Object** class **resynchObject** method has no effect. With automatic cache coherency, an object updated on another node is automatically reloaded in cache, even when it is the receiver of a method currently being executed.

See also the **Object** class **edition** and **latestEdition** methods, and "Using Object Editions" under "Unlocking Objects", in Chapter 6 of the *JADE Developer's Reference*.

sdeCauseEvent

Signature

```
sdeCauseEvent(eventType: Integer;
              immediate: Boolean;
              userInfo:  Any);
```

The **sdeCauseEvent** method of the **Object** class is used for inter-system event notification in a Synchronized Database Environment (SDE).

Calling the **sdeCauseEvent** method on a secondary database system notifies subscribers of a user event on that secondary system as well as on the primary database server.

This method combines the actions of the **Object** class **causeEvent** and **sdsCauseEvent** methods, in that subscribers are notified of user events on the local system as well as on SDS secondary or primary systems, where applicable. For example, when used by an application running in an SDS primary system, the **sdeCauseEvent** method notifies subscribers of user events on the primary database as well as on all attached secondary databases.

In contrast, the **causeEvent** method would notify subscribers of a user event only on the primary database system and the **sdsCauseEvent** method only on the secondary database systems.

The parameters for the **sdeCauseEvent** method are listed in the following table.

Parameter	Description
eventType	Integer in the range **User_Base_Event** through **User_Max_Event** that represents the event being caused.
immediate	Boolean value specifying the timing of the event; **false** indicates that notifications occur at the end of transaction and **true** indicates that the notification is sent immediately. If the client is not within a begin/commit transaction cycle and this parameter is set to **false**, the notification waits for the next commit on that client. On a primary database, subscribers are notified only on the secondaries if the target object is persistent, the value of the **immediate** parameter is **false**, and the process is currently in transaction state. On a secondary database system, subscribers are notified on the primary database system only if the target object is persistent. The value of the **immediate** parameter is immaterial. However, subscribers are always notified immediately on the primary database system, even when the value of the **immediate** parameter is **false**, to defer notification on the secondary system.

Parameter	Description
userInfo	A value of any primitive type value (for example, a **String** or an **Integer**) or persistent object reference that is passed to the **userNotification** or **userNotify** event handlers when the event is notified. (Notifications containing string and binary data of up to 48K bytes can be sent across the network.)

The following table lists the **UserEvents** category global constants for notification events.

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

You can define your own constants to represent event types in the range **User_Base_Event** through **Max_Integer**.

The actions of the **sdeCauseEvent** method are summarized in the following table, which lists the contexts in which the event is caused.

Database Role	Transient Target Object	Persistent Target Object, Immediate	Persistent Target Object, Deferred, in Transaction State	Persistent Target Object, Deferred, not in Transaction State
Undefined	Process only	Local system	Local system	Local system
Primary	Process only	Primary system	Primary system and secondary	Primary system only systems
Secondary	Process only	Primary system and secondary system	Secondary system and immediately on the primary system	Secondary system and immediately on the primary system

sdsCauseEvent

Signature

```
sdsCauseEvent(eventType: Integer;
              immediate: Boolean;
              userInfo:  Any);
```

The **sdsCauseEvent** method of the **Object** class is used for inter-system event notification in a Synchronized Database Service (SDS).

The role-dependent usage scenarios are as follows.

- From a primary system, to cause persistent events audited by the primary database for replay by secondary database servers. Calling this method on primary databases outside of transaction state raises an exception.
- From a secondary system, to cause events that are notified to event subscribers on the primary system.

The behavior of the **sdsCauseEvent** method is database role-dependent. The three database role categories are listed in the following table.

Role	Action
Primary	When invoked within an SDS primary system, the **sdsCauseEvent** method audits the event for subsequent replay by SDS secondary databases. The event is not notified on the primary. The value of the **immediate** parameter must be **false**.
Secondary	When invoked within an SDS secondary system connected to a primary database server, the **sdsCauseEvent** method triggers a corresponding event on the same receiver object in the primary system. The user event is not notified on the secondary system. Events caused on a secondary are assumed to be immediate, so the **immediate** parameter is therefore ignored.
None	When invoked within a non-SDS-capable system, the method behavior is the same as the **Object** class **causeEvent** method.

The parameters for the **sdsCauseEvent** method are listed in the following table.

Parameter	Description
eventType	Integer in the range **User_Base_Event** through **User_Max_Event** that represents the event being caused.
immediate	You must set this parameter to **false** when the method is invoked from a primary system in a Synchronized Database Environment (SDE). An exception is raised if you call this method with the immediate parameter set to **true** on an SDS primary or a non-SDS database.
userInfo	A value of any primitive type value (for example, a **String** or an **Integer**) or persistent object reference that is passed to the **userNotification** or **userNotify** event handlers when the event is notified. (Notifications containing string and binary data of up to 48K bytes can be sent across the network.)

The following table lists the **UserEvents** category global constants for notification events. (You can define your own constants to represent event types in the range **User_Base_Event** through **Max_Integer**.)

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

The actions of the **sdsCauseEvent** method are summarized in the following table, which lists the contexts in which the event is caused.

Database Role	Transient Target Object	Persistent Target Object, Immediate	Persistent Target Object, Deferred, in Transaction State	Persistent Target Object, Deferred, not in Transaction State
Undefined	Ignored	Exception	Local system	Exception
Primary	Ignored	Exception	Secondary system	Exception
Secondary	Ignored	Primary system	Immediately to primary system	Immediately to primary system

sendMsg

Signature `sendMsg(message: String): Any;`

The **sendMsg** method of the **Object** class sends the specified message (method or condition) to the receiver.

The return type of the **sendMsg** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **message** parameter must be the name of a valid method or condition, which is executed when the **sendMsg** method is called. See also the **Object** class **sendMsgWithParams** method.

The following code fragment is an example of the **sendMsg** method.

```
retCode:=self.sendMsg(meth).Integer;
```

sendMsgWithIOParams

Signature
```
sendMsgWithIOParams(msg:       String;
                    paramList: ParamListType io): Any;
```

The **sendMsgWithIOParams** method of the **Object** class sends the specified message (method or condition) containing a variable list of parameters to the receiver.

The return type of the **sendMsgWithIOParams** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **msg** parameter must be the name of a valid method or condition, which is executed when the **sendMsgWithIOParams** method is called.

Use the **paramList** parameter to specify a variable list of dictionary-key parameters of any type that are passed to the method or condition specified in the **msg** parameter when it is executed. You can:

- Call dictionary methods or conditions where the keys are not known at run time; for example, use the key access methods defined in the **DynaDictionary** class to perform ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries.
- Define your own generic methods or conditions of an abstract dictionary class.
- Define and call external methods with a variable list of parameters.

Note If the number or type of the actual parameters passed to a method or condition by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result can occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also:

- "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*
- The **Object** class **sendMsg** method

sendMsgWithParams

Signature `sendMsgWithParams(msg: String; paramList: ParamListType): Any;`

The **sendMsgWithParams** method of the **Object** class sends the specified message (method or condition) containing a variable list of parameters to the receiver.

The return type of the **sendMsgWithParams** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **msg** parameter must be the name of a valid method or condition, which is executed when the **sendMsgWithParams** method is called.

Use the **paramList** parameter to specify a variable list of dictionary-key parameters of any type that are passed to the method or condition specified in the **msg** parameter when it is executed. You can:

- Call dictionary methods or conditions where the keys are not known at run time; for example, use the key access methods defined in the **DynaDictionary** class to perform ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries.
- Define your own generic methods or conditions of an abstract dictionary class.
- Define and call external methods with a variable list of parameters.

Note If the number or type of the actual parameters passed to a method or condition by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result can occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

The following code fragment is an example of the **sendMsgWithParams** method.

```
retCode := self.sendMsgWithParams(meth, param1, param2).Integer;
```

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also:

- "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*
- The **Object** class **sendMsg** method

sendTypeMsg

Signature `sendTypeMsg(message: String): Any;`

The **sendTypeMsg** method of the **Object** class sends the specified message (type method) to the receiver.

The return type of the **sendTypeMsg** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **message** parameter must be the name of a valid type method, which is executed when the **sendTypeMsg** method is called. See also the **Object** class **sendTypeMsgWithParams** method.

sendTypeMsgWithIOParams

Signature `sendTypeMsgWithIOParams(msg: String; paramList: ParamListType io): Any;`

The **sendTypeMsgWithIOParams** method of the **Object** class sends the specified message (type method) containing a variable list of parameters to the receiver.

The return type of the **sendTypeMsgWithIOParams** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **msg** parameter must be the name of a valid type method, which is executed when the **sendTypeMsgWithIOParams** method is called.

Use the **paramList** parameter to pass one or more parameters to the method being called.

Note If the number or type of the actual parameters passed to a method or condition by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result can occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also:

- "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*
- The **Object** class **sendTypeMsg** method

sendTypeMsgWithParams

Signature `sendTypeMsgWithParams(msg: String; paramList: ParamListType): Any;`

The **sendTypeMsgWithParams** method of the **Object** class sends the specified message (type method) containing a variable list of parameters to the receiver.

The return type of the **sendTypeMsgWithParams** method is to allow for an optional return value from the method being called. If the called method returns a value, the **Any** result must be cast to the appropriate type, to access that result.

The **msg** parameter must be the name of a valid type method, which is executed when the **sendTypeMsgWithParams** method is called.

Use the **paramList** parameter to pass one or more parameters to the method being called.

Note If the number or type of the actual parameters passed to a method or condition by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result can occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also:

- "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*
- The **Object** class **sendTypeMsg** method

setPartitionID

Signature `setPartitionID(partID: Integer);`

The **setPartitionID** method of the **Object** class specifies the absolute partition in which to locate the receiver. It must be called within the creating transaction.

The value of the **partID** parameter must be a value in the range **1** through the **Max_Integer** minus **15**, the target partition must be present, not frozen, and it must be version-compatible with the source object.

The value that is set is observed only when the transaction commits; that is, only the last value that was set is used.

Notes If you are using window-relative index values, use the **Object** class **setPartitionIndex** method to specify the create partition in which to locate the receiver.

The **setPartitionID** method overwrites an existing partition index value.

Exception 3146 is raised if the specified partition id or partition index is out of range. Exception 3187 is raised if the object buffer is not being created; that is, it is already in a committed state.

setPartitionIndex

Signature `setPartitionIndex(partIndex: Integer);`

The **setPartitionIndex** method of the **Object** class specifies the partition in which to locate the receiver. It must be called within the creating transaction.

The value of the **partIndex** parameter must be a value in the range **1** through the value of the **DbFile** class **setPartitionModulus** method minus **1** and the target partition must be present, not frozen, and version-compatible with the source object.

The value set is observed only when the transaction commits; that is, only the last value set is used.

A 3146 exception is raised if the specified partition identifier is out of range.

setPropertyValue

Signature
```
setPropertyValue(name:  String;
                 value: Any) updating, allowBrowserExecution;
```

The **setPropertyValue** method of the **Object** class sets the static or dynamic property specified in the **name** parameter to the value specified in the **value** parameter.

If the property specified in the **name** parameter is invalid, an exception is raised.

Note You should not use the **setPropertyValue** method as a replacement for direct assignment to a property when the property name is known at compile time, as it incurs additional overhead and prevents the compiler from checking the type compatibility of the value being assigned.

You should use it only in special cases when property names are determined at run time.

---

sharedLock

Signature `sharedLock(lockTarget: Object);`

The **sharedLock** method of the **Object** class attempts to acquire a shared lock on the object specified in the **lockTarget** parameter.

An object that is locked by a shared lock cannot be locked with an exclusive lock or updated by other processes, but it can be locked by another shared lock or a reserve lock.

For more details, see "Locking Objects", in Chapter 6 of the *JADE Developer's Reference*.

sysNotification

Signature
```
sysNotification(eventType: Integer;
                theObject: Object;
                eventTag:  Integer) updating;
```

The **sysNotification** method of the **Object** class is called automatically by the system when a subscribed event occurs.

The **sysNotification** method is notified automatically by the system when the nominated object events are committed for persistent object events. For example, if you registered a system notification on an object, specifying a type of **Object_Update_Event**, you then receive a notification whenever that object is updated.

Use the **sysNotification** method to code actions that are required to be performed when a specified system event occurs; for example:

```
sysNotification(eventType: Integer; theObject: Object; eventTag: Integer)
                updating;
begin
    if eventType = Object_Create_Event then         // new fault
        self.createGraph;
        self.loadTable;
    endif;
end;
```

The **sysNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Contains…
eventType	The type of event received from the **causeEvent** or **causeClassEvent** method
theObject	The object for which the notification is to be received
eventTag	An integer value that is received with each notification

eventType

The **eventType** parameter of the **sysNotification** method contains the type of event that is being notified.

The types of system event that can be notified, represented by global constants in the **SystemEvents** category, are listed in the following table.

Global Constant	Integer Value	Object has been ...
Object_Create_Event	4	Created
Object_Delete_Event	6	Deleted
Object_Update_Event	3	Updated

theObject

The **theObject** parameter of the **sysNotification** method contains the target object for which the event occurred.

Attempts to access properties or methods for the object of a notification of a delete event type (**Object_Delete_Event**) raise an exception.

eventTag

The **eventTag** parameter of the **sysNotification** method contains the value used to subscribe to the notification using the **beginClassNotification**, **beginNotification**, or **beginClassesNotification** method.

timerEvent

Signature `timerEvent(eventTag: Integer) updating;`

The **timerEvent** method of the **Object** class is called by the system when the timer period (armed by using the **beginTimer** method) expires.

Use the **eventTag** parameter to identify a specific timer event when the receiver has multiple timers armed.

The **eventTag** value is registered with the **beginTimer** method.

Timers are deactivated when the process that armed them terminates. Similarly, notifications are unsubscribed when the process that subscribed to them terminates. As timer events are not transported between nodes, a timer armed in a server method will not invoke the **timerEvent** callback on the client node.

tryGetPropertyValue

Signature
```
tryGetPropertyValue(name:         String;
                    instantiated: Boolean output): Any;
```

The **tryGetPropertyValue** method of the **Object** class returns the value of the property specified in the **name** parameter if the property is a static property. For a static property, the value of the instantiated property is always **true**.

If the property is a dynamic property that has been assigned a value, that value is returned and the **instantiated** parameter is set to **true**. If it has not been assigned a value or the value has been deleted, the null value for the property type is returned and the **instantiated** parameter is set to **false**.

If the **name** parameter does not correspond to a static or a dynamic property, **false** is returned in the **instantiated** parameter.

The return result can be assigned to a variable of type **Any** or it can be converted to a specific primitive type or class if the type is known.

tryLock

Signature
```
tryLock(lockTarget:   Object;
        lockType:     Integer;
        lockDuration: Integer;
        timeout:      Integer): Boolean;
```

The **tryLock** method of the **Object** class attempts to acquire a lock of the specified type and duration, waiting up to the timeout period (in milliseconds) to obtain the lock on the object specified in the **lockTarget** parameter.

If the lock can be acquired, the method returns **true**. If the lock cannot be obtained, this method returns **false** and no lock exception is raised.

The following table lists the lock type, lock duration, and timeout system global constant values.

Global Constant	Integer Value	Category
Exclusive_Lock	3	Locks
Reserve_Lock	2	Locks
Share_Lock	1	Locks
Update_Lock	4	Locks
Persistent_Duration	2	LockDurations
Session_Duration	1	LockDurations
Transaction_Duration	0	LockDurations
LockTimeout_Immediate	-1	LockTimeouts
LockTimeout_Infinite	Max_Integer (#7FFFFFFF)	LockTimeouts
LockTimeout_Process_Defined	-2	LockTimeouts
LockTimeout_Server_Defined	0 (use the server-defined default)	LockTimeouts

The following example shows the use of the **tryLock** method.

```
lockException(lockException: LockException): Integer;
vars
    result  : Integer;
    message : String;
begin
    message := "Cannot get lock for " & lockException.lockTarget.String
               & ". It is locked by user ";
    result := app.msgBox(message & lockException.targetLockedBy.userCode &
                         ". Retry?", "Lock Error", MsgBox_Question_Mark_Icon
                         + MsgBox_Yes_No);
    if result = MsgBox_Return_Yes then
        app.mousePointer := Busy;
        while not tryLock(lockException.lockTarget, lockException.lockType,
                          lockException.lockDuration,
                          LockTimeout_Server_Defined) do
            app.mousePointer := Idle;
            result := app.msgBox(message &
                         lockException.targetLockedBy.userCode &
                         ". Retry?", "Lock Error", MsgBox_Question_Mark_Icon
```

```
                    + MsgBox_Yes_No);
            if result = MsgBox_Return_No then
                return Ex_Abort_Action;
            endif;
            app.mousePointer := Busy;
        endwhile;
        return Ex_Resume_Next;
    else
        return Ex_Abort_Action;
    endif;
epilog
    app.mousePointer := Idle;
end;
```

unlock

Signature `unlock(unlockTarget: Object);`

The **unlock** method of the **Object** class removes the current lock from the object specified in the **unlockTarget** parameter.

Objects that are manually unlocked after a **beginLoad** instruction or **beginTransaction** instruction are not unlocked until an **endLoad**, **commitTransaction**, or **abortTransaction** instruction is encountered. For more details, see "Locking Objects", in Chapter 6 of the *JADE Developer's Reference*.

updateLock

Signature `updateLock(lockTarget: Object);`

The **updateLock** method of the **Object** class attempts to acquire an update lock on the persistent object specified in the **lockTarget** parameter. An update lock can be acquired on an object whilst other processes hold shared locks on it. The purpose of the lock is to make updates to the object, although for the updates to be committed the update lock must be upgraded to an exclusive lock.

Notes Use the **useUpdateLocks** method of the **Process** class to specify whether an **update** lock or an **exclusive** lock is implicitly acquired when an object is first updated, as shown in the following code fragment.

```
process.useUpdateLocks(true);
```

The **updateLock** method can be used only within transaction state. If used outside transaction state, an exception (1026 - Not in transaction state) is raised.

For details about update locks, see "Locking Objects", in Chapter 6 of the *JADE Developer's Reference*.

updateObjectEdition

Signature `updateObjectEdition(object: Object);`

The **updateObjectEdition** method of the **Object** class increments the **edition** number of the object specified in the **object** parameter by one (**1**).

You can use the **updateObjectEdition** method in an RPS environment to perform a *null update* operation on a selected object in the primary system. Such a null update, where only the **edition** of the object changes, is audited on the primary and applied to the relational target through RPS replication.

userNotification

Signature
```
userNotification(eventType: Integer;
                 theObject: Object;
                 eventTag:  Integer;
                 userInfo:  Any) updating;
```

The **userNotification** user events are notified when the **causeClassEvent** method of the **Class** class is invoked on a target instance for the **beginNotification** method or the **causeEvent**, **sdeCauseEvent**, or **sdsCauseEvent** method of the **Object** class is invoked on a target instance for the **beginClassNotification** or **beginClassesNotification** method.

Use the **userNotification** event to code the tasks that are required to be performed when a specified event occurs.

The **userNotification** method parameters, described in the following subsections, are summarized in the following table.

Parameter	Contains…
eventType	The type of event caused by the **causeEvent**, **sdeCauseEvent**, **sdsCauseEvent**, or **causeClassEvent** method
theObject	The target object for which the event occurred
eventTag	An integer value that is called automatically by the system when a subscribed user event occurs
userInfo	A value of **Any** type that is received from the **causeEvent**, **sdeCauseEvent**, **sdsCauseEvent**, or **causeClassEvent** method

eventType

The **eventType** parameter of the **userNotification** method contains the type of event being notified and corresponds to the event type passed to the **causeEvent**, **sdeCauseEvent**, **sdsCauseEvent**, or **causeClassEvent** method that caused the event. The **UserEvents** category global constants for the types of user event that can be received are listed in the following table.

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

theObject

The **theObject** parameter of the **userNotification** method contains the target object for the notification.

eventTag

The **eventTag** parameter of the **userNotification** method identifies the notification subscription (that is, matching the **beginNotification** method for the notified event).

userInfo

The **userInfo** parameter of the **userNotification** method is a value of **Any** primitive type (that is, a **String**, **Integer**, or **Character**) that was passed in the **userInfo** parameter for the **causeEvent**, **sdeCauseEvent**, **sdsCauseEvent**, or **causeClassEvent** method.

Notifications containing binary and string (**Binary**, **String**, **StringUtf8**) data of up to 48K bytes can be sent across the network. For applications running within the server node, the limit for notifications containing binary or string data is 2G bytes. Note, however, that this applies only to single user and server applications. In multiuser applications, persistent notifications are sent via the database server, even if the receiving process is on the same node as the sender. In notification cause events, exception 1267 (*Notification info object too big*) is raised if the binary of string **userInfo** data exceeds the applicable value.

version

Signature `version(): Integer;`

The **version** method of the **Object** class returns the object version of the receiver as an integer value.

ObjectArray Class

The **ObjectArray** class is the superclass of all arrays that contain objects.

Object arrays inherit the methods defined in the **Array** class. For details about the method defined in the **ObjectArray** class, see "ObjectArray Method", in the following section.

Inherits From: Array

Inherited By: The subclasses listed in the following table

ObjectArray Subclass	Membership
ApplicationArray	Application
ClassColl	Class
ConstantColl	Constant
DbFileArray	DbFile
FileNodeArray	FileNode
JadeDbFilePartitionArray	JadeDbFilePartition
JadeDynamicObjectArray	JadeDynamicObject
JadeInterfaceColl	JadeInterface
JadePrintDataArray	JadePrintData
JadeSkinsColl	JadeSkin
JadeWebServiceUnknownHdrArray	JadeWebServiceUnknownHeader
JadeXMLAttributeArray	JadeXMLAttribute
JadeXMLElementArray	JadeXMLElement
JadeXMLNodeArray	JadeXMLNode
LockArray	Lock
NotificationArray	Notification
ProcessStackArray	MethodCallDesc
RectangleArray	Rectangle
SchemaColl	Schema
SortActorArray	SortActor
TypeColl	Type
XamlItemObjectArray	Object
XamlResultsDataGridItemArray	XamlResultsDataGridItem

ObjectArray Method

The method defined in the **ObjectArray** class is summarized in the following table.

Method	Description
addAll	Adds the contents of the collection to the object that invokes the method

addAll

Signature `addAll(coll: Collection) updating;`

The **addAll** method of the **ObjectArray** class adds the contents of the collection to the receiver.

ObjectByObjectDict Class

The **ObjectByObjectDict** collection class is an external key dictionary with a single object reference key.

Duplicate keys are disallowed.

Inherits From: ExtKeyDictionary

Inherited By: (None)

ObjectLongNameDict Class

The **ObjectLongNameDict** class is an external key dictionary with a single string key of length 304.

Duplicate keys are disallowed.

Inherits From: ExtKeyDictionary

Inherited By: (None)

ObjMethodCallDesc Class

The **ObjMethodCallDesc** class provides information at run time about a currently active method call for an object method.

For details about the property and method defined in the **ObjMethodCallDesc** class, see "ObjMethodCallDesc Property" and "ObjMethodCallDesc Method", in the following subsections. (For details about method calls made to methods defined on primitive types, see the **PrimMethodCallDesc** class.)

Inherits From: MethodCallDesc

Inherited By: (None)

ObjMethodCallDesc Property

The property defined in the **ObjMethodCallDesc** class is summarized in the following table.

Method	Contains a reference to the…
interfaceMethod	Interface method that was used

interfaceMethod

Type: JadeInterfaceMethod

The **interfaceMethod** property of the **ObjMethodCallDesc** class contains a reference to the interface method that was used. (The called method is an implementation of the interface method.)

Note This property is not yet implemented, as it is reserved for future use.

ObjMethodCallDesc Method

The method defined in the **ObjMethodCallDesc** class is summarized in the following table.

Method	Description
getReceiver	Returns the receiver object

getReceiver

Signature `getReceiver(): Object;`

The **getReceiver** method of the **ObjMethodCallDesc** class returns a reference to the method receiver.

ObjectSet Class

The **ObjectSet** class is the superclass of sets containing objects.

Inherits From: Set

Inherited By: User-defined Set classes

ODBCException Class

The **ODBCException** class is the transient-only class that defines the behavior for exceptions that occur as a result of ODBC communications.

All unexpected exceptions returned by the ODBC interface are reported with the error code of 8000. The **nativeError** and **state** properties, together with the **extendedErrorText** property of the **Exception** class, are used to describe the exception in more detail. Errors specific to the JADE query engine itself result in error codes in the range 8001 through 8256.

For details about the properties and method defined in the **ODBCException** class, see "ODBCException Properties" and "ODBCException Method", in the following subsections.

Inherits From: NormalException

Inherited By: (None)

ODBCException Properties

The properties defined in the **ODBCException** class are summarized in the following table.

Property	Description
nativeError	Contains the native data-source-specific error code
state	Five-character ODBC-defined state variable

nativeError

Type: Integer

The **nativeError** property of the **ODBCException** class contains the native error code that is specific to the data source.

For a description of the meaning of the native error, refer to your data source documentation.

state

Type: String[5]

The **state** property of the **ODBCException** class contains the five-character ODBC state code that is returned by the ODBC driver. The first two characters indicate the class of the error. The next three characters indicate the subclass of the error.

For a description of this state code, refer to your ODBC driver documentation or to the general errors provided in Microsoft ODBC documentation.

ODBCException Method

The method defined in the **ODBCException** class is summarized in the following table.

Method	Description
showDialog	Displays the ODBC exception dialog

showDialog

Signature `showDialog(): Boolean;`

The **showDialog** method of the **ODBCException** class displays the ODBC exception dialog.

If the **showDialog** method returns **true**, the action is resumed. If this method returns **false**, the action is aborted.

OleObject Class

The **OleObject** class is used to store the Object Linking and Editing (OLE) object images for the **OleControl** class. The **OleObject** class can also be used to store programmatically controlled OLE images that are not attached to a control.

Notes This class is not available on a server node.

Memory overheads are reduced by always compressing **OleObject** data when passing it to and from the application server and presentation clients. This is transparent if you use the **OleObject** class **copy**, **getData**, and **setData** methods to manipulate the binary image.

For details about the properties and methods defined in the **OleObject** class, see "OleObject Properties" and "OleObject Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

OleObject Properties

The properties defined in the **OleObject** class are summarized in the following table.

Property	Description
compressed	Specifies whether stored **OleObject** data is compressed
fullName	Contains the full name of the OLE object
oleData	Contains the OLE object data
shortName	Contains the short name of the OLE object

compressed

Type: Boolean

Availability: Read-only at any time

The **compressed** property of the **OleObject** class specifies whether stored data is compressed.

Whenever an **OleObject** object is stored, the data is compressed automatically and this property is set to **true**.

Use the **OleObject** class **copy**, **getData**, and **setData** methods to manipulate the binary image.

fullName

Type: String

Availability: Read or write at run time only

The **fullName** property of the **OleObject** class contains the full name of the OLE object. This name defaults to the OLE class or file name used to create the object.

The **fullName** property allows the object to have an identifying description assigned to the control and OLE object.

oleData

Type: OleArray

Availability: Read or write at run time only

The **oleData** property of the **OleObject** class contains a reference to the OLE object data. The **oleData** property allows the object to have programmatically controlled OLE images stored for the control.

shortName

Type: String[100]

Availability: Read or write at run time only

The **shortName** property of the **OleObject** class contains the short name of the OLE object. The short name defaults to the OLE class or short file name used to create the object.

The **shortName** property allows the object to have an identifying short description assigned to the control and OLE object.

OleObject Methods

The methods defined in the **OleObject** class are summarized in the following table.

Method	Description
copy	Copies an existing OLE object image to another instance of the **OleObject** class
getData	Returns an uncompressed **OleControl** COM object as a binary
isServerRegistered	Tests if the server that is required to run the OLE object is a registered OLE Server
setData	Stores the COM data into the object

copy

Signature `copy(obj: OleObject) updating;`

The **copy** method of the **OleObject** class copies an existing OLE object image to another instance of the **OleObject** class. The **copy** method handles any combination of transient, permanent, system-defined, and non-system-defined class objects. An exception is raised if this method is invoked from a server method.

The following example shows the copying of the contents of an OLE control (transient) to permanent storage.

```
vars
    obj : OleObject;
begin
    beginTransaction;
    create obj;
    obj.copy(self.aOleControl.oleObject);    // Copy OLE control object
                                             // to permanent object.
    commitTransaction;
    ...
end;
```

getData

Signature `getData(): Binary;`

The **getData** method of the **OleObject** class returns the uncompressed **OleControl** COM object as a binary.

isServerRegistered

Signature `isServerRegistered(): Boolean;`

The **isServerRegistered** method of the **OleObject** class tests if the server that is required to run the OLE object is a registered OLE Server on the client.

An exception is raised if this method is invoked from a server method.

setData

Signature `setData(bin: Binary) updating;`

The **setData** method of the **OleObject** class stores the COM data into the object.

To cause an **OleControl** to load this COM data, call both the **oleObject**.**setData** method followed by the **loadFromDB** methods for the **OleControl** object.

Note The **setData** method sets the OLE data only.

Use the **copy** method of the **OleObject** class to copy from one OLE object to another. The **copy** method copies the data for the object as well as the full name and short names.

The code fragment in the following example shows the use of the **setData** method.

```
foreach obj in ReviewOLEObj.instances do
    count := 1 + count;
    if count = 1 then
        oleReview2.oleObject.setData(obj.bin);
        oleReview1.loadFromDB;
    elseif count = 2 then
        oleReview2.oleObject.setData(obj.bin);
        oleReview2.loadFromDB;
    elseif count = 3 then
        oleReview2.oleObject.setData(obj.bin);
        oleReview3.loadFromDB;
    endif;
endforeach;
```

PointArray Class

The **PointArray** class is an ordered collection of **Point** values in which the values are referenced by their position in the collection.

Point arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a **Point** array.

Inherits From: Array

Inherited By: (None)

PrimMethodCallDesc Class

The **PrimMethodCallDesc** class provides information at run time about a currently active primitive method call.

For details about the property and method defined in the **PrimMethodCallDesc** class, see "PrimMethodCallDesc Property" and "PrimMethodCallDesc Method" in the following subsections. (For details about method calls made to object methods, see the **ObjMethodCallDesc** class.)

Inherits From: MethodCallDesc

Inherited By: (None)

PrimMethodCallDesc Property

The property defined in the **PrimMethodCallDesc** class is summarized in the following table.

Property	Description
primNo	Contains the number of the primitive type of the receiver

primNo

Type: Integer

Availability: Read-only

The **primNo** property of the **PrimMethodCallDesc** class contains the number of the primitive type of the receiver.

PrimMethodCallDesc Method

The method defined in the **PrimMethodCallDesc** class is summarized in the following table.

Method	Description
getReceiver	Returns the receiver object

getReceiver

Signature `getReceiver(): Object;`

The **getReceiver** method of the **PrimMethodCallDesc** class returns **null**.

Printer Class

The **Printer** class is a transient-only class that handles printing. A transient instance of the **Printer** class called **app.printer** is automatically created at run time.

Notes Client-side facilities only are available. Print facilities cannot be invoked from a server method.

You can create additional transient instances of the **Printer** class if you want to output to multiple printers or create multiple print tasks simultaneously.

If you are running JADE in thin client mode, the printing is performed on the presentation client using a printer attached to the presentation client workstation. (For details about optimizing performance when previewing print output in JADE thin client mode, see "Previewing Print Output", later in this section.)

The default printer is re-evaluated every time a new print task is initiated and JADE logic has not specifically set the printer name required. If logic sets a specific printer name (even if is the default printer), that printer continues to be used, regardless of any change to the default printer.

The following example shows the use of the **setPrinter** method to return to printing with the default printer.

```
vars
    prnt : String;
begin
    prnt := "";
    app.printer.setPrinter(prnt);
end;
```

For details about the **Printer** class constants, the properties and methods defined in the **Printer** class, defining report layouts, using the Print Progress dialog, free-format printing, and previewing print output, see the following sections.

- Defining Your JADE Report Layouts
  - Layering Print Output
- Printer Class Constants
- Printer Properties
- Printer Methods
- Using the Common Print Setup Dialog
- Using the Print Progress Dialog
- Examples of Printer Methods
- Free-Format Printing
- Previewing Print Output
  - Searching Previewed Output
- Portable Printing

Refer also to "Global Constants Reference", in Appendix A of the *JADE Encyclopaedia of Primitive Types*, for the global constants defined in the **Printer** category and to "JadePrintData Class", which is the abstract superclass of report output data classes that enable your print data to be stored or sent directly to a display device for previewing. (Use the **Printer** class **setReport** method to capture this output for storage, manipulation, and printing to meet your requirements.)

Tip When you use the **create** instruction to create an instance of a transient form class that is referenced by a local variable, a GUI form is created. If you want to create a print form at run time that simulates the entire GUI process, use the **GUIClass** class **createPrintForm** method. The **createPrintForm** method creates a form that will not create an actual GUI form and will not apply a skin (which may change the size of the client area). See also "Portable Printing", later in this section.

An exception is raised if a printing operation (for example, calling the **setPrinter** method to set the output printer) is invoked from any of the following.

- A **serverExecution** method.
- A server application running under the **jadrap.exe** JADE Remote Node Access utility (because printing requires the **jade.exe** program).

Inherits From: Object

Inherited By: (None)

Defining Your JADE Report Layouts

Define your report layouts in terms of frames on a standard JADE form. When you create a form and you specify that the form is of type **Printer**, the JADE Painter uses the appropriate default properties for printing controls.

Each report frame represents a logical grouping of data to ensure that it is printed together; for example, a header, a footer, a line on an invoice, and an outstanding balance on a statement. Printed frames can contain any standard JADE control.

If you want to generate white space on a page, you can use logic like that shown in the following code fragment rather than generating a blank frame.

```
if printer.getPrintPosition + whiteSpaceSize >=
                printer.pageHeight - printer.footer.height then
    printer.newPage;
else
    printer.setPrintPosition(printer.getPrintPosition + whiteSpaceSize);
endif;
```

Use the **formatOut** property of the **TextBox** class or the **Label** class to specify the system-defined format of data in text boxes or labels. The format options listed in the following table are available.

Option	Action
=date	Prints the current date as specified in Control Panel.
=direct	Sends the text of the control formatted in the font of the control directly to the printer. This provides you with the ability to send commands to the print driver; for example, the facsimile (fax) number when printing to a fax device. See the **JadePrintDirect** class for details about the transient class that holds output directives that are sent directly to the printer.

Option	Action
=formatdate	Prints the date in the format supplied in the **formatOut** text box of the Properties dialog **Specific** sheet in JADE Painter, as shown in the following example. `=formatdate dd/MM/yyy`
=longdate	Prints the current date in the long date format.
=page	Prints the current page number.
=pagenofm	Prints the current page number of the total number of pages in the document (for example, **2 of 8**).
=shortdate	Prints the current data in the short date format.
=time	Prints the current time (in hh:mm:ss am / pm format).
=totalpages	Prints the total number of pages in the document (for example, **8**).

For details about placing print output directly on a printer page at any location on the page without using frames, see "Free-Format Printing", later in this section.

Layering Print Output

You can layer print output; for example, when printing a background picture over which is drawn the report itself. This allows multiple drawing and printing over the same area of the printer page, retaining the underlying print image, where possible.

The background area of a control is drawn during an **app.printer.print(frame)** call if it is not transparent and the following applies.

- The **backColor** property of the control is not set to **White**
- Visible sibling controls occupy the same position within their parents
- The value of the **backColor** property of any of the parents of the control (up to and including the frame in the **print** method) is not set to **White**
- One of the parents of the control is not a **Frame** control
- The **caption** property is set for the parent **Frame** control

This affects existing systems only if the following applies.

- The **draw** methods of the **Printer** class are followed by executing the **print** method over the same area. (This is intended to let the drawn image remain, where possible.)
- The **print** method of the **Printer** class is followed by executing the **setPrintPosition** method and then the **print** method over the same area. (This is intended to allow multiple output in the same space, where possible.)
- Tables with the value of the **backColor** property set to **White** show white table cells as transparent.

Printer Class Constants

The **Printer** class constants are listed in the following table.

Printer Class Constant	Value	Description
DrawFillStyle_Cross	6	Cross
DrawFillStyle_DiagonalCross	7	Diagonal cross
DrawFillStyle_DownDiagonal	5	Downward diagonal
DrawFillStyle_HorzLine	2	Horizontal line
DrawFillStyle_Solid	0	Solid (the default)
DrawFillStyle_Transparent	1	Transparent
DrawFillStyle_UpDiagonal	4	Upward diagonal
DrawFillStyle_VertLine	3	Vertical line
DrawGrid_Crosses	1	Small crosses drawn at the grid line intersection
DrawGrid_Dots	2	Dots drawn at the grid line intersections
DrawGrid_Lines	0	Horizontal and vertical grid lines
DrawStyle_Dash	1	Dash
DrawStyle_DashDot	3	Dash-dot
DrawStyle_DashDotDot	4	Dash-dot-dot
DrawStyle_Dot	2	Dot
DrawStyle_InsideSolid	6	Draws inside the bounding rectangle, taking the width of the pen into account
DrawStyle_Solid	0	Solid (the default)
DrawStyle_Transparent	5	Transparent
DrawTextAlign_Center	2	Positions text so that it is centered
DrawTextAlign_Left	0	Text is output starting at the specified left position (the default)
DrawTextAlign_Right	1	Positions text so that it ends at the specified position
Duplex_Horizontal	3	Prints on both sides of the paper to read by flipping over like a notepad (that is, the duplex **Short Side** setting)
Duplex_Simplex	1	Print is output to one side of the paper only (the default)
Duplex_Vertical	2	Prints on both sides of the paper to read by turning like a book (that is, the duplex **Long Side** setting)
PrintedStatus_Aborted	3	The report printing was aborted
PrintedStatus_All	1	The entire report was printed
PrintedStatus_Cancelled	4	The report printing process was canceled, producing only partial output
PrintedStatus_Not	0	No printing occurred
PrintedStatus_Partial	2	The user printed specific pages only in print preview

Printer Properties

The properties defined in the **Printer** class are summarized in the following table.

Property	Description
autoPaging	Specifies whether the system is to control the incrementing of the page number for each page
bottomOfPage	Contains the margin at the bottom of the printed page of output
collate	Specifies whether the print output is collated
copies	Contains the number of copies to be printed
documentType	Contains the printer form type
drawFillColor	Contains the color used to fill in shapes drawn with the printer graphics methods
drawFillStyle	Contains the pattern used to fill the shapes drawn using the printer graphics methods
drawFontBold	Used when constructing the font used for drawing text
drawFontItalic	Used when constructing the font used for drawing text.
drawFontName	Used when constructing the font used for drawing text
drawFontSize	Used when constructing the font used for drawing text
drawFontStrikethru	Used when constructing the font used for drawing text
drawFontUnderline	Used when constructing the font used for drawing text
drawStyle	Defines the line style for output from printer graphics methods
drawTextAlign	Contains the alignment used when outputting text on the printer using the **drawTextAt** and **drawTextIn** methods
drawTextCharRotation	Specifies the angle in degrees between each characters base line and the *x* axis of the device
drawTextRotation	Specifies the angle in degrees between the base line of the text output and the *x* axis of the page
drawWidth	Contains the line width for output from printer graphics methods
duplex	Contains the duplex setting for the print output
footerFrame	Contains the frame that is printed automatically at the end of each page
headerFrame	Contains the frame that is printed automatically at the beginning of each page
leftMargin	Contains the left margin of the printed page of output
orientation	Contains the orientation of your printed output
pageBorderWidth	Specifies whether a border is to be printed around the page
pageNumber	Contains the page number to be printed in a label or text box
paperSource	Contains the paper source, or tray, for the print output
printPreview	Specifies whether the printed output is to be directed to the preview file
printPreviewAllowPrint	Specifies whether previewed output can be directed to the printer

Property	Description
printPreviewAllowSelect	Specifies whether the **Print Selected** button is displayed during print preview
printPreviewReduce	Specifies whether previewed output is reduced to display a full page on the screen
retainCMDValues	Specifies whether printer values are retained when the printer is closed
rightMargin	Contains the right margin of the printed page of output
suppressDialog	Specifies whether the system-supplied print progress dialog is to be displayed
title	Contains the title to be displayed on the system-supplied print progress dialog
topOfPage	Contains the margin at the top of the printed page of output

autoPaging

Type: Boolean

The **autoPaging** property of the **Printer** class specifies whether the system controls the incrementing of the page number on the printing of each page. The default value is **true**.

bottomOfPage

Type: Integer

The **bottomOfPage** property of the **Printer** class contains the margin at the bottom of the printed page of output. Specify the required value in millimeters. This property can be modified at any time. The default value is zero (**0**).

collate

Type: Boolean

The **collate** property of the **Printer** class specifies whether the print output is collated; that is, prints the copies in proper binding order by separating copies into groups. The default value is **false**.

Notes This property cannot be modified after printing has begun.

This property applies only when the **Printer** class **copies** property is greater than **1** (the default) and the printer device supports the collation of multiple copies.

For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

copies

Type: Integer

The **copies** property of the **Printer** class contains the number of copies to be printed. The default value is **1**.

Notes This property cannot be modified after printing has begun.

Multiple copies are produced only if the printer device driver supports the printing of multiple copies.

For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

documentType

Type: Integer

The **documentType** property of the **Printer** class contains the printer form type; that is, the paper size. The default value is **Print_A4**.

You can change the **documentType** property dynamically, to allow allocation of the paper type to be used on a page-by-page basis.

Changing this property causes a **newPage** method to be executed before the **documentType** property is changed if the print is not at the start of a new page. Changing the **documentType** property also causes the **pageHeight** and **pageWidth** methods to return the appropriate values for the new paper type size.

Use the **Printer** class **getDefaultDocumentType** method to return the default document type that is set for the physical printer.

Use the **Printer** class **setCustomPaperSize** method to dynamically set a custom printer paper size at run time, using the specified width and height in units of a tenth of a millimeter (for example, call **app.printer.setCustomPaperSize(2100, 2970);** to set the paper size equivalent to **A4**). If you call **printer.documentType** to set **Print_Custom_Paper**, an exception is raised.

The **Printer** global constant category document (printer form) types are listed in the following table.

Global Constant	Integer Value	Description
Print_10X11	45	10 x 11 in
Print_10X14	16	10x14 inches
Print_11X17	17	11x17 inches
Print_15X11	46	15 x 11 in
Print_9X11	44	9 x 11 in
Print_A2	66	A2 420 x 594 mm
Print_A3	8	A3 297 x 420 mm
Print_A3_Extra	63	A3 Extra 322 x 445 mm
Print_A3_Extra_Transverse	68	A3 Extra Transverse
Print_A3_Transverse	67	A3 Transverse 297 x 420 mm
Print_A4	9	A4 210 x 297 mm
Print_A4Small	10	A4 Small 210 x 297 mm
Print_A4_Extra	53	A4 Extra 9.27 x 12.69 in
Print_A4_Plus	60	A4 Plus 210 x 330 mm
Print_A4_Transverse	55	A4 Transverse 210 x 297 mm
Print_A5	11	A5 148 x 210 mm
Print_A5_Extra	64	A5 Extra 174 x 235 mm
Print_A5_Transverse	61	A5 Transverse 148 x 210 mm
Print_A_Plus	57	SuperA - A4 227 x 356 mm

Global Constant	Integer Value	Description
Print_B4	12	B4 250 x 354 mm
Print_B5	13	B5 182 x 257 mm
Print_B5_Extra	65	B5 (ISO) Extra 201 x 276 mm
Print_B5_Transverse	62	B5 (JIS) Transverse 182 x 257 mm
Print_B_Plus	58	SuperB - A3 305 x 487 mm
Print_CSheet	24	C size sheet
Print_Custom_Paper	256	Customized paper size
Print_DSheet	25	D size sheet
Print_ESheet	26	E size sheet
Print_Env_10	20	Envelope #10 4 1/8 x 9 1/2 inches
Print_Env_11	21	Envelope #11 4 1/2 x 10 3/8 inches
Print_Env_12	22	Envelope #12 4 3/4 x 11 inches
Print_Env_14	23	Envelope #14 5 x 11 1/2 inches
Print_Env_9	19	Envelope #9 3 7/8 x 8 7/8 inches
Print_Env_B4	33	Envelope B4 250 x 353 mm
Print_Env_B5	34	Envelope B5 176 x 250 mm
Print_Env_B6	35	Envelope B6 176 x 125 mm
Print_Env_C3	29	Envelope C3 324 x 458 mm
Print_Env_C4	30	Envelope C4 229 x 324 mm
Print_Env_C5	28	Envelope C5 162 x 229 mm
Print_Env_C6	31	Envelope C6 114 x 162 mm
Print_Env_C65	32	Envelope C65 114 x 229 mm
Print_Env_DL	27	Envelope DL 110 x 220 mm
Print_Env_Invite	47	Envelope Invite 220 x 220 mm
Print_Env_Italy	36	Envelope 110 x 230 mm
Print_Env_Monarch	37	Envelope Monarch 3.875 x 7.5 inches
Print_Env_Personal	38	6 3/4 Envelope 3 5/8 x 6 1/2 inches
Print_Executive	7	Executive 7 1/4 x 10 1/2 inches
Print_Fanfold_Lgl_German	41	German Legal Fanfold 8 1/2 x 13 inches
Print_Fanfold_Std_German	40	German Std Fanfold 8 1/2 x 12 inches
Print_Fanfold_US	39	US Std Fanfold 14 7/8 x 11 inches
Print_Folio	14	Folio 8 1/2 x 13 inches
Print_ISO_B4	42	B4 (ISO) 250 x 353 mm
Print_Japanese_PostCard	43	Japanese Postcard 100 x 148 mm

Global Constant	Integer Value	Description
Print_LetterSmall	2	Letter Small 8 1/2 x 11 inches
Print_Ledger	4	Ledger 17 x 11 inches
Print_Legal	5	Legal 8 1/2 x 14 inches
Print_Legal_Extra	51	Legal Extra 9.275 x 15 in
Print_Letter	1	Letter 8 1/2 x 11 inches
Print_LetterSmall	2	Letter Small 8½ x 11 inches
Print_Letter_Extra	50	Letter Extra 9.275 x 12 in
Print_Letter_Extra_Transverse	56	Letter Extra Transverse 9.275 x 12 in
Print_Letter_Plus	59	Letter Plus 8.5 x 12.69 in
Print_Letter_Transverse	54	Letter Transverse 8.275 x 11 in
Print_Note	18	Note 8 1/2 x 11 inches
Print_Quarto	15	Quarto 215 x 275 mm
Print_Statement	6	Statement 5 1/2 x 8 1/2 inches
Print_Tabloid	3	Tabloid 11 x 17 inches
Print_Tabloid_Extra	52	Tabloid Extra 11.69 x 18 in

The code fragment in the following example shows the use of the **documentType** property.

```
// Specify the format of the pages to be printed.  As these default
// to Print_Portrait and Print_A4, you only need to redefine them
// if you require a different format.
app.printer.orientation  := app.printer.Print_Landscape;
app.printer.documentType := app.printer.Print_Letter;
```

For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

drawFillColor

Type: Integer

Availability: Read or write at run time only

The **drawFillColor** property of the **Printer** class contains the color used to fill shapes drawn with the printer graphics methods. By default, the **drawFillColor** property is set to **0** (black).

JADE uses the RGB scheme for colors. The valid range for a normal RGB color is **0** through **16,777,215** (#FFFFFF). The high byte of a number in this range equals **0**; the lower three bytes, from least to most significant byte, determine the amount of red, green, and blue, respectively. The red, green, and blue components are each represented by a number in the range **0** through **255** (#FF). If the high byte is not zero (**0**), JADE uses the system colors, defined in the Control Panel of the user.

When the **drawFillStyle** property is set to **DrawFillStyle_Transparent** (1), the setting of the **drawFillColor** property is ignored.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFillStyle

Type: Integer

Availability: Read or write at run time only

The **drawFillStyle** property of the **Printer** class contains the pattern used to fill the shapes drawn with the printer graphics methods.

When the **drawFillStyle** property is set to **DrawFillStyle_Transparent** (1), the **drawFillColor** property is ignored.

The settings of the **drawFillStyle** property are listed in the following table.

Printer Class Constant	Value	Description
DrawFillStyle_Cross	6	Cross
DrawFillStyle_DiagonalCross	7	Diagonal cross
DrawFillStyle_DownDiagonal	5	Downward diagonal
DrawFillStyle_HorzLine	2	Horizontal line
DrawFillStyle_Solid	0	Solid (the default)
DrawFillStyle_Transparent	1	Transparent
DrawFillStyle_UpDiagonal	4	Upward diagonal
DrawFillStyle_VertLine	3	Vertical line

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontBold

Type: Boolean

Availability: Read or write at run time only

The **drawFontBold** property of the **Printer** class, together with the **drawFontItalic**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontName**, **drawFontSize**, **drawTextRotation**, and **drawTextCharRotation** properties, determines the font used for printer graphics text drawing methods. The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontItalic

Type: Boolean

Availability: Read or write at run time only

The **drawFontItalic** property of the **Printer** class, together with the **drawFontBold**, **drawFontName**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontSize**, **drawTextCharRotation**, and **drawTextRotation** properties, determines the font used for printer graphics text drawing methods. The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontName

Type: String

Availability: Read or write at run time only

The **drawFontName** property of the **Printer** class, together with the **drawFontBold**, **drawFontItalic**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontSize**, **drawTextCharRotation**, and **drawTextRotation** properties, determines the font used for printer graphics text drawing methods. The font that is used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontSize

Type: Real

Availability: Read or write at run time only

The **drawFontSize** property of the **Printer** class, together with the **drawFontBold**, **drawFontName**, **drawFontStrikethru**, **drawFontUnderline**, **drawFontItalic**, **drawTextCharRotation**, and **drawTextRotation** properties, determines the font used for printer graphics text drawing methods.

The font used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontStrikethru

Type: Boolean

Availability: Read or write at run time only

The **drawFontStrikethru** property of the **Printer** class, together with the **drawFontBold**, **drawFontName**, **drawFontItalic**, **drawFontSize**, **drawFontUnderline**, **drawTextCharRotation**, and **drawTextRotation** properties, determines the font used for printer graphics text drawing methods.

The font used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFontUnderline

Type: Boolean

Availability: Read or write at run time only

The **drawFontUnderline** property of the **Printer** class, together with the **drawFontBold**, **drawFontName**, **drawFontSize**, **drawFontItalic**, **drawFontStrikethru**, **drawTextCharRotation**, and **drawTextRotation** properties, determines the font used for printer graphics text drawing methods.

The font used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawStyle

Type: Integer

Availability: Read or write at run time only

The **drawStyle** property of the **Printer** class contains the line style for output from printer graphics methods. The settings of the **drawStyle** property are listed in the following table.

Printer Class Constant	Value	Description
DrawStyle_Dash	1	Dash
DrawStyle_DashDot	3	Dash-dot
DrawStyle_DashDotDot	4	Dash-dot-dot
DrawStyle_Dot	2	Dot
DrawStyle_InsideSolid	6	Draws inside the bounding rectangle, taking the width of the pen into account
DrawStyle_Solid	0	Solid (the default)
DrawStyle_Transparent	5	Transparent

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextAlign

Type: Integer

Availability: Read or write at run time only

The **drawTextAlign** property of the **Printer** class contains the alignment used when outputting text on the printer using the **drawTextAt** and **drawTextIn** methods.

The settings of the **drawTextAlign** property are listed in the following table.

Printer Class Constant	Value	Description
DrawTextAlign_Center	2	The **drawTextAt** method positions the text so that it is centered horizontally over the specified position. The **drawTextIn** method positions the text so that it is centered within the specified rectangle.
DrawTextAlign_Left	0	Text is output starting at the specified left position (default).
DrawTextAlign_Right	1	The **drawTextAt** method positions the text so that it ends at the specified position. The **drawTextIn** method positions the text so that it ends at the right hand edge of the requested rectangle.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextCharRotation

Type: Real

Availability: Read or write at run time only

The **drawTextCharRotation** property of the **Printer** class specifies the angle, in degrees, between the base line and the *x*-axis of the device of each character. For example, a value of **90** draws the characters so that they are positioned on their side with their base parallel with the right hand edge of the page. The default value is **0** degrees.

This property, in conjunction with the **drawTextRotation** property, allows the output of non-horizontal left to right text. Use this property only with the **drawTextAt** method, as the rotated text could be rotated outside the rectangle defined by the **drawTextIn** method.

The **drawTextCharRotation** property, together with the **drawFontBold**, **drawFontStrikethru**, **drawFontItalic**, **drawFontName**, **drawFontSize**, **drawTextRotation**, and **drawFontUnderline** properties, determines the font used for printer graphics text drawing methods. The font used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextRotation

Type: Real

Availability: Read or write at run time only

The **drawTextRotation** property of the **Printer** class specifies the angle, in degrees, between the base line of the text output and the *x*-axis of the page. For example, a value of **270** draws text upright down the page. The default value is **0** degrees.

This property, in conjunction with the **drawTextCharRotation** property, allows the output of non-horizontal left to right text. Use this property only with the **drawTextAt** method, as the rotated text could be rotated outside the rectangle defined by the **drawTextIn** method.

The **drawTextRotation** property, together with the **drawFontBold**, **drawFontStrikethru**, **drawFontItalic**, **drawFontName**, **drawFontSize**, **drawTextCharRotation**, and **drawFontUnderline** properties, determines the font used for printer graphics text drawing methods. The font used defaults to the application font defined by the **fontName** property of the **Application** class.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawWidth

Type: Integer

Availability: Read or write at run time only

The **drawWidth** property of the **Printer** class contains the line width for output from printer graphics methods. To increase the width of the line, increase the value of the **drawWidth** property.

Set the **drawWidth** property to a value in the range **1** through **32,767**. This value represents the width of the line in pixels. The default value is **1** pixel wide.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

duplex

Type: Integer

The **duplex** property of the **Printer** class contains the duplex value; that is, the number of sides on which the paper is printed.

The default value is **Duplex_Simplex**; that is, printing is single-sided.

The duplex options provided by the **Printer** class are listed in the following table.

Printer Class Constant	Integer Value	Description
Duplex_Horizontal	3	Prints on both sides of the paper to read by flipping over like a notepad (that is, the duplex **Short Side** setting
Duplex_Simplex	1	Print is output to one side of the paper only
Duplex_Vertical	2	Prints on both sides of the paper to read by turning like a book (that is, the duplex **Long Side** setting)

The code fragment in the following example shows the use of the **duplex** property.

```
app.printer.duplex := app.printer.Duplex_Horizontal;
```

This property cannot be modified after printing has begun. For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

footerFrame

Type: Frame

The **footerFrame** property of the **Printer** class contains a reference to a frame that is printed automatically at the end of each page. The **footerFrame** property is set by the **setFooter** method.

To return the current value of this property, call the **Printer** class **getFooter** method.

The code fragment in the following example shows the use of the **footerFrame** property.

```
if footerFrame <> null then
    footerHgt := footerFrame.height.Integer;
endif;
```

The footer frame can be modified at any time. The default value is **null**.

headerFrame

Type: Frame

The **headerFrame** property of the **Printer** class contains a reference to a frame that is printed automatically at the beginning of each page. The **headerFrame** property is set by the **setHeader** method.

To return the current value of this property, call the **Printer** class **getHeader** method.

The code fragment in the following example shows the use of the **headerFrame** property.

```
if headerFrame <> null then
    headerHgt := headerFrame.height.Integer;
endif;
```

The header frame can be modified at any time, but the new frame value takes effect only at the start of a new page of output. The default value is **null**.

leftMargin

Type: Integer

The **leftMargin** property of the **Printer** class contains the left margin of the printed page of output. Specify the required value in millimeters; for example:

```
app.printer.leftMargin := 25;
```

The code fragment in the following example shows the use of the **leftMargin** property.

```
printForm.label6.caption := $S_Line & lineCount.String;
if not app.printer.frameFits(printForm.detail1) then
    timeTaken := (app.clock - startTime).Time;
    printForm.timeTaken1.caption := timeTaken.String;
    printForm.timeTaken2.caption := timeTaken.String;
    checkForHeaderSwap(printForm);
    startTime := app.clock;
    if cb_ChangeMargins.value then
        leftMargin.text := (leftMargin.text.Integer + 5).String;
        app.printer.leftMargin  := leftMargin.text.Integer;
        app.printer.rightMargin := rightMargin.text.Integer + 5;
    endif;
endif;
result := app.printer.print(printForm.detail1);
```

This property can be modified at any time. The default value is **10**.

orientation

Type: Integer

The **orientation** property of the **Printer** class contains the orientation of your printed output.

You can change the **orientation** property dynamically, to allow allocation of the page orientation to be used on a page-by-page basis.

Changing this property causes a **newPage** method to be executed before the **orientation** property is changed if the print is not at the start of a new page. Changing the **orientation** property also causes the **pageHeight** and **pageWidth** methods to return the appropriate values for the new page size.

Set this property to one of the global constants provided by the **Printer** category listed in the following table.

Global Constant	Integer Value	Action
Print_Landscape	2	Landscape (horizontal page orientation)
Print_Portrait	1	Portrait (vertical page orientation)

The code fragment in the following example shows the use of the **orientation** property.

```
app.printer.orientation := Print_Landscape;
```

The default value is **Print_Portrait** (portrait orientation).

For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

pageBorderWidth

Type: Integer

The **pageBorderWidth** property of the **Printer** class contains the width of a border that is to be printed around the page.

Set this property to a non-zero value (in pixels) to specify the width of the border that is to be printed; as shown in the code fragments in the following examples.

```
if pageBorder.value then
    if borderWidth.text.Integer > 0 then
        app.printer.pageBorderWidth := borderWidth.text.Integer;
    endif;
endif;

app.printer.pageBorderWidth := 2;
```

This property can be modified at any time, but the new value takes effect only at the start of the next page of output. The default value is zero (**0**), specifying that no border is to be printed.

pageNumber

Type: Integer

The **pageNumber** property of the **Printer** class contains the page number to be printed in any label or text box with the **=page** output format.

This property is incremented automatically with each new page that is printed, if the **autoPaging** property value is **true**. The user can modify this property at any time.

The default value is **1**.

paperSource

Type: Integer

The **paperSource** property of the **Printer** class contains the location in the printer of the paper tray that you want to use for your printed output so that you can use different paper for a part of a document.

You can change the **paperSource** property dynamically, to allow allocation of the paper source to be used on a page-by-page basis. Changing this property causes a **newPage** method to be issued if the print is not at the start of a new page before the **paperSource** property is changed.

As this property is not supported by all printers, the value of this property is printer driver-specific; that is, different printer models may support different paper sources. (For example, your printer driver may assign **256** to an upper tray, **257** to a lower tray, and **4** to manual feed.) The default value of zero (**0**) indicates that all paper sources are displayed in the common Print dialog.

For a printer with no paper sources defined, the **paperSource** property is ignored. This is the case for some PDF printer drivers, for example.

The code fragment in the following example shows the use of the **paperSource** property to print output to the manual feeder.

```
app.printer.paperSource := 4;
```

Use the **Printer** class **getAllPaperSources** method to access the valid paper sources of a printer and the **getDefaultPaperSource** method to return the default paper source that is set for the printer.

For details about retaining the setting of this property when the printer is closed, see the **Printer** class **retainCMDValues** property.

printPreview

Type: Boolean

The **printPreview** property of the **Printer** class specifies whether the printed output is to be directed to the preview file. This property cannot be modified after printing has begun. The default value is **false**.

The following example shows the use of the **printPreview** property.

```
buttonPreview_click(btn: Button input) updating;
vars
    report : ReportForm;
begin
    // Creates an instance of the ReportForm transient form class, and
    // references it by the report local variable.  This variable can then
    // access the controls on the form.
    create report;
    // Specifies that the output is to be directed to the preview file
    // before being printed.
    app.printer.printPreview := true;
    // Specifies the format of the pages to be printed.  As these are
    // the default values, it is unnecessary to redefine them unless
    // a different format is required.
    app.printer.orientation  := Print_Portrait;
    app.printer.documentType := Print_A4;
    // Uses the print method to output frameDetail of the form to the print
    // preview file twice.  The close method then sends all buffered
    // output to the preview file. The preview file becomes available
    // for browsing at this point.
    app.printer.print(report.frameDetail);
    app.printer.print(report.frameDetail);
    app.printer.close;
epilog
    // Deletes the transient report form instance.
    delete report;
end;
```

See also "Previewing Print Output", later in this section.

printPreviewAllowPrint

Type: Boolean

The **printPreviewAllowPrint** property of the **Printer** class specifies whether previewed output can be directed to the printer. This property can be modified at any time. The default value is **true**.

The code fragment in the following example shows the use of the **printPreviewAllowPrint** property.

```
app.printer.printPreviewAllowPrint := not disallowPrint.value;
```

See also the **printPreviewAllowSelect** property and "Previewing Print Output", later in this section.

printPreviewAllowSelect

Type: Boolean

The **printPreviewAllowSelect** property of the **Printer** class specifies whether the **Print Selected** button is displayed during print preview. This property can be modified at any time.

The default value is **true**.

By default, the **Print Report** and **Print Selected** buttons are displayed during print preview, allowing the user to print the whole report and specific pages. When the **printPreviewAllowPrint** property is set to **false**, neither button is displayed. When the **printPreviewAllowPrint** property is set to **true** and the **printPreviewAllowSelect** property is set to **false**, only the **Print Report** button is displayed. See also "Previewing Print Output", later in this section.

printPreviewReduce

Type: Boolean

The **printPreviewReduce** property of the **Printer** class specifies whether previewed output is reduced to display a full page of print output on the screen. Set this property to **false** at development time to display the output across the width of the screen (or click the **Expand** button in the Preview window at run time).

This property can be modified at any time. The default value is **true**.

The code fragment in the following example shows the use of the **printPreviewReduce** property.

```
if expand.value then
    app.printer.printPreviewReduce := false;
endif;
```

See also "Previewing Print Output", later in this section.

retainCMDValues

Type: Boolean

The **retainCMDValues** property of the **Printer** class specifies whether the following property values are retained after the printer is closed.

- **collate**
- **copies**
- **documentType**
- **duplex**

- orientation
- paperSource

By default, all property values are re-initialized to the JADE default values when the printer is closed. The default value for the **retainCMDValues** property is **false**. When you set it to **true**, the property values set when the common print dialogs are used are retained. The values are retained regardless of how they were set (that is, dynamically in your logic or by the **CMDPrint** class Print Setup dialog or Print dialog).

Note The JADE development environment sets this property **true** for the printing of any method or class so that any values that you specify in the common Print Setup dialog are retained for subsequent print requests.

rightMargin

Type: Integer

The **rightMargin** property of the **Printer** class contains the right margin of the printed page of output. This property can be modified at any time and it affects only the position of the page border.

Specify the required value in millimeters. The default value is **10**.

The code fragments in the following examples show the use of the **rightMargin** property.

```
app.printer.rightMargin := 15;

app.printer.rightMargin := rightMargin.text.Integer + 5;
```

See also the **pageBorderWidth** property.

suppressDialog

Type: Boolean

The **suppressDialog** property of the **Printer** class specifies whether the system-supplied print progress dialog is to be displayed.

This property can be modified at any time. The default value is **false**; that is, the print progress dialog is displayed.

title

Type: String

The **title** property of the **Printer** class contains the title to be displayed on the system-supplied print progress dialog; for example:

```
app.printer.title := "Print Test - Line number " & lineCount.String;
```

If this property is empty, the application name is used. This property can be modified at any time.

topOfPage

Type: Integer

The **topOfPage** property of the **Printer** class contains the margin at the top of the printed page of output.

Specify the required value in millimeters, as shown in the following examples.

```
app.printer.topOfPage := 25;

app.printer.topOfPage := topOfPage.text.Integer;
```

This property can be modified at any time but it takes effect only at the start of the next page of output.

The default value is zero (**0**).

Printer Methods

The methods defined in the **Printer** class are summarized in the following table.

Method	Description
abort	Closes the printer or the preview file and discards all of the generated output
centreFrame	Centers the frame on the page
close	Sends all buffered output to the printer or the preview file
drawArc	Draws an elliptical arc on the printer page
drawChord	Draws a chord on the printer page (that is, an arc with the end points joined and the interior filled)
drawEllipse	Draws an ellipse on the printed page
drawFilledRectangle	Draws a filled rectangle on the printed page
drawGrid	Draws a grid
drawLine	Draws a line on the printed page
drawPie	Draws a pie-shaped wedge on the printed page
drawRectangle	Draws the border of a rectangle on the printed page
drawRoundRectangle	Draws a rectangle with rounded corners on the printed page
drawSolidRectangle	Draws a rectangle filled with the same color as the border on the printed page
drawTextAt	Draws a text string on the printer page
drawTextIn	Draws a text string with a bounded rectangle on the printer page
drawTextSize	Returns the size of the text on the print page, using the current **drawFont** property values
drawTextSizeIn	Returns the size of the text in a bounding rectangle on the print page, using the current **drawFont** property values
frameFits	Returns **true** if the selected frame can be fitted on the current printer page
getAllPaperSources	Returns all valid paper sources for the current printer device
getAllPrinterPaperSources	Returns the paper sources for the specified printer on the application server or a presentation client
getAllPrinters	Returns a string array of available printer names
getDefaultDocumentType	Returns the default document type that is set for the physical printer

Method	Description
getDefaultPaperSource	Returns the default paper source type (printer tray) set for the physical printer
getFooter	Returns the current footer frame for the printer or **null** if a footer frame has not been set
getHeader	Returns the current header frame for the printer or **null** if a footer frame has not been set
getPrintedStatus	Returns the status of the print task after the printer is closed, to enable you to determine if the report was printed during preview
getPrinterName	Returns the name of the current printer device
getPrintPosition	Returns the current pixel position to be used for the next print statement
getReport	Returns the value of the report instance set by the **app**.**printer**.**setReport** method call
isPrinterOpen	Returns **true** if the printer is currently open
newPage	Skips to the top of a new page
pageHeight	Returns the height of the page
pageWidth	Returns the width of the page
print	Outputs the specified frame to the printer
printActive	Prints the currently active form or control
printPage	Prints the specified page of print output on the current printer
printReport	Prints the specified report on the current printer if the **printPreview** property is set to **false**
printUnformatted	Bypasses the Windows Printer Control to print unformatted text
setCustomPaperSize	Sets a custom printer paper size at run time
setFooter	Sets the footer frame to a specified frame
setHeader	Sets the header frame to a specified frame
setMargins	Combines the settings of the margin properties
setPrinter	Sets the output printer
setPrintFileName	Writes the print output to a file
setPrintPosition	Sets the next print position to be used
setReport	Captures all **JadeReport** object **JadePrintDirect** and **JadePrintPage** entries for storage, manipulation, or printing
useCustomPrinterSettings	Combines standard printing properties (for example, **copies**, **duplex**, and so on) with previously cached advanced settings from the common Print Setup dialog

Printer class methods cannot be invoked from a server method.

abort

Signature `abort() updating;`

The **abort** method of the **Printer** class closes the printer or the print preview file and discards all of the generated output.

An exception is raised if this method is invoked from a server method.

The code fragment in the following example shows the use of the abort method.

```
if abortPrint.value then
    app.printer.abort;
else
    app.printer.close;
endif;
```

centreFrame

Signature `centreFrame(frame: Frame input) updating;`

The **centreFrame** method of the **Printer** class centers the frame specified in the **frame** parameter in the center of the horizontal axis of the page.

An exception is raised if this method is invoked from a server method.

The code fragment in the following example shows the use of the **centreFrame** method.

```
// Sets frame1 and frame3 of the form to be the header and footer frames.
// These frames are printed at the top and bottom of each page printed by
// the application.
app.printer.setHeader(frame1);
app.printer.setFooter(frame3);
app.printer.centreFrame(frame2);
```

close

Signature `close(): Integer updating;`

The **close** method of the **Printer** class sends all buffered output to the printer or the preview file. (The preview file becomes available for browsing at this point.) A **Printer** category global constant is returned, indicating the result of this method (for example, **Print_Successful**).

An exception is raised if this method is invoked from a server method.

The code fragment in the following example shows the use of the **close** method.

```
app.printer.print(lett.heading);
printLetterText(app.printer, lett);
// The close method sends all buffered output to the preview file.
// The preview file becomes available for browsing at this point.
app.printer.close;
epilog
    delete lett;            // Deletes the transient form instance
end;
```

The destructor invokes the **close** method when you delete a printer object.

Note By default, all property values are re-initialized to the JADE default values when the printer is closed. For details about retaining property values when the printer is closed, see the **Printer** class **retainCMDValues** property.

drawArc

Signature

```
drawArc(x1:     Integer;
        y1:     integer;
        x2:     Integer;
        y2:     Integer;
        startX: Integer;
        startY: Integer;
        endX:   Integer;
        endY:   Integer;
        color:  Integer);
```

The **drawArc** method of the **Printer** class draws an elliptical arc on the printer page, by using a pen the width of the **drawWidth** property value and the style of the **drawStyle** property. An exception is raised if this method is invoked from a server method. The **drawArc** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the arc is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinate of the point that defines the starting point of the arc
endX, endY	Logical *x* and *y* coordinate of the point that defines the end point of the arc
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page. The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the arc.

The starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The end point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse. As an arc is not a closed figure, it is not filled.

The width and height of a rectangle must each be in the range **2** units through **32,767** units.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawChord

Signature

```
drawChord(x1:     Integer;
          y1:     Integer;
          x2:     Integer;
          y2:     Integer;
          startX: Integer;
          startY: Integer;
          endX:   Integer;
          endY:   Integer;
          color:  Integer);
```

The **drawChord** method of the **Printer** class draws an elliptical arc on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

A line is drawn through the end points of the arc and the figure is filled by using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object.

An exception is raised if this method is invoked from a server method.

The **drawChord** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the arc is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinate of the point that defines the starting point of the chord
endX, endY	Logical *x* and *y* coordinate of the point that defines the end point of the chord
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page. The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the chord. The width and height of a rectangle must each be in the range **2** units through **32,767** units.

The starting point of the chord is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The end point of the chord is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawEllipse

Signature

```
drawEllipse(x1:    Integer;
            y1:    Integer;
            x2:    Integer;
            y2:    Integer;
            color: Integer);
```

The **drawEllipse** method of the **Printer** class draws an ellipse on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

The figure is filled by using the color and style of the **drawFillColor** and **drawFillStyle** properties. An exception is raised if this method is invoked from a server method.

The **drawEllipse** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle bounding the ellipse
x2, y2	Right and bottom points of the rectangle bounding the ellipse
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page.

If the width or the height of the bounding rectangle is zero (**0**), the ellipse is not drawn.

As the figure drawn by this method extends up to but does not include the right and bottom coordinate, the height of the figure is **y2** through **y1**, and the width is **x2** through **x1**.

The width and the height of a rectangle must be in the range **2** units through **32,767** units. To draw an unfilled ellipse, set the **drawFillStyle** property to **DrawFillStyle_Transparent** (1).

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawFilledRectangle

Signature

```
drawFilledRectangle(x1:    Integer;
                    y1:    Integer;
                    x2:    Integer;
                    y2:    Integer;
                    color: Integer);
```

The **drawFilledRectangle** method of the **Printer** class draws a rectangle on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

The figure is filled by using the color and style of the **drawFillColor** and **drawFillStyle** properties of the object.

An exception is raised if this method is invoked from a server method.

The **drawFilledRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page. If the width or the height of the rectangle is zero (**0**), the rectangle is not drawn.

As the figure drawn by this method extends up to but does not include the right and bottom coordinate, the height of the figure is **y2** through **y1**, and the width is **x2** through **x1**. The width and the height of a rectangle must be in the range **2** units through **32,767** units.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position. To draw an unfilled rectangle, use the **drawRectangle** method or set the **drawFillStyle** property to **DrawFillStyle_Transparent** (1).

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawGrid

Signature

```
drawGrid(x1:     Integer;
         y1:     Integer;
         x2:     Integer;
         y2:     Integer;
         style:  Integer;
         width:  Integer;
         height: Integer;
         color:  Integer) updating;
```

The **drawGrid** method of the **Printer** class draws a grid into the specified rectangle, by using the **Printer** class grid style constants listed in the following table.

Printer Class Constant	Value	Description
DrawGrid_Crosses	1	Small crosses drawn at the grid line intersection
DrawGrid_Dots	2	Dots drawn at the grid line intersections
DrawGrid_Lines	0	Horizontal and vertical grid lines

The **drawGrid** method parameters are listed in the following table.

Parameter	Description
x1, y1	Horizontal and vertical left and top points of the rectangle, respectively
x2, y2	Horizontal and vertical right and bottom points of the rectangle, respectively
style	**DrawGrid_Lines** (0), **DrawGrid_Crosses** (1), or **DrawGrid_Dots** (2)
width	Increment in pixels between each vertical grid line
height	Increment in pixels between each horizontal grid line
color	Color of the pen used to draw the grid

Grid lines for the left and top edges of the rectangle are not drawn.

The grid lines are drawn by using the **Printer**::**drawWidth** and **Printer**::**drawStyle** properties. For the line style **drawWidth** = 1, **drawWindow** = 0 (client area), and **scaleMode**=0 (pixels), the result is the same as if you wrote the following code.

```
vars
    x : Integer;
    y : Integer;
begin
```

```
        foreach x in self.width to self.clientWidth - 1 step 5 do
            window.drawLine(x, 0, x, self.clientHeight, self.color);
        endforeach;
        foreach y in self.height to self.clientHeight - 1 step 5 do
            window.drawLine(0, y, self.clientWidth, y, self.color);
        endforeach;
    end;
```

drawLine

Signature

```
drawLine(x1:    Integer;
         y1:    Integer;
         x2:    Integer;
         y2:    Integer;
         color: Integer);
```

The **drawLine** method of the **Printer** class draws a line on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

An exception is raised if this method is invoked from a server method.

The **drawLine** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top start points of the line, respectively
x2, y2	Right and bottom end points of the line, respectively
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page.

The line drawn by this method extends up to but does not include the end point.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawPie

Signature

```
drawPie(x1:     Integer;
        y1:     Integer;
        x2:     Integer;
        y2:     Integer;
        startX: Integer;
        startY: Integer;
        endX:   Integer;
        endY:   Integer;
        color:  Integer);
```

The **drawPie** method of the **Printer** class draws a pie-shaped wedge on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

The wedge is an elliptical arc whose center and two end points are joined by lines. The figure is filled by using the color and style of the **drawFillColor** and **drawFillStyle** properties.

An exception is raised if this method is invoked from a server method.

The **drawPie** method parameters are listed in the following table.

Parameter	Description
x1, y1, x2, y2	Rectangle bounding the ellipse of which the pie is a part
startX, startY	Logical *x* and *y* (horizontal and vertical) coordinate of the point that defines the starting point of the arc
endX, endY	Logical *x* and *y* coordinate of the point that defines the end point of the arc
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page. The **startX**, **startY**, **endX**, and **endY** parameter points do not need to lie exactly on the arc. The starting point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified starting point intersects the ellipse. The end point of the arc is the point at which a ray drawn from the center of the bounding rectangle through the specified end point intersects the ellipse.

The figure drawn by this function extends up to but does not include the right and bottom coordinate, so that the height of the figure is **y2** through **y1** and the width is **x2** through **x1**. The width and height of a rectangle must each be in the range **2** units through **32,767** units.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position. For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawRectangle

Signature

```
drawRectangle(x1:    Integer;
              y1:    Integer;
              x2:    Integer;
              y2:    Integer;
              color: Integer);
```

The **drawRectangle** method of the **Printer** class draws the border of a rectangle on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property.

The inside of the rectangle is not filled. An exception is raised if this method is invoked from a server method.

If the width or the height of the rectangle is zero (**0**), the rectangle is not drawn. As the figure drawn by this function extends up to but does not include the right and bottom coordinate, the height of the figure is **y2** through **y1** and the width is **x2** through **x1**. The figure drawn by this rectangle is equivalent to using the **drawFilledRectangle** method with the **drawFillStyle** property set to **DrawFillStyle_Transparent** (1).

The **drawRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Horizontal and vertical top points of the rectangle
x2, y2	Horizontal and vertical bottom points of the rectangle
color	Color of the pen used

All of the positional values are relative to the left and top margins, and need not lie within the page.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawRoundRectangle

Signature

```
drawRoundRectangle(x1:          Integer;
                   y1:          Integer;
                   x2:          Integer;
                   y2:          Integer;
                   xRoundSize:  Integer;
                   yRoundSize:  Integer;
                   color:       Integer);
```

The **drawRoundRectangle** method of the **Printer** class draws a rectangle with rounded corners on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property. The figure is filled by using the **drawFillColor** and **drawFillStyle** properties of the object.

An exception is raised if this method is invoked from a server method.

If the width or the height of the rectangle is zero (**0**), the rectangle is not drawn. As the figure drawn by this function extends up to but does not include the right and bottom coordinate, the height of the figure is **y2** through **y1** and the width is **x2** through **x1**. The width and the height of a rectangle must be in the range **2** units through **32,767** units.

The **drawRoundRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
xRoundSize	Width of ellipse for rounded corners
yRoundSize	Height of ellipse for rounded corners
color	Color of the pen used

All of the position values are relative to the left and top margins, and need not lie within the page.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawSolidRectangle

Signature
```
drawSolidRectangle(x1:    Integer;
                   y1:    Integer;
                   x2:    Integer;
                   y2:    Integer;
                   color: Integer);
```

The **drawSolidRectangle** method of the **Printer** class draws a rectangle on the printer page, by using a colored pen the width of the **drawWidth** property and the style of the **drawStyle** property. The figure is solidly filled, using the same color as the border. The **drawFillColor** and **drawFillStyle** properties are ignored. An exception is raised if this method is invoked from a server method.

The **drawSolidRectangle** method parameters are listed in the following table.

Parameter	Description
x1, y1	Left and top points of the rectangle
x2, y2	Right and bottom points of the rectangle
color	Color of the pen used

If the width or the height of the rectangle is zero (**0**), the function does not draw the rectangle. As the figure drawn by this function extends up to but does not include the right and bottom coordinate, the height of the figure is **y2** through **y1** and the width is **x2** through **x1**.

The figure drawn by this rectangle is equivalent to using the **drawFilledRectangle** method with the **drawFillColor** property set to the appropriate color and the **drawFillStyle** property set to **DrawFillStyle_Solid** (0).

All of the position values are relative to the left and top margins, and need not lie within the page.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextAt

Signature
```
drawTextAt(text:  String;
           x1:    Integer;
           y1:    Integer;
           color: Integer);
```

The **drawTextAt** method of the **Printer** class draws a text string on the printer page, using the current values of the **drawFont**, **drawTextRotation**, **drawTextCharRotation**, and **drawTextAlign** properties. An exception is raised if this method is invoked from a server method.

The **drawTextAt** method parameters are listed in the following table.

Parameter	Description
text	Text string that is to be drawn
x1, y1	Horizontal and vertical positions for the text
color	Color of the text

The way in which the text is drawn is determined by the value of the **drawTextAlign** property, as listed in the following table.

Printer Class Constant	Value	Description
DrawTextAlign_Center	2	Center-aligned (centered) around **x1**
DrawTextAlign_Left	0	Left-aligned from **x1**
DrawTextAlign_Right	1	Right-aligned at **x1**

The text is drawn in a single line, unless it has embedded carriage return characters within it. Each embedded carriage return character forces a new line for the remaining text. All of the position values are relative to the left and top margins, and need not lie within the page.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextIn

Signature

```
drawTextIn(text:  String;
           x1:    Real;
           y1:    Real;
           x2:    Real;
           y2:    Real;
           color: Integer);
```

The **drawTextIn** method of the **Printer** class draws a text string on the printer page within a bounded rectangle, using the current values of the **drawFont**, **drawTextRotation**, **drawTextCharRotation**, and **drawTextAlign** properties. An exception is raised if this method is invoked from a server method.

The **drawTextIn** method parameters are listed in the following table.

Parameter	Description
text	Text string that is to be drawn
x1, y1	Left and top points of the bounding rectangle
x2, y2	Right and bottom points of the bounding rectangle
color	Color of the text

The text is drawn into the bounding rectangle with word wrap.

The way in which the text is drawn is determined by the value of the **drawTextAlign** property, as listed in the following table.

Window Class Constant	Value	Description
DrawTextAlign_Center	2	Center-aligned (centered)
DrawTextAlign_Left	0	Left-aligned
DrawTextAlign_Right	1	Right-aligned

Any embedded carriage return character within the text forces a new line for the remaining text.

The text always starts at the vertical point specified by the **y1** parameter. All of the position values are relative to the left and top margins, and need not lie within the page.

This method causes the header frame to be printed if the method is called at the start of a new page. The method has no subsequent affect on the current print position.

This method is not normally suitable with non-zero values of the **drawTextCharRotation** and **drawTextRotation** properties, as the rotation may cause some of the text to be outside the requested rectangle and therefore it may not be totally visible.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextSize

Signature
```
drawTextSize(str:        String;
             textHeight: Integer output): Integer;
```

The **drawTextSize** method of the **Printer** class returns the size of the text, using the current values of the **drawFont** properties. An exception is raised if this method is invoked from a server method.

The text alignment and the text rotation properties (that is, **drawTextAlign**, **drawTextCharRotation**, and **drawTextRotation**) are not used in determining this size.

The parameters for the **drawTextSize** method are listed in the following table.

Parameter	Description
str	Text string that is to be measured
textHeight	Text height returned

The **drawTextSize** method returns the text width, which can be used in the **drawTextAt** method.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

drawTextSizeIn

Signature
```
drawTextSizeIn(str:        String;
               width:      Integer;
               textHeight: Integer output): Integer;
```

The **drawTextSizeIn** method of the **Printer** class returns the size of the text by using the current values of the **drawFont** properties, setting a bounding rectangle equal to the **width** parameter, and applying word wrap. An exception is raised if this method is invoked from a server method.

The text alignment and the text rotation properties (that is, **drawTextAlign**, **drawTextCharRotation**, and **drawTextRotation**) are not used in determining this size.

The parameters for the **drawTextSizeIn** method are listed in the following table.

Parameter	Description
str	Text string that is to be measured
width	Width of the rectangle (in pixels) for word wrap
textHeight	Text height returned

The **drawTextSizeIn** method returns the text width, which you could then use in the **drawTextIn** method.

For details about placing output directly on a printer page at any location without the use of frames and drawing rotated text and characters, see "Free-Format Printing", later in this chapter.

frameFits

Signature `frameFits(fr: Frame): Boolean updating;`

The **frameFits** method of the **Printer** class returns **true** if the frame specified in the **frame** parameter can be fitted within the remaining height of the current printer page (the width is not checked), excluding the space taken by the current footer. An exception is raised if this method is invoked from a server method.

The code fragments in the following examples show the use of the **frameFits** method.

```
if not app.printer.frameFits(printForm.detail1) then
    timeTaken := (app.clock - startTime).Time;
    printForm.timeTaken1.caption := timeTaken.String;
    printForm.timeTaken2.caption := timeTaken.String;
    checkForHeaderSwap(printForm);
    startTime := app.clock;
    if cb_ChangeMargins.value then
        leftMargin.text := (leftMargin.text.Integer + 5).String;
        app.printer.leftMargin  := leftMargin.text.Integer;
        app.printer.rightMargin := rightMargin.text.Integer + 5;
    endif;
endif;

while app.printer.frameFits(printForm.fillerFrame) do
    result := app.printer.print(printForm.fillerFrame);
    if result <> 0 then
        break outerWhile;
    endif;
endwhile;
```

getAllPaperSources

Signature `getAllPaperSources(ia: IntegerArray input;`
`sa: StringArray input): Integer updating;`

The **getAllPaperSources** method of the **Printer** class populates integer and string arrays with the numbers and names, respectively, of the paper sources available for the current printer.

The return value represents the number of paper sources for the current printer.

The following example shows the use of the **getAllPaperSources** method.

```
vars
    ia : IntegerArray;
    sa : StringArray;
begin
    create sa transient;
    create ia transient;
    app.printer.getAllPaperSources(ia, sa);
    write "Paper sources = " & app.printer.getAllPaperSources(ia,
                               sa).String;  // Outputs Paper sources  = 10
```

```
epilog
    delete sa;
    delete ia;
end;
```

For details about returning the valid paper sources for a specified printer device on the application server or a presentation client, see the **Printer** class **getAllPrinterPaperSources** method.

getAllPrinterPaperSources

Signature `getAllPrinterPaperSources(printerName: String;`
`ia: IntegerArray input;`
`sa: StringArray input): Integer updating;`

The **getAllPrinterPaperSources** method of the **Printer** class populates integer and string arrays with the numbers and names, respectively, of the paper sources available for the printer specified in the **printerName** parameter.

The return value represents the number of paper sources for the specified printer.

In JADE thin client mode, this method returns:

- Paper sources for a specified printer local to the presentation client only, when executed on a presentation client
- Paper sources for a specified printer attached to that server, when called from a method executing on the server node (which enables you to get the paper sources for an application server printer if you want to schedule report tasks for a reporting presentation client that runs on the application server)

This method has no impact on the **Printer** object that is used to call the method.

For details about returning the valid paper sources for the current printer device, see the **Printer** class **getAllPaperSources** method.

getAllPrinters

Signature `getAllPrinters(sa: Array input): Integer updating;`

The **getAllPrinters** method of the **Printer** class fills an array with the names of the available printers. The return value represents the number of printers in the array.

The array type passed to the method must be an array with a membership of **String**, to allow the array type to be a **HugeStringArray** or a **StringArray**. If another array type with a membership other than **String** is passed to the method, exception 1000 *(Invalid parameter type)* is raised.

In JADE thin client mode, this method returns:

- Printers local to the presentation client only, when executed on a presentation client
- Printers attached to that server, when called from a method executing on the server node (which enables you to get a list of valid printers for the application server if you want to schedule report tasks for a reporting presentation client that runs on the application server)

The following example shows the use of the **getAllPrinters** method.

```
load() updating;
vars
    sa : Array;
begin
```

```
    // Creates an array and populates it with the currently
    // available printers using the getAllPrinters method. The array
    // is then displayed in a combo box, allowing the user to select
    // a printer.
    create sa transient;
    app.printer.getAllPrinters(sa);
    comboBox.listCollection(sa, false, 0);
    ...                              // do some more processing here
epilog
    // Deletes the transient array object.
    delete sa;
end;
```

getDefaultDocumentType

Signature `getDefaultDocumentType(): Integer updating;`

The **getDefaultDocumentType** method of the **Printer** class returns the document type that is set for the physical printer.

This method opens the printer if it is not already open. The printer document default is that set by the user for the specific physical printer. A user can set a default value for the **documentType** property for a printer by:

1. Opening the Windows Printers form (from the Start menu **Printers** command in the Settings submenu, for example)
2. Right-clicking on the icon for the specific printer
3. Selecting the **Document Defaults** or **Printing Preferences** command from the popup menu that is displayed, depending on the version of the Windows operating system that is running

Note This method does not return the type of paper in the printer, but only the default type set by the user. For most printers, the default **documentType** is **A4**.

The JADE default value for the **documentType** property is **A4**. The following code fragment sets the default document type for users.

```
app.printer.documentType := app.printer.getDefaultDocumentType;
```

getDefaultPaperSource

Signature `getDefaultPaperSource(): Integer updating;`

The **getDefaultPaperSource** method of the **Printer** class returns the default paper source (printer tray) that is set for the physical printer. This method opens the printer if it is not already open. The default printer paper source is that set by the user for the specific physical printer.

A user can set a default value for the **paperSource** property for a printer by:

1. Opening the Windows Printers form (from the Start menu **Printers** command in the Settings submenu, for example)
2. Right-clicking on the icon for the specific printer
3. Selecting the **Document Defaults** or **Printing Preferences** command from the popup menu that is displayed, depending on the version of the Windows operating system that is running

Note This method does not return the paper source that is set for the printer itself, but only the default paper source set by the user. For most printers, the default **paperSource** is **Automatically Select**.

The JADE default value for the **paperSource** property is **Automatically Select**. The following code fragment sets the default paper source for users.

```
app.printer.paperSource := app.printer.getDefaultPaperSource;
```

getFooter

Signature `getFooter(): Frame;`

The **getFooter** method of the **Printer** class returns the current value of the **footerFrame** property for the printer or **null** if a header frame has not been set.

getHeader

Signature `getHeader(): Frame;`

The **getHeader** method of the **Printer** class returns the current value of the **headerFrame** property for the printer or **null** if a header frame has not been set.

getPrintedStatus

Signature `getPrintedStatus(): Integer;`

The **getPrintedStatus** method of the **Printer** class returns the status of the print task after the printer is closed, to enable you to determine if the user printed the report during preview or if the user aborted or cancelled the print task.

The values returned by this method are valid only after the printer has been closed. The printed status is cleared the next time the printer is opened.

The following table lists the values that can be returned by the **getPrintedStatus** method.

Printer Class Constant	Integer Value	Description
PrintedStatus_Not	0	No printing occurred
PrintedStatus_All	1	The entire report was printed
PrintedStatus_Partial	2	The user printed specific pages only in print preview
PrintedStatus_Aborted	3	The report printing was aborted
PrintedStatus_Cancelled	4	The report printing process was canceled, producing only partial output

getPrinterName

Signature `getPrinterName(): String;`

The **getPrinterName** method of the **Printer** class returns the name of the current printer; for example:

```
HP LaserJet 4Si/4Si MX PS
```

Exception 15021 *(No default printer exists for this user)* is raised, if you do not have a printer set up.

getPrintPosition

Signature `getPrintPosition(): Integer;`

The **getPrintPosition** method of the **Printer** class returns the current pixel position to be used for the next print statement on a page. This position is initially set to zero (**0**) after an end-of-page condition. (The pixel position is zero-relative to the top margin of the page.) An exception is raised if this method is invoked from a server method.

This method is not relevant for direct printing (that is, when the **formatOut** property option is set to **=direct**).

getReport

Signature `getReport(): JadeReport;`

The **getReport** method of the **Printer** class returns the value of the report instance set by the **app.printer.setReport(report)** method call during the print production.

If the **setReport** method is not called, the **getReport** method returns **null**.

Note The reference to the report is cleared when the printer is closed and the **getReport** method then returns **null**.

isPrinterOpen

Signature `isPrinterOpen(): Boolean;`

The **isPrinterOpen** method of the **Printer** class returns **true** if any output has been directed to the printer or to print preview in the current application and the printer or print preview has not yet been closed; that is, an **app.printer.print** was issued but no **app.printer.close** has yet been performed. An exception is raised if this method is invoked from a server method.

The code fragment in the following example shows the use of the **isPrinterOpen** method.

```
if app.printer.isPrinterOpen = true then
    if self.printoutWanted then
        app.printer.close;
    else
        app.printer.abort;
    endif;
endif;
```

newPage

Signature `newPage(): Integer updating;`

The **newPage** method of the **Printer** class skips to the top of a new page and it opens the printer if it is not already open. This method has no effect if the print output is already positioned at the start of a page.

A **Printer** category global constant is returned, indicating the result of this method (for example, **Print_Successful**).

An exception is raised if this method is invoked from a server method.

The code fragments in the following examples show the use of the **newPage** method.

```
// The variable lastKey contains the key of the last entry to be
// displayed on the table.  This value is compared with the key of the
// last entry in the collection to determine whether the entire collection
// has been displayed.  If not, the printer is told to start a new page
// and the table is cleared.  The startPos variable is then set to the
// last entry to be displayed so that the next page displays only the
// entries which have not yet been displayed.
if lastKey < customerDict.last.key then
    app.printer.newPage;
    table1.clear;
    table1.rows := 1;
    self.setColumnHeadings;
    self.startPos := lastKey;
endif;

if cb_NewPage.value then
    if lineCount mod tb_Lines.text.Integer = 0 then
        if cb_ChangeMargins.value then
            leftMargin.text := (leftMargin.text.Integer + 5).String;
            app.printer.leftMargin  := leftMargin.text.Integer;
            app.printer.rightMargin := rightMargin.text.Integer + 5;
        endif;
        app.printer.newPage;
    endif;
endif;
```

pageHeight

Signature `pageHeight(): Integer updating;`

The **pageHeight** method of the **Printer** class returns the height of the page, in pixels. The page height that is returned excludes the borders. An exception is raised if this method is invoked from a server method.

pageWidth

Signature `pageWidth(): Integer updating;`

The **pageWidth** method of the **Printer** class returns the width of the page, in pixels. The page width that is returned excludes the borders. An exception is raised if this method is invoked from a server method.

print

Signature `print(win: Frame input): Integer updating;`

The **print** method of the **Printer** class outputs the frame specified in the **win** parameter to the printer. The space used by the frame on the printed page is determined by the currently set height of the frame (that is, it can be changed during printing) rather than the height of the frame that was painted during the form definition in the JADE Painter.

An exception is raised if this method is invoked from a server method.

For details about calling the **print** method when printing a background picture over which is drawn the report itself, see "Layering Print Output", earlier in this section.

Note Unpredictable results will occur if you mix **print** and **printUnformatted** method calls within the same print task.

The **Printer** category global constants that can be returned by the **print** method are listed in the following table.

Return Value	Global Constant	Description
0	Print_Successful	The print was successful
15001	Print_Invalid_Control	You have attempted to print a control that is not a frame
15002	Print_NewPage_Failed	When trying to print a new page, the printer may be off-line or incorrectly configured
15003	Print_Printer_Not_Open	You attempted to close a printer that is not currently open
15004	Print_TextOut_Error	When trying to print a frame, the printer may be off-line or incorrectly configured
15005	Print_Printer_Open_Failed	When trying to print a frame, the printer may be off-line or incorrectly configured
15006	Print_Header_Footer_Too_Large	A header or footer frame has exceeded the page depth (height)
15007	Print_Frame_Too_Large	You attempted to print a frame that has a depth greater than the page height
15008	Print_Preview_Ignored	You attempted to change the **printPreview** property after printing has begun
15010	Print_Copies_Ignored	You attempted to change the **copies** property after printing has begun
15011	Print_Orientation_Invalid	You have assigned a value other than **Print_Portrait** or **Print_Landscape** to the **orientation** property
15013	Print_Currently_Open	The specified printer is currently open (that is, the application currently has the printer open)
15014	Print_Failed_To_Obtain_Printer	The task failed to obtain the specified printer (that is, the API call to obtain the available printers has failed)
15015	Print_Cancelled	The **Cancel** button on the runtime Print progress dialog was clicked
15016	Print_Stopped	The **Stop** button on the runtime Print progress dialog was clicked
15017	Print_Not_Available	The specified printer does not match the available printers
15021	Print_NoDefaults_Printer	Your workstation has no default printer set up
15022	Print_PrintReport_Ignored	Printing started, so change of print report ignored
15023	Print_Printer_Ignored	You attempted to change the printer in use after printing has begun or before any printing has occurred (the printer must be closed before commencing new output on a different printer)

Return Value	Global Constant	Description
15024	Print_Invalid_Position	Attempt to set a print position that is outside the valid range
15025	Print_Unformatted_failed	Printing of unformatted text failed; that is, the **printUnformatted** method request failed
15026	Print_PaperSource_Ignored	Printing started, so change to the **paperSource** property ignored
15027	Print_PaperSource_Invalid	Value of the **paperSource** property is invalid
15028	Print_Duplex_Invalid	Value of the **duplex** property is invalid
15029	Print_Duplex_Ignored	Printing started, so change to the **duplex** property ignored
15030	Print_Collate_Ignored	Printing started, so change to the **collate** property ignored
15031	Print_In_Preview	Value of the **printPreview** property is **true** so printer object cannot be reused
15032	Print_DocumentType_Invalid	You changed **printer.documentType** to **Print_Custom_Paper** instead of calling the **printer.setCustomPaperSize** method
15033	Print_Metafile_Playback_Error	Internal error occurred when attempting to play back a print metafile

If a value of **15015** or **15016** is returned (that is, printing was canceled or stopped), the printer has been closed and code is required in your method to logically end the print processing.

The following example shows the use of the **print** method.

```
buttonPrint_click(btn: Button input) updating;
vars
    result : Integer;
    report : ReportForm;
begin
    // Creates ReportForm transient class instance and references it by the
    // report local variable, which can be used to access the form controls.
    create report;
    // Specifies output is not directed to the preview file before printing.
    app.printer.printPreview := false;
    // Specifies the format of the pages to be printed.  As these are
    // default values, it is unnecessary to redefine them unless you
    // require a different format.
    app.printer.orientation  := Print_Portrait;
    app.printer.documentType := Print_A4;
    // Uses the print method to output frame2 of the form to the print
    // file twice.  The return value is stored in the result local
    // variable, and is checked to ensure that the print task has not
    // been stopped or cancelled.  The method returns if this is the case.
    // The close method then sends all buffered output to the printer
    // that prints the document.
    result := app.printer.print(report.frame2);
    if result = Print_Stopped or result = Print_Cancelled then
        return;
    endif;
```

```
    result := app.printer.print(report.frame2);
    if result = Print_Stopped or result = Print_Cancelled then
        return;
    endif;
epilog
    app.printer.close;
    delete report;      // Deletes the transient form instance.
end;
```

printActive

Signature `printActive(win: Window): Integer updating;`

The **printActive** method of the **Printer** class prints the specified **Form** or **Control**, as shown in the following example.

```
bPrint_click(button: Button);
vars
    result : Integer;
begin
    app.printer.setMargins(Print_Landscape, 25, 25, 25, 25);
    result := app.printer.printActive(drawing);
    app.printer.close;
end;
```

Use the **win** parameter to specify a form or control that is to be printed. This method returns **Print_Successful** (**0**) if the active form or control printed successfully. An exception is raised if this method is invoked from a server method.

Note Unpredictable results occur if you mix **print** and **printUnformatted** method calls within the same print task.

printPage

Signature `printPage(page: JadePrintData) updating;`

The **printPage** method of the **Printer** class prints the output specified in the **page** parameter on the current printer. The **JadePrintData** object that is specified in the page parameter can be a **JadePrintDirect** or a **JadePrintPage** object.

printReport

Signature `printReport(report: JadeReport) updating;`

The **printReport** method of the **Printer** class prints the report specified in the **report** parameter on the current printer if the **printPreview** property is set to **false**. If the **printPreview** property is set to **true**, the specified report is displayed in print preview mode on the display device of the node in which the method is executed.

printUnformatted

Signature `printUnformatted(text: String): Integer updating;`

The **printUnformatted** method of the **Printer** class takes unformatted print text and passes it to the printer, bypassing the Windows Printers Control. This method enables you to output text to a dot matrix printer; for example, using your specified page depth.

Use the **text** parameter to specify the unformatted text that is to be printed.

You must embed all control commands that you require in your parameter string; for example, top-of-page, carriage return, line feed, and so on.

Notes Unpredictable results will occur if you mix **print** and **printUnformatted** method calls within the same print task.

This method is compatible only with dot matrix, daisy wheel, or a printer that is capable of printing a single line or single line feed. It does not work with LaserJet printers, which must print a page at a time. An exception is raised if the method fails.

The following example shows the use of the **printUnformatted** method.

```
vars
    text    : String;
    loop    : Integer;
    printer : String;
begin
    // Set the printer to the IBM Proprinter
    printer := 'IBM Proprinter XL';
    app.printer.setPrinter(printer);
    // Set page depth to 3 inches - don't bother setting up the documentType
    text := #'1b 43 00 03';
    app.printer.printUnformatted(text);
    // Print 4 * 4 labels
    loop := 1;
    while loop < 50 do
        loop := loop + 1;
        text := CrLf & CrLf & CrLf;
        app.printer.printUnformatted(text);
        text := '         100831.7                    100831.8
                 100831.7                    100831.8     ' & CrLf & CrLf;
        app.printer.printUnformatted(text);
        text := 'Sockburn                     Sockburn
                 Sockburn                    Sockburn' & CrLf & CrLf;
        app.printer.printUnformatted(text);
        text := '123456                       123457
                 1234568                     1234569     ' & CrLf & CrLf;
        app.printer.printUnformatted(text);
        // Skip to top of page ( 3inch page );
        text := #"0C";
        app.printer.printUnformatted(text);
    endwhile;
    app.printer.close;
end;
```

The **Printer** category global constant values that can be returned by the **printUnformatted** method are listed in the following table.

Return Value	Global Constant	Description
0	Print_Successful	The print was successful
15003	Print_Printer_Not_Open	The printer is not open
15004	Print_TextOut_Error	Text output to printer failed

Return Value	Global Constant	Description
15005	Print_Printer_Open_Failed	Opening the printer failed (it may be off-line or incorrectly configured)
15013	Print_Currently_Open	The printer is currently open
15014	Print_Failed_To_Obtain_Printer	The task failed to obtain the printer
15017	Print_Not_Available	The printer does not match the available printers
15025	Print_Unformatted_failed	Printing of unformatted text failed

An exception is raised if this method is invoked from a server method.

setCustomPaperSize

Signature `setCustomPaperSize(width: Integer; height: Integer) updating;`

The **setCustomPaperSize** method of the **Printer** class dynamically set a custom printer paper size at run time. Specify the values of the **width** and **height** parameters in units of a tenth of a millimeter; for example, calling **app.printer.setCustomPaperSize(2100, 2970);** sets the paper size equivalent to **Print_A4**. The **setCustomPaperSize** method sets the **Printer** class **documentType** to the **Printer** category **Print_Custom_Paper** (256) global constant value and sets the paper size to the specified width and height.

Notes Changing the value of the **printer.documentType** property resets any customized paper size. Changing **printer.documentType** to **Print_Custom_Paper** raises exception 15032 (you must call the **setCustomPaperSize** method).

Calling the **setCustomPaperSize** method during printing causes a **printer.newPage** call before being applied if printing is not currently at the start of a new page.

setFooter

Signature `setFooter(fr: Frame) updating;`

The **setFooter** method of the **Printer** class sets the footer frame to the frame specified in the **fr** parameter.

To clear the footer, call the **setFooter** method, passing **null** as the frame. The footer frame can be changed at any time.

The following example shows the use of the **setFooter** method.

```
setupHeaderFooterOrientation(printForm: PrintForm);
begin
    if portrait.value then
        app.printer.orientation := Print_Portrait;
    else
        app.printer.orientation := Print_Landscape;
    endif;
    if header.value then
        app.printer.setHeader(printForm.header);
    else
        app.printer.setHeader(null);
    endif;
    if footer.value then
```

```
            app.printer.setFooter(printForm.footer);
        else
            app.printer.setFooter(null);
        endif;
    end;
```

setHeader

Signature `setHeader(fr: Frame) updating;`

The **setHeader** method of the **Printer** class sets the header frame to the frame specified in the **fr** parameter.

To clear the header, call the **setHeader** method, passing **null** as the frame. The header frame can be changed at any time but it takes effect only at the start of the next page of output.

The following example shows the use of the **setHeader** method.

```
checkForHeaderSwap(printForm : PrintForm);
begin
    if headerSwap.value then
        if app.printer.pageNumber.isEven then
            app.printer.setHeader(printForm.header2);
        else
            app.printer.setHeader(printForm.header);
        endif;
    endif;
    if footerSwap.value then
        if app.printer.pageNumber.isEven then
            app.printer.setFooter(printForm.footer2);
        else
            app.printer.setFooter(printForm.footer);
        endif;
    endif;
end;
```

setMargins

Signature

```
setMargins(orient: Integer;
           top:    Real;
           bottom: Real;
           left:   Real;
           right:  Real): Integer updating;
```

The **setMargins** method of the **Printer** class combines the settings of the properties for:

- Orientation
- Top of page
- Bottom of page
- Left margin
- Right margin

You can change the margins at any time, but the top and right margins take effect only at the start of the next page of output.

An exception is raised if this method is invoked from a server method.

The following example shows the use of the **setMargins** method.

```
bPrint_click(button: Button);
vars
    printer : Printer;
begin
    create printer transient;
    printer.setMargins(Print_Landscape, 25, 25, 25, 25);
    if graphFolder.topSheet = bgFrame then
        printer.printActive(bgFrame);
    else
        printer.printActive(lgFrame);
    endif;
epilog
    printer.close;
    delete printer;
end;
```

The **orient** parameter is the first to be specified, with a value of **Print_Portrait** (**1**) or **Print_Landscape** (**2**). The subsequent parameters for the **setMargins** method are specified in millimeters.

The **setMargins** method always returns zero (**0**).

setPrinter

Signature `setPrinter(name: String io): Integer updating;`

The **setPrinter** method of the **Printer** class enables you to programmatically set the output printer, by specifying a valid printer in the **name** parameter.

To reset the printer back to the default printer of the user, pass an empty string in the **name** parameter, and the method updates the string with the name of the default printer. The current printer must be closed for this to be valid; that is, you cannot pass a **null** value when the printer is active.

Note The **setPrinter** method causes the current printer to be closed and all printer properties to be re-initialized (with the possible exception of those controlled by the **retainCMDValues** property). You should therefore call the **app**.**printer**.**setPrinter** method before you set any other printer values.

The printer cannot be altered after printing has begun. The return values, represented by global constants in the **Printer** category, are listed in the following table.

Return Value	Global Constant	Description
0	Print_Successful	The print to the specified printer was successful.
15013	Print_Currently_Open	The specified printer is currently open. (The application currently has the printer open.) A resumable exception is raised and the method returns this **15013** value.
15014	Print_Failed_To_Obtain_Printer	The task failed to obtain the specified printer. (The API call to obtain the available printers has failed.)
15017	Print_Not_Available	The specified printer does not match the available printers.

Return Value	Global Constant	Description
15023	Print_Printer_Ignored	You attempted to change the printer in use after printing has begun, before any printing has occurred (the printer must be closed before commencing new output on a different printer).
15032	Print_DocumentType_Invalid	You changed **printer.documentType** to **Print_Custom_Paper** instead of calling the **printer.setCustomPaperSize** method.

The default printer is re-evaluated every time a new print task is initiated and JADE logic has not specifically set the printer name required. If logic sets a specific printer name (even if is the default printer), that printer continues to be used, regardless of any change to the default printer.

In JADE thin client mode, this method sets the local (presentation client) printer to the specified printer.

An exception is raised if this method is invoked from any of the following.

- A **serverExecution** method.
- A server application running under the **jadrap.exe** JADE Remote Node Access utility (because printing requires the **jade.exe** program).

The following examples show the use of the **setPrinter** method.

```
comboBox_click(combobox: ComboBox input) updating;
vars
    printer : String;
    result  : Integer;
begin
    // Uses the setPrinter method to set the printer when the user
    // selects one from the combo box.
    printer := comboBox.listObject.String;
    result  := app.printer.setPrinter(printer);
    if result = Print_Not_Available or result = Print_Currently_Open then
        app.msgBox('Printer is not available', 'Error',
                   MsgBox_Exclamation_Mark_Icon);
        return;
    endif;
end;

buttonUnload_click(btn: Button input) updating;
vars
    default : String;
begin
    default := "";
    app.printer.setPrinter(default);
    self.unloadForm;  // Unloads the form and resets to the default printer
end;
```

setPrintFileName

Signature `setPrintFileName(name: String);`

The **setPrintFileName** method of the **Printer** class requests the printer to write the print output to the file specified in the **name** parameter in the current directory.

In JADE thin client mode, this method always writes the print output to a file on the presentation client. This method is ignored if it is not called before the printer is opened.

Notes It is the responsibility of the printer driver to format and create the output file from the print commands that are issued. Although this method may therefore not be supported by some printers, most postscript printers can create print files. If using Postscript printing, your printer must support Postscript level 2 or greater.

The type of file suffix that you specify is dependent on the type of print file created by your printer driver. For example, a printer driver may enable you to create a .**prn** print file.

The following example shows the use of the **setPrintFileName** method.

```
directTest();
vars
    report : ReportForm;
begin
    create report;
    report.tb1.formatOut := '';
    app.printer.setPrintFileName("TestFile.prn");
//  app.printer.setPrintFileName("FILE:");      // displays a dialog
    app.printer.print(report.directFrame);
epilog
    app.printer.close;
    delete report;
end;
```

If the **name** parameter is set to **"FILE:"**, the Print To File dialog is displayed by the printer, enabling the user to specify the output file name that they require. (The **Output File Name** text box in this dialog also enables you to specify a valid existing path, if required; for example, **d:\jade\MyFile.prn**.)

setPrintPosition

Signature `setPrintPosition(pos: Integer) updating;`

The **setPrintPosition** method of the **Printer** class sets the next print position to be used. (The position is zero-relative to the top margin of the page.)

This method enables logic to print a left panel, reset to a specified position, and then print a right panel, for example.

The position specified in the pos parameter must be greater than or equal to zero (**0**) and less than the value returned by the **app.printer.pageHeight** method. The print position can be anywhere on the page, including being reset to line positions that have already been printed on.

If the value specified in the **pos** parameter is less than zero (**0**) or greater than or equal to the value returned by the **app.printer.pageHeight** method, an error is returned.

An exception is raised if this method is invoked from a server method.

This method is not relevant for direct printing (that is, when the **formatOut** property option is set to **=direct**).

For details about calling the **setPrintPosition** method when printing a background picture over which is drawn the report itself, see "Layering Print Output", earlier in this section.

setReport

Signature `setReport(report: JadeReport) updating;`

The **setReport** method of the **Printer** class captures all **JadeReport** object **JadePrintDirect** and **JadePrintPage** objects for storage, manipulation, or printing.

You can call this method before any print output is created, to set the **JadeReport** transient object into which any report output is stored.

If the **printPreview** property is not set to **true**, the print output is not sent to the defined printer and the **JadeReport** object retains the print output after the printer is closed. If the **printPreview** property is set to **true**, the print preview process occurs as normal and the **JadeReport** object retains the print output on completion.

The **JadeReport** data can be manipulated, stored, or printed to meet your requirements.

Note To store the printed output, a persistent copy of the report output must be made in the **JadeReport**, **JadePrintDirect**, and **JadePrintPage** subclasses.

Delete the **JadeReport** object when the process has completed. The delete process also removes the **JadePrintData** objects that it references.

When running in JADE thin client mode and your application calls the **setReport** method to indicate that user logic subsequently stores or manipulates the report output, each page of output is transferred to the application server. (For details about optimizing print preview performance, see "Previewing Print Output", later in this section.)

When you use the **formatOut** property **=pagenofm** or **=totalpages** option for formats of data in text boxes or labels and the report is being stored in the database (that is, the report uses the **Printer** class **setReport** method), output is retrieved from a temporary file and stored in the database only after the printer is closed. (This is most evident when running in JADE thin client mode, as the printed output must be retrieved from the presentation client and passed to the application server at the end of the report rather than page by page as the report is produced.) For details, see the **TextBox** class or **Label** class **formatOut** property.

useCustomPrinterSettings

Signature `useCustomPrinterSettings(): Integer;`

The **useCustomPrinterSettings** method of the **Printer** class instructs the JADE printing engine to combine standard printing properties (for example, **copies**, **duplex**, and so on) with *advanced* settings that were cached from the most-recent occasion the common Print Setup dialog was opened. The return value of zero (**0**) indicates that the method executed successfully.

For details about setting advanced properties, see "Using the Common Print Setup Dialog", in the following section.

Using the Common Print Setup Dialog

The Windows Common Print Setup dialog enables you to set options for a print task. Some options are standard and apply to most printers; for example, **copies**, **duplex**, **orientation**, and so on. You can set these properties in your JADE code. These properties are combined with the default settings of the printer to carry out the printing task in JADE.

Other advanced options are specific to the printer; for example, the number of pages per sheet and booklet printing. You cannot set these *advanced* options in your JADE code. These options are normally set on an *advanced settings* sheet on the Print Setup dialog. The **useCustomPrinterSettings** method instructs the JADE printing engine to combine the standard JADE printing properties (**copies**, **duplex**, **orientation**, and so on) with *advanced* settings that were cached from the most-recent occasion the common Print Setup dialog was opened.

The following example shows a common Print Setup dialog being displayed by setting the value of the **printSetup** property to **true**. Before starting to print, the **useCustomPrinterSettings** method is called so that the *advanced* settings from the Print Setup dialog are used instead of the default printer settings.

```
vars
    dialog : CMDPrint;
begin
    create dialog transient;
    dialog.initializeWith := CMDPrint.InitializeWith_MostRecentSetup;
    dialog.printSetup := true;
    if dialog.open <> null then
        return;      // User clicked Cancel button or Windows error occurred
    endif;
    // To use the printer-specific options you set on the Print Setup dialog
    app.printer.useCustomPrinterSettings();
    // Printing instructions such as "app.printer.print" would follow
    ...
end;
```

Notes This method is available only when the value of the **PrintDataType** parameter in the [JadePrinting] section of the JADE initialization file is set to **GDI**.

A JADE application can remember one set of advanced printer settings only. The set of properties can be different for each application.

When the Print Setup dialog is displayed more than once, only the last set of settings are cached.

Each printer retains the *advanced* settings until the print task is complete; that is, until the **close** method is called.

Using the Print Progress Dialog

The system-supplied Print Progress dialog is displayed on the first execution of the **print** statement when the **suppressDialog** property is set to the default value of **false**.

The current page number and the user-supplied title are displayed.

The Print Progress dialog contains two buttons, as follows.

- The **Cancel** button cancels the print function and discards the output. **Print_Cancelled** is returned in response to the **print** statement.
- The **Stop** button terminates the printing but does not discard the output. **Print_Stopped** is returned in response to the **print** statement.

Note Your method should contain code to logically end print processing if a **Print_Cancelled** or **Print_Stopped** value is returned. (For details, see the **Printer** class **print** method.)

Examples of Printer Methods

The method shown in the following example prints a currently active form.

```
bPrint_click(btn: Button input);
vars
    result : Integer;
begin
    result := app.printer.printActive(self);
    app.printer.close;
end;
```

The following method shows how **Printer** methods can be referenced.

```
vars
    count, result : Integer;
    rpf           : RepPrintFaults;
    today         : Date;
begin
    create rpf;
    app.printer.setMargins(Print_Portrait, 10, 10, 10, 10);
    app.printer.setHeader(rpf.heading);
    app.printer.pageBorderWidth := 2;
    app.printer.setFooter (rpf.footer);
    rpf.theDate.text  := today.String;
    rpf.customer.text := listCust.text;
    rpf.product.text  := listProd.text;
    rpf.employee.text := listEmp.text;
    foreach count in 1 to listFaults.listCount do
        rpf.detailLine.text := listFaults.itemText[count];
        result              := app.printer.print(rpf.detail);
        if result = Print_Stopped or result = Print_Cancelled then
            return;
        endif;
    endforeach;
epilog
    app.printer.close;
    delete rpf;
end;
```

For an example of a method that places output directly on to a printer page at a specified location on the page without using frames, see "Free-Format Printing", in the following subsection.

Free-Format Printing

JADE enables you to place output directly on a printer page at any location on the page without the use of frames and to draw rotated text and characters.

Although the **getPrintPosition** and **setPrintPosition** methods enable you to control the line position for your print output, printing graphics and multiple panes across the page when using print frames is not straightforward. (For details about using frames to define your report layouts, see "Defining Your JADE Report Layouts", earlier in this chapter.)

You can use the free-format print facilities to dynamically construct a page of output, treating the output page as whole canvas, or you can supplement the use of frames with these facilities.

Support for free-format printing:

- Provides flexibility in constructing print output
- Reduces complexity when constructing multiple-paned output
- Removes the reliance on the paint events of controls to be able to draw graphics
- Allows the output of non-horizontal left-to-right text

The **Printer** class graphics properties and methods, similar to those defined in the **Window** class, allow dynamic text and graphics to be output to a window.

The **Printer** class properties that you can use for free-format printing are summarized in the following table.

Property	Description
drawFillColor	Contains the color used to fill in shapes drawn with the printer graphics methods
drawFillStyle	Contains the pattern used to fill the shapes drawn using the printer graphics methods
drawFontBold	Used when constructing the font used for drawing text
drawFontItalic	Used when constructing the font used for drawing text.
drawFontName	Used when constructing the font used for drawing text
drawFontSize	Used when constructing the font used for drawing text
drawFontStrikethru	Used when constructing the font used for drawing text
drawFontUnderline	Used when constructing the font used for drawing text
drawStyle	Defines the line style for output from printer graphics methods
drawTextAlign	Contains the alignment used when outputting text on the printer using the **drawTextAt** and **drawTextIn** methods
drawTextCharRotation	Specifies the angle in degrees between each characters base line and the *x* axis of the device
drawTextRotation	Specifies the angle in degrees between the base line of the text output and the *x* axis of the page
drawWidth	Contains the line width for output from printer graphics methods

The **Printer** class methods that you can use for free-format printing are summarized in the following table.

Method	Description
drawArc	Draws an elliptical arc on the printer page
drawChord	Draws a chord on the printer page (that is, an arc with the end points joined and the interior filled)
drawEllipse	Draws an ellipse on the printed page
drawFilledRectangle	Draws a filled rectangle on the printed page
drawLine	Draws a line on the printed page
drawPie	Draws a pie-shaped wedge on the printed page
drawRectangle	Draws the border of a rectangle on the printed page

Method	Description
drawRoundRectangle	Draws a rectangle with rounded corners on the printed page
drawSolidRectangle	Draws a rectangle filled with the same color as the border on the printed page
drawTextAt	Draws a text string on the printer page
drawTextIn	Draws a text string with a bounded rectangle on the printer page
drawTextSize	Returns the size of the text on the print page, using the current **drawFont** property values
drawTextSizeIn	Returns the size of the text in a bounding rectangle on the print page, using the current **drawFont** property values

The method shown in the following example prints a calendar page for a specified month, by placing output directly on to a printer page at a specified location on the page.

```
printCalendarMonth(printer: Printer input; prettyPicture: Frame input;
                   month: Integer; year: Integer) updating;
vars
    x, y, xinc, yinc           : Integer;
    xstart, ystart, day, width : Integer;
    date                       : Date;
begin
    width := printer.pageWidth;
    // print pretty picture
    prettyPicture.left   := (width - prettyPicture.width)/2 + 30;
    printer.setPrintPosition(40);
    printer.drawFontName := "Arial";      // now print the month name
    printer.drawFontSize := 30;
    printer.drawTextAlign := printer.DrawTextAlign_Center;
    ystart := printer.getPrintPosition() + 10;
    date.setDate(1, month, year);
    printer.drawTextAt(date.monthName, (width/2).Integer, ystart, Black);
    // set next position after the month text
    x := printer.drawTextSize(date.monthName, y);
    ystart := ystart + y + 30;
    yinc   := ((printer.pageHeight - ystart - 50)/5).Integer;
    xinc   := ((width - 60)/ 7).Integer;
    xstart := ((width - xinc * 7)/2).Integer + 30;
    printer.drawFontSize := 12;
    date.setDate(7, 11, 1999);            // Sunday
    // draw squares
    foreach x in xstart to xstart + xinc*7 step xinc do
        printer.drawLine(x, ystart - 20, x, ystart + yinc*5, Black);
        if x < xstart + xinc * 7 then
            printer.drawTextAt(date.dayName, (x + xinc/2).Integer,
                               ystart - 20, Black);
            date := date + 1;
        endif;
    endforeach;
    foreach y in ystart to ystart + yinc*7 step yinc do
        printer.drawLine(xstart, y, xstart + xinc*7, y, Black);
    endforeach;
    date.setDate(1, month, year);         // print day numbers
```

```
    day := date.dayOfWeek mod 7;          // day number
    printer.drawTextAlign := Printer.DrawTextAlign_Left;
    printer.drawFontSize := 14;
    y := ystart;
    while month = date.month do
        printer.drawTextAt(date.day.String, xstart + day *
                           xinc + 6, y + 4, Black);
        date := date + 1;
        day := day + 1;
        if day = 7 then
            day := 0;
            y := y + yinc;
            if y >= ystart + yinc*5 then
                y := ystart;
            endif;
        endif;
    endwhile;
    printer.newPage;
    app.printer.close;
end;
```

Previewing Print Output

When the **Print Preview** option button is selected in the JADE development environment Print Options dialog or the appropriate option is selected in a JADE runtime application, the first page of your report is displayed on the workstation monitor when the **OK** button is clicked, instead of output to the printer.

Print output is directed to the preview file when the **Printer** class **printPreview** property is set to **true**.

As the print preview shows what your output will look like on a specific printer, the printer must be known before the preview is generated. (Paper sizes, printable regions, paper trays, and so on are not consistent across all printers.)

The **Print** button on the print preview does not necessarily send the output to your default printer; it is sent to the printer of the current print task (which defaults to your default printer).

The Title caption contains the current page that is being previewed and the total number of pages for your report; for example:

```
Preview of page 3 of 8 page report
```

Tip When you click on an area of a page previewed in a reduced form, the page is expanded with the selected area of the page centered on the workstation monitor. Click on an expanded page to reduce the previewed page.

You can page or scroll through the report, or you can select one of the buttons listed in the following table.

Button	Action
Previous Page	Displays the previous page of the report.
Next Page	Displays the next page of the report.
First Page	Displays the first page of the report.
Last Page	Displays the last page of the report.

Button	Action
Specific Page	Displays the Print Page Select dialog, to enable you to specify a valid number of the page that you want to preview.
Expand (or Reduce)	Zooms in to display the report across the width of the monitor. When the report is expanded, the button is captioned Reduce, to enable expanded output to be reduced or zoomed out.
Print Report	Directs the report to the default printer or the printer specified in the Print Setup dialog.
Print Selected	Displays the Select Pages To Print dialog (for details, see "Using the Select Pages To Print Dialog", in the following subsection).
Find	Displays the Find Text dialog (for details, see "Searching Previewed Output", later in this section).
Find Next	Finds the next occurrence of the text specified in the Find Text dialog.
Cancel	Cancels the print preview.

Buttons that are not valid are disabled. For example, if the report has only one page the **Next Page**, **Previous Page**, **First Page**, **Last Page**, and **Specific Page** buttons are disabled. If the first page of a multiple page report is displayed, the **Previous Page** button is disabled. For details about controlling the display of the **Print Report** and **Print Selected** buttons during print preview, see the **Printer** class **printPreviewAllowSelect** property, earlier in this section.

Note The preview output is held in transient objects on your workstation until you choose to print or cancel it.

When the presentation client requests a print preview, the pages of the printed report do not have to be transferred to and from the application server. (This optimizes the performance of the print preview process when running JADE thin client mode over a slow network.) However, if your application calls the **Printer** class **setReport** method to indicate that user logic subsequently stores or manipulates the report output, each page of output is transferred to the application server.

Using the Select Pages To Print Dialog

When print output is previewed, clicking the **Print Selected** button displays the Select Pages To Print dialog.

For details about previewing print output, see "Previewing Print Output", in the previous subsection. See also the **Printer** class **printPreviewAllowSelect** property, for details about controlling the display of the **Print Selected** button.

The Select Pages To Print dialog enables the user to select the printing of the current page, all pages, selected pages, or a range of pages of a print document-previewed in the JADE development environment or in a JADE application.

To specify print criteria

1. To print selected pages, perform one of the following actions.
   - Select the page or pages that you want to print in the **Pages Available** list box and then click the **>** button to copy the selected pages to the **Pages Selected** list box. (Use the Ctrl or Shift key to make or extend multiple selections.)

     The selected pages are then displayed in the **Pages Selected** list box and are highlighted in the **Pages Available** list box.
   - Specify the required page numbers in the **Selected Pages** text box and then click the **Selected Pages**

option button to confirm the selection. Specific page numbers or a range of pages can be specified, separated by a comma; for example, **2,3, 8-10**.

2. Select the **All Pages** option button to print all pages of the previewed document or the **Current Page** option button to print only the page currently displayed in the preview window.
3. In the **Copies** text box, specify the number of copies of the selected pages that are required. (By default, one copy only is printed.)
4. Click the **OK** button.

The selected pages are then output to the printer. Alternatively, click the **Cancel** button to abandon your selections.

Searching Previewed Output

When print output is previewed, clicking the **Find** button displays the Find Text dialog. For details about previewing print output, see "Previewing Print Output", in the previous subsection.

The Find Text dialog enables you to select the print output search criteria.

To specify search criteria

1. In the **Find Text** text box, specify the text that is to be located.
2. If you want the exact match by case (where uppercase or lowercase is significant), check the **Case Sensitive?** check box. A search is then performed for text with the same capitalization as the text in the **Find Text** text box. By default, searching is case-insensitive; that is, this check box is unchecked.
3. In the **Starting page number** text box, specify the page number on which to start the search if the search is to start on a page other than the current page. (The search is always performed in a forwards direction.)
4. If you want the search restricted to the full word specified in the **Find Text** text box (for example, **Adams** will not find **Adamson**), check the **Whole words only?** check box.

   A search is then performed for full words that match your specified search criterion. As this check box is not checked by default, the search will match on part of a word.
5. Click the **Find** button to initiate the search. Alternatively, click the **Cancel** button to abandon the search.

If the search finds an instance of the specified text, the page containing that text is displayed and positioned so that the text is visible, with the first occurrence of the located text displayed in red.

If the search does not locate the specified text, the Print Preview Find message box advises you of this.

If text has been located, the **Find Next** button is enabled, so that you can locate the next occurrence of the specified text. The Print Preview Find message box advises you when no more occurrences are located.

When searching for text on previewed print output, note the following points.

- When a search has been initiated and has located an occurrence, the **Find Next** button remains enabled, even after all occurrences have been processed.

  Changing focus to another page of the previewed print output and then clicking the **Find Next** button restarts the search from that page.
- The text is searched in lines across the output. The next field to be considered is to the right of the current line or the next field to the left at a greater vertical position.

This becomes important when searching for multiple words. A match of **quick fox** will succeed even if **quick** and **fox** are on different parts of the same line or on different lines if **fox** is the next logical text field on from **quick**.

- Printed output may be constructed so that **A1234** is actually made up of two fields: **A** and **1234**, for example. Using a part-word search of **A1234** or **A 1234** will both find the text in this example.

  However, a whole-word search of **A 1234** will only succeed because physically there are two separate words. If a whole-word search fails unexpectedly, try a part-word search. A warning to this effect is displayed on the failure message box for a whole-word search.

Portable Printing

JADE printing uses standard GUI objects and commands to generate the required output. You can save these objects and commands in a file for later use; that is, you can "replay" the file to generate a print preview or you can send the data to a printer.

JADE print data can be saved in the database in the following formats.

- Scalable Vector Graphics (SVG), which is the default value on all operating systems

  Note As the contents of **ActiveXControl**, **JadeDotNetVisualComponent**, **JadeRichText**, and **OleControl** controls are displayed by the Windows Graphical Device Interface (GDI) calls, they cannot generate SVG format files. When one of these controls is printed to an SVG meta file, a picture is therefore created from the controls that are currently displayed and it is this picture that is stored in the SVG file, which may result in a slightly lower-quality display. The resolution the generated image is specified by the **GeneratedImageResolution** parameter in the [JadePrinting] section of the JADE initialization file. In addition, if you perform a find operation from a print preview screen, JADE cannot search in these controls.

- Windows Enhanced Meta Files (EMF)

Data can be sent to the printer in the following print data formats.

- Windows Graphical Device Interface (GDI) commands, which is the default value.
- Postscript (PS). If using Postscript printing, your printer must support Postscript level 2 or greater.

Windows can provide the GUI objects and commands, which can be replayed to provide a print preview and data for printing. Although these GUI objects and commands are saved as an SVG file by default, you can force JADE to use an EMF format for print data.

The choice of meta file format (EMF or SVG) and print data type (GDI or PS) are controlled by the **PrintFileFormat** and **PrintDataType** parameters, respectively, in the [JadePrinting] section of the JADE initialization file.

Although output to a printer can be done using GDI or PS commands, the format of the meta file (that is, EMF or SVG) determines whether you can use the Postscript data type for print output. Not all options are available under all operating systems and executables. Valid combinations for the **jade.exe** executable are listed in the following table.

Print File Format	PS Data Format	GDI Data Format
SVG	Yes	Yes
EMF	No	Yes

The format of the data sent to the printer depends on the format in which the command data has been saved and the operating system and executable that is used. If the commands are saved as:

- **EMF**, the output can be **GDI** format only
- **SVG**, printer output can be **GDI** or **PS**

For more details, see "JADE Printing Section [JadePrinting]", in the *JADE Initialization File Reference*.

Process Class

An instance of the **Process** class is created for each sign-on to each application running in a JADE system. A node can have several processes when running the JADE development environment.

For details about the class constants, properties, and methods defined in the **Process** class, see "Process Class Constants", "Process Properties", and "Process Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

Process Class Constants

The constants provided by the **Process** class are listed in the following table.

Class Constant	Integer Value	Description
RPS_EXTRACT_FAILED_EVENT	202	Notification event that loading of RPS table data failed
RPS_EXTRACT_FINISHED_EVENT	203	Notification event that loading of RPS table data finished successfully
SignOn_Usage_NoAudit	2	Signed on in **NoAudit** mode
SignOn_Usage_OdbcLogin	3	Signed on over a JADE ODBC driver
SignOn_Usage_ReadOnly	1	Signed on in **ReadOnly** mode
SignOn_Usage_Update	0	Signed on in **Update** mode

Process Properties

The properties defined in the **Process** class are summarized in the following table.

Property	Description
adminInfo	Contains information that can be used in tools that monitor the status of processes in the system
node	Contains a read-only reference to the node in which the process is executing
number	Contains a read-only internal number that distinguishes the process from other concurrent processes
persistentApp	Contains a reference to the persistent application object that corresponds to the process
schema	Contains a reference to the persistent schema object that corresponds to the process
signOnTime	Contains the date and time that the process started executing
signOnUserCode	Contains the user code specified when signing on
status	Not yet implemented (reserved for future use)
type	Contains the type of the current process
userCode	Contains the current user code
userExitCode	Contains a value returned by your applications when the **jade.exe** program exits
userInfo	Contains any additional information stored by the user

adminInfo

Type: String

Availability: Read and write

The **adminInfo** property of the **Process** class contains administration information that can be used in tools that monitor the status of the processes in the system.

node

Type: Node

Availability: Read-only

The **node** property of the **Process** class contains a read-only reference to the node in which the process is executing.

number

Type: Integer

Availability: Read-only

The **number** property of the **Process** class contains a read-only **Integer** value that is an internal number, relative to the system, that distinguishes the process from other concurrent processes.

The value is zero (**0**) until the process has successfully passed the validation and initialization stages.

persistentApp

Type: Application

Availability: Read-only

The **persistentApp** property of the **Process** class contains a reference to the persistent application object that corresponds to the process.

The name of the application instance is that specified in any of:

- The **app** parameter in the JADE executable (**jade.exe**) command line
- The **Application** class **startApplication**, **startAppMethod**, or **startApplicationWithParameter** method
- The **jomSignOn** Application Programming Interface (API) call

schema

Type: Schema

Availability: Read-only

The **schema** property of the **Process** class contains a reference to the persistent schema object that corresponds to the process.

The name of the schema instance is that specified in any of:

- The **schema** parameter in the JADE executable (**jade.exe**) command line
- The **Application** class **startApplication**, **startAppMethod**, or **startApplicationWithParameter** method
- The **jomSignOn** API call

signOnTime

Type: TimeStamp

Availability: Read-only

The **signOnTime** property of the **Process** class contains the date and time that the process started its execution. The date and time is that of the node hosting the process for all processes apart from thin client processes, for which the date and time is obtained from the thin client.

signOnUserCode

Type: String[30]

Availability: Read-only

The **signOnUserCode** property of the **Process** class contains the user code specified when signing on to the application.

The following example shows the use of the **signOnUserCode** property.

```
load() updating;
begin
    self.centreWindow;
    self.caption        := process.signOnUserCode;
    connectionName.text := app.computerName;
    sendIt.value        := true;
end;
```

status

Type: Integer

Availability: Read-only

The **status** property of the **Process** class is not yet implemented. It is reserved for future use.

type

Type: ASCII character[1]

Availability: Read-only

The **type** property of the **Process** class contains the type of current process, as shown in the example in the following code fragment.

```
if process.type = 5.Character then...           // do some processing here
```

The process types are listed in the following table.

Value	Description
0	Background process
1	JADE development (that is, JADE itself)
2	A non-development process
3	JADE Debugger
4	JADE Painter or JADE Translator utility
5	Shadow process (that is, a process replaying transactions on a secondary server)
6	A JADE utility application, such as JADE Monitor or the JADE Database Utility

As all processes running on the secondary server in an SDS environment share the same node object, you can use the **Process** class **type** property to distinguish shadow (replaying) transactions on a secondary server from reader processes.

userCode

Type: String[30]

Availability: Read-only

The **userCode** property of the **Process** class contains the current user code, which may differ from that specified when signing on to the application.

Changes to the **userCode** property are audited in a journal record of type **Jaa_Type_ChangeUser**. Information from this record can be retrieved by using the **getChangeUserData** method of the **JadeAuditAccess** class.

The following example shows the use of the **userCode** property.

```
vars
    pd   : ProcessDict;
    proc : Process;
    coy  : Company;
begin
    coy := Company.firstInstance;
    self.lock(coy, 1, 1, 100);
    create pd transient;
    system.getObjectLockProcesses(coy, pd, 100);
    foreach proc in pd do
        write(proc.userCode);
    endforeach;
end;
```

userExitCode

Type: Integer

The **userExitCode** property of the **Process** class a contains a value returned by your applications when the **jade.exe** program exits. The default value is zero (**0**). For more details, see Appendix A, "Exit Values", in the *JADE Installation and Configuration Guide*.

You can use this property, for example, to set a non-zero exit code that can then be checked in a batch file by using the **ERRORLEVEL** keyword to check for appropriate **userExitCode** values, as shown in the following example.

```
begin
    beginTransaction;
    process.userExitCode := 123;
    commitTransaction;
    terminate;
end;
```

The specified value is returned only if the JADE program would have normally returned zero (**0**); that is, if JADE wants to return a non-zero exit value, the JADE value takes precedence over your value specified in this attribute.

Note As the **userExitCode** property applies to the **jade exe** node, any JADE application executing from that same client environment can set this value. Cooperation between applications wanting to set this property may therefore be required.

This property gets or sets the exit code for the running **jade exe** client. The property can be accessed only on the running process. Accessing it on a non-**jade exe** client or a process other than the current process of the application results in exception 1265 *(Environmental object operation is out of scope for process)*.

If the client is running as a standard client, the method is equivalent to **node.userExitCode** for the current processes node.

If the client is running as a presentation client, the exit code is retrieved or set on the **jade.exe** program of the presentation client.

Note As the process object is persistent, you must be in transaction state to set the value.

userInfo

Type: String

The **userInfo** property of the **Process** class contains any additional information that the user needs to store for a process. For example, you can display user data in the JADE Monitor Users window, if required, by executing the following code.

```
beginTransaction;
    process.userInfo := "specify-the-user-supplied-text-here";
commitTransaction;
```

The following examples show the use of the **userInfo** property.

```
startSvrApp4() updating;
vars
    timestamp : TimeStamp;
begin
    beginTransaction;
    process.userInfo := app.name & ' ' & method.name;
    commitTransaction;
    beginTimer(30000, Timer_OneShot, Start_Exe);
end;

beginTransaction;
```

```
        process.userInfo := 'Sort started at ' & ts.String;
    commitTransaction;
```

Process Methods

The methods defined in the **Process** class are summarized in the following table.

Method	Description
addLockCallStackFilter	Restricts the saving of the lock call stack to instances of the specified class or classes only
adjustObjectCachePriority	Changes how long an object is to be retained in persistent or transient object cache
allTransientInstances	Populates the specified array with all non-shared transient instances that have been created by the receiver process and not yet deleted
allowTransientToPersistentInvs	Enables a transient object to reference a persistent object without its inverse being maintained
allowTransientToSharedTranInvs	Enables a non-shared transient object to reference a shared transient object without its inverse being maintained
analyzeTransientFileUsage	Returns a string containing a detailed analysis of the transient database file
appServerPort	Returns the TCP/IP communications port number of the application server node
beginMethodProfiling	Starts method profiling for the receiving process
changeUserCode	Changes the current value of the **userCode** property
classAccessFrequenciesStatus	Returns **true** if at least one process has enabled the counting of accesses to classes
clearLockCallStackFilter	Clears the list of classes for which saving lock call stack information has been enabled for this process
compactTransientFile	Defragments the transient database file for the receiving process
countQueuedNotifications	Returns the number of unprocessed notifications queued for the calling process
createTransientMethod	Creates an executable transient JADE method
currentStack	Populates the process stack array with references to method call descriptor objects
debug	Displays information about your current process stack, and enables you to inspect variables
deleteTransientMethod	Deletes a transient JADE method
disableAllTransTraceCallbacks	Disables all transaction trace callbacks for the receiver process
enableClassAccessFrequencies	Enables or disables the counting of accesses to classes
enableTransTraceCallback	Enables or disables a specified transaction trace callback for the receiver process
endMethodProfiling	Stops method profiling for the receiving process

Method	Description
executeIOScript	Executes a JADE script passing parameters as **io** (input-output) usage
executeScript	Executes a JADE script
executeTransientIOMethod	Executes a transient JADE method passing parameters as **io** (input-output) usage
executeTransientMethod	Executes a transient JADE method
extractRequestStatistics	Extracts local or remote request statistics from notifications sent in response to a **sendRequestStatistics** method request
extractWebStatistics	Extracts the performance statistics relating to Web activity from a notification
finalizePackages	Performs any terminate function common to all schema applications that contain packages
getAllApps	Populates an array with all applications that are active in the process of the receiver
getBufferStatistics	Returns cache-related information about a specified object
getCallStackInfo	Retrieves information about the call stack of the current process
getCommandLine	Returns a string containing the command line of the receiving node object of the process
getComputerName	Returns a string containing the computer name of the receiving node object of the process
getDateTimeDelta	Retrieves the values used to adjust initial date and time local variable values used by the receiving process
getErrorText	Returns the message text for a JADE-defined error code
getExceptionHandlerStack	Populates an array with transient objects representing exception handlers armed by the current process
getIniFileName	Returns a string containing the name and full path of the JADE initialization file of the process
getJadeHomeDirectory	Returns a string containing the JADE HOME directory
getJadeInstallDirectory	Returns a string containing the directory in which the JADE binaries are installed
getJadeWorkDirectory	Returns a string containing the directory in which JADE work files are created
getLastExtFunctionCallError	Returns the error code set by the last external function call made by the current process
getLockCallStackFilter	Returns the list of classes for which saving lock call stack information has been enabled for this process
getMethodCacheLimit	Retrieves the method cache limit for the executing process
getMethodCacheStatistics	Retrieves information about the method cache of the executing process and stores it in the passed-in **JadeDynamicObject** instance
getMethodProfileInfo	Retrieves method profiling information for the receiving process

Method	Description
getOSDetails	Returns comprehensive information about the operating system and machine architecture of the process of the receiver
getOSPlatform	Returns the operating system of the process of the receiver
getPersistentDeadlockPriority	Retrieves the priority value to be used when dealing with deadlocks involving persistent objects
getProcessApp	Returns a reference to the main **Application** object of the current process
getProfileString	Retrieves a string from the specified section in the initialization file of the process
getProgramDataDirectory	Returns a string containing the program data directory
getRequestStatistics	Retrieves node sampling values relating to the receiver process
getRpcServerStatistics	Retrieves node sampling statistics relating to RPC activity between the database server node and the receiver process
getSaveLockCallStack	When an object is locked, returns **true** if the lock call stack is being saved for a process
getSignOnUsage	Returns the way in which a currently logged on user signed on to JADE
getStringPoolLimit	Retrieves the string pool limit for the executing process
getTempPath	Returns a string containing the architecture-specific version of the directory in which temporary files are created on the process of the receiver object
getTimers	Returns timer-related information from the receiving process
getTrackedMethod	Returns the tracked method that caused the specified preamble or postamble method to be invoked
getTrackedMethodReceiver	Returns the object used as the receiver for the method being tracked
getTrackedMethodReturnValue	Retrieves the return value of the method being tracked
getTrackedMethodStatus	Returns a value representing the current status of the tracked method
getTransactionId	Returns the latest identifier from the most recent transaction as a **Decimal** value
getTransactionId64	Returns the latest identifier from the most recent transaction as an **Integer64** value
getTransactionTraceCallbacks	Returns the method and receiver for all currently enabled transaction trace callbacks
getTransactionTraceObject	Returns the transaction trace object associated with the current process
getTransientDeadlockPriority	Retrieves the priority value to be used when dealing with deadlocks involving shared transient objects
getTransientFileLength	Returns the physical size of the transient database file in use by the executing process
getTransientFileName	Returns the name of the transient database file in use by the executing process
getUserDataDirectory	Returns a string containing the user data directory

Method	Description
initializePackages	Performs any initialization function common to all schema applications that contain packages
isCommitting	Returns **true** if the process is currently committing a transaction
isInExceptionState	Returns **true** if the epilog is being executed as a result of an exception
isInImportedContext	Returns **true** if the current (executing) process has invoked the current method from a package
isInLoadState	Returns **true** if the process is currently in load state
isInLockState	Returns **true** if the process is currently in lock state
isInTransactionState	Returns **true** if the process is currently in persistent transaction state
isInTransientTransactionState	Returns **true** if the process is currently in transient transaction state
isRunningScript	Returns **true** if the process is running a **JadeScript** or Workspace method
isUserDataPump	Returns **true** if the process is running as a user-defined **Datapump** application
isUsingThinClient	Returns **true** if the process is running in JADE thin client mode
iteratorsExcludeOfflineObjects	Specifies whether objects stored in offline partitions should be excluded when collections are iterated with an iterator or a foreach instruction
networkAddress	Returns the IP address of the network interface connection to the application server or database server
profileMethod	Selects or deselects a method to be profiled for the receiving process
profiler	Returns the profiler for the receiving process
prohibitBeginTransaction	Stops the current process entering transaction state
prohibitPersistentUpdates	Enables the updating of persistent objects in the current process to be prohibited
removeMethodProfileInfo	Removes all method profiling information for the receiving process
resumeTimers	Resumes all timers suspended for a process
rpsSuppressTransactionDeletes	Specifies that an object deletion on a primary system is not replicated to a relational database
sendCallStackInfo	Requests a process to send notifications containing information about its call stack
sendMethodCacheStatistics	Requests a target process (the receiver) to send a notification containing statistics about the method cache of the target process
sendRequestStatistics	Requests a process to send a notification containing local or remote request statistics
sendTransientFileAnalysis	Requests a process to send notifications containing a detailed analysis of the contents of the transient database file
sendTransientFileInfo	Requests a process to send a notification containing the oid of the process, and the name and length of the transient database file

Method	Description
sendWebStatistics	Requests a process to send a notification containing performance statistics for Web activity
setDateTimeDelta	Sets the values used to adjust the initial values of the **Date**, **Time**, and **TimeStamp** local variables
setDefaultLockTimeout	Changes the default lock timeout period for the receiving process
setMethodCacheLimit	Programmatically sets the method cache limit for the executing process
setObjectCachePriority	Specifies how long an object is to be retained in persistent or transient object cache
setPersistentDeadlockPriority	Sets the priority value to be used when dealing with deadlocks involving persistent objects
setProfileString	Copies a string into the specified section of the JADE initialization file of the process
setSaveLockCallStack	Specifies whether the call stack is recorded when the receiving process locks an object
setStringPoolLimit	Programmatically sets the string pool limit for the executing process
setTransientDeadlockPriority	Sets the priority value to be used when dealing with deadlocks involving shared transient objects
sleep	Suspends execution of the thread of the receiver process for a specified time
startMethodTracking	Initiates method tracking for the receiver process
startTransactionTrace	Initiates transaction tracing for transactions carried out by the current process
stopMethodTracking	Turns off method tracking for the receiver process
stopTransactionTrace	Turns off transaction tracing for the receiver process
suspendTimers	Suspends all timers registered by a process
transactionTraceStarted	Returns **true** if transaction tracing is enabled for the current process
transientPersistentInvsEnabled	Returns the current state of the Boolean set by calls to **allowTransientToPersistentInvs** on the process
transientSharedTranInvsEnabled	Returns the current state of the Boolean set by calls to **allowTransientToSharedTranInvs** on the process
truncateOnDecimalOverflow	Specifies whether an exception is raised when a decimal overflow occurs
useUpdateLocks	**Update** locks rather than **Exclusive** locks are implicitly acquired when an object is updated
waitForMethods	Suspends the process until one of the method contexts completes or times out

addLockCallStackFilter

Signature `addLockCallStackFilter(classList: ParamListType);`

The **addLockCallStackFilter** method of the **Process** class enables you to define a filter to restrict the saving of lock call stack information to instances of specific classes only. By default, the lock call stack is saved for all objects that the process locks.

You can call this method for any running **Process** instance in the system.

The code fragment in the following example shows the use of the **addLockCallStackFilter** method.

```
process.addLockCallStackFilter(Customer, Account, SaleItem);
```

adjustObjectCachePriority

Signature
```
adjustObjectCachePriority(obj:   Object;
                          delta: Integer): Boolean;
```

The **adjustObjectCachePriority** method of the **Process** class changes, through the **delta** parameter, how long an object, specified by the **obj** parameter, is to be retained in object cache. If the object is in cache, **true** is returned to indicate that the retention of the object in cache has been changed. If the object is not in cache or is being updated by another process, **false** is returned.

The value of the **delta** parameter effectively changes the number of *lives* in cache of the object specified by the **obj** parameter. The number of lives for an object in cache is one (**1**) through **255**. A positive value for the **delta** parameter increases the number of lives up to the upper limit of **255**, and a negative value decreases the number of lives to the lower limit of zero (**0**) lives, at which point the object is removed from cache.

When an object in cache is not used for the specified amount of time, it becomes a candidate to be removed from cache. Its number of lives is examined. If it is equal to one (**1**), the object is removed from cache. If it is greater than one (**1**), the number of lives is decremented and instead of being removed from cache, the object is treated as if it had just been accessed. This results in it being retained longer in cache, instead of being removed. Conversely, when the number of lives for an object is set to zero (**0**), it is removed from cache.

When an object is removed, its subobjects are also removed, including string large objects (slobs) and binary large objects (blobs) but not *exclusive* collections, which must be removed separately.

The **adjustObjectCachePriority** method is a variation of the **setObjectCachePriority** method that enables you to adjust the number of lives relative to the current value, rather than specify the exact number of lives.

The number of lives an object has applies only while the object is in cache. When an object is first loaded into cache, it is assigned one life only. Lives are not recorded for objects that are not in cache.

You can use the **adjustObjectCachePriority** method with persistent and transient objects; that is, it applies to persistent and transient object caches. With transient objects, a process can affect only shared transient objects and its own non-shared transient objects.

A process must use its own **Process** instance as the method receiver. Using any other **Process** instance causes a 1265 exception (*Environmental object operation is out of scope for process*) to be raised.

In the following method, an application decrements the number of lives each object in a collection; that is, reduces the number of lives by one (**1**) on an individual basis.

```
foreach cust in root.allCusts do
    totalSales := totalSales + cust.purchases;
    process.adjustObjectCachePriority(cust, -1);
endforeach;
```

allowTransientToPersistentInvs

Signature `allowTransientToPersistentInvs(allow: Boolean);`

The **allowTransientToPersistentInvs** method of the **Process** class enables a transient object to reference a persistent object without its inverse being maintained. Calling this method with the **allow** parameter set to **true** is equivalent to enabling the **Allow Transient to Persistent Reference** check box on the extended Define References dialog for all references until this method is called with the **allow** parameter set to **false**.

For more details, see "Defining an Inverse Reference Property", in Chapter 4 of the *JADE Development Environment User's Guide*.

The initial state of a process is to disallow such references (that is, as if this method had been called with a value of **false**) and attempts to do so raise a **1215** exception (that is, *Persistent objects cannot reference transient objects*).

Use the **transientPersistentInvsEnabled** method to return the current state of the process.

allowTransientToSharedTranInvs

Signature `allowTransientToSharedTranInvs(allow: Boolean);`

The **allowTransientToSharedTranInvs** method of the **Process** class enables a non-shared transient object to reference a shared transient object without its inverse being maintained.

The initial state of a process is to disallow such references (that is, as if this method had been called with a value of **false**) and attempts to do so raise a **1289** exception (that is, *Shared transient objects cannot reference non-shared transient objects*).

Use the **transientSharedTranInvsEnabled** method to return the current state of the process.

allTransientInstances

Signature
```
allTransientInstances(objArray: ObjectArray;
                      maxInsts: Integer);
```

The **allTransientInstances** method of the **Process** class populates the array specified in the **objArray** parameter with all non-shared transient instances that have been created by the receiver process and not yet deleted. This includes instances of internal classes such as **NumberFormat** and **Printer** but does not include shared transient instances. Use this method to check for transient objects that have not been deleted when an application terminates.

The **maxInsts** parameter specifies the maximum number of transient instances. A **maxInsts** parameter value of zero (**0**) indicates that there is no maximum number of transient instances.

You can use the **allTransientInstances** method only a **Process** instance passed by the current process. If this method is called with a foreign process as the receiver, an exception is raised (that is, a 1265 - *Environmental object is out of scope for process*).

Notes Do not use the **allTransientInstances** method in **serverExecution** methods or in **clientExecution** methods called from **serverExecution** methods. When executed from a **serverExecution** method, the only instances included in the array will be transient objects that have been created or updated by the **serverExecution** method.

If executed from a **clientExecution** method called from a **serverExecution** method, it will not include any transient objects that were created in the **serverExecution** method and not yet accessed by the **clientExecution** methods.

The code fragment in the following example shows the use of the **allTransientInstances** method.

```
create objectArray transient;
process.allTransientInstances(objectArray, 0);
write 'Transient instances are -';
foreach object in objectArray do
    write object.String;
endforeach;
delete objectArray;
```

analyzeTransientFileUsage

Signature `analyzeTransientFileUsage(): String;`

The **analyzeTransientFileUsage** method of the **Process** class returns a string containing a detailed analysis of the transient database file, including counts of objects by class number plus other useful information. Each line of the analysis is delimited by the line feed (**Lf**) character.

Tip An easy way to view this output is to use the **writeString** method of the **File** class to write the string returned by the **analyzeFileTransientUsage** method to a file and then view it with WordPad or another text editor.

See also "Transient Database File Analysis", in Chapter 3 of the *JADE Database Administration Guide*.

appServerPort

Signature `appServerPort(): Integer;`

The **appServerPort** method of the **Process** class returns the TCP/IP communications port number of the application server node on which the process is executing. This method does *not* return the port number of the receiver of the method.

When running in single user mode or the application is not running in JADE thin client mode, the **appServerPort** method returns zero (**0**).

beginMethodProfiling

Signature `beginMethodProfiling(option: Integer);`

The **beginMethodProfiling** method of the **Process** class starts method profiling for the receiving **Process** instance, which can be any current process including processes running on other nodes.

The values for the **option** parameter and the corresponding range of methods to be profiled are listed in the following table.

Value	Profiles …
1	All called methods, whether nominated or not
2	Nominated methods and their nested method calls
3	Nominated methods only

Methods are nominated using the **profileMethod** method of the **Process** class.

The following actions occur if the **beginMethodProfiling** method is called when profiling is already in effect for the target process.

1. Profiling information is reset
2. The profiling option is adjusted to match the **option** parameter
3. The list of nominated methods is retained

Notes One process can start method profiling on a target process, and a different process can clear or end the profiling.

It is recommended that when investigating application performance, only one of the JADE Profiler, JADE Monitor, or method profiling is used at any one time, as the results reported when any of these are combined is undefined.

Methods specified as **serverExecution** are not profiled, unless executed from server applications or in single user mode.

changeUserCode

Signature `changeUserCode(userCode: String);`

The **changeUserCode** method of the **Process** class changes the current value of the **userCode** property. The code fragment in the following example shows the use of the **changeUserCode** method.

```
process.changeUserCode("newUserCode");
```

Use the **userCode** parameter to specify the new value of the **userCode** property for the process.

classAccessFrequenciesStatus

Signature
```
classAccessFrequenciesStatus(processList: ProcessDict input;
                             startTime:   TimeStamp output): Boolean;
```

The **classAccessFrequenciesStatus** method of the **Process** class returns **true** if at least one process has enabled the counting of accesses to classes by using the **enableClassAccessFrequencies** method.

The **processList** parameter is populated with the object identifiers (oids) of the processes that enabled the counting of accesses to classes. The value of the **startTime** parameter is set to the time that the *first* process enabled the counting of accesses.

clearLockCallStackFilter

Signature `clearLockCallStackFilter();`

The **clearLockCallStackFilter** method of the **Process** class clears the list of classes for which saving lock call stack information has been enabled for this process; that is, it clears the filtering options set for this process by the **addLockCallStackFilter** method of the **Process** class.

You can call this method for any running **Process** instance in the system.

compactTransientFile

Signature `compactTransientFile();`

The **compactTransientFile** method of the **Process** class defragments the transient database file associated with the current process.

countQueuedNotifications

Signature `countQueuedNotifications(): Integer;`

The **countQueuedNotifications** method of the **Process** class returns the number of unprocessed notifications queued for the calling process.

You can call this method only on the process instance of the current process. An exception is raised if you call it on an instance of another process.

createTransientMethod

Signature
```
createTransientMethod(methodName:        String;
                      schemaType:        Type;
                      schema:            Schema;
                      sourceCode:        String;
                      isWorkspaceMethod: Boolean;
                      returnType:        Type;
                      errorCode:         Integer output;
                      errorPosition:     Integer output;
                      errorLength:       Integer output): JadeMethod;
```

The **createTransientMethod** method of the **Process** class creates and compiles a transient JADE method and returns a JADE method that can be subsequently executed by using the **Process** class **executeTransientMethod** or **executeTransientIOMethod** method.

The **createTransientMethod** method parameters are listed in the following table.

Parameter	Description
methodName	The name of the method to be created. For non-Workspace methods, this parameter must match the method name specified in the signature in the source code.
schemaType	The owner type, or owner root type, of the method (that is, the type of the method receiver) and it is the type of the **self** system variable.
schema	The schema against which the method is to be compiled and which is searched when resolving the names of classes referenced in the method.
sourceCode	The method source code to be compiled.
isWorkspaceMethod	Specifies whether the source code is in JADE Workspace format (that is, it has no method signature). JADE Workspace format transient methods cannot update the receiver directly.
returnType	Specifies the type of the result value returned by Workspace methods. For non-Workspace methods, this parameter is not used and the return type is specified in the method signature in the source code.

Parameter	Description
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the method compiled successfully.
errorPosition	The position of the error in the source code. Note that the first character of the source code has a position of zero (**0**).
errorLength	The length in characters of the error in the source code.

If the method returns a null value, the error parameters return compiler information that indicates the cause of the error. For an example of the use of the **createTransientMethod** method, see the **Process** class **executeTransientMethod** method.

Use the **Process** class **deleteTransientMethod** method to delete the transient method.

currentStack

Signature `currentStack(procStack: ProcessStackArray);`

The **currentStack** method of the **Process** class populates the process stack array specified in the **procStack** parameter with references to method call descriptor objects. The process stack array represents a snapshot of the current execution history of the application thread of the current process. For more details, see "MethodCallDesc Class".

An exception is raised if an attempt is made to call this method for a process other than the current process.

Note As this method creates transient instances of the **MethodCallDesc** class, it is the responsibility of the method caller to purge the collection used by the method to delete these transient instances. The collection should be purged *before* the deletion of the process stack array passed to the method in the **procStack** parameter.

The following example shows an **Exception** method called from your exception handler.

```
getMethodCallers();
vars
    callStack    : ProcessStackArray;
    methCallDesc : MethodCallDesc;
begin
    create callStack;
    // get the stack for the current process
    process.currentStack(callStack);
    // iterate through the process stack array and display each
    // MethodCallDesc
    foreach methCallDesc in callStack do
        write methCallDesc.display;
    endforeach;
epilog
    // before finishing, delete the transient MethodCallDesc
    // objects created by the currentStack method
    callStack.purge;
    delete callStack;
end;
```

debug

Signature `debug();`

The **debug** method of the **Process** class displays a modal window containing your current stack and the source of your current method, with the current line highlighted. Use this window to display the contents of variables, if required.

An exception is raised if this method is invoked from a server method.

deleteTransientMethod

Signature `deleteTransientMethod(meth: JadeMethod io);`

The **deleteTransientMethod** method of the **Process** class deletes the transient method specified in the **meth** parameter. This method must be used to delete a transient method that was created by using the **Process** class **createTransientMethod** method.

For an example of the use of the **deleteTransientMethod** method, see the **Process** class **executeTransientMethod** method.

disableAllTransTraceCallbacks

Signature `disableAllTransTraceCallbacks();`

The **disableAllTransTraceCallbacks** method of the **Process** class unregisters all transaction trace callbacks for the receiver, which must be the current process.

If you want to unregister a specific callback, use the **enableTransTraceCallback** method specifying the method, receiver, and passing the value of **false** to the **enable** parameter.

The following code fragment specifies that when a transaction for the current process commits, a method **AnyClass** class **commitCallback** is no longer to be called for the receiver **inst**, which is of type **AnyClass**.

```
process.enableTransTraceCallback(AnyClass::commitCallback, inst, false);
```

The following code fragment specifies that when a transaction for the current process commits, no callbacks are to be made.

```
process.disableAllTransTraceCallbacks();
```

enableClassAccessFrequencies

Signature `enableClassAccessFrequencies(enable: Boolean): Boolean;`

The **enableClassAccessFrequencies** method of the **Process** class enables or disables the counting of accesses to classes depending on the value of the **enable** parameter. If the **enable** parameter is **true**, counting of accesses is enabled.

The method receiver can be any current process, including the requesting process itself or a process executing on another node.

The **enableClassAccessFrequencies** method returns **true** if the counting of accesses is already enabled for the current process and **false** otherwise.

Enabling counting of accesses where it is already enabled for the current process is ignored. Similarly, disabling counting of accesses where it is not enabled for the current process is ignored.

The **getClassAccessFrequencies** method of the **System** class returns access counts for the specified classes provided counting of class accesses is enabled.

enableTransTraceCallback

Signature `enableTransTraceCallback(callbackMethod: Method; callbackReceiver: Object; enable: Boolean);`

The **enableTransTraceCallback** method of the **Process** class registers or unregisters a specified method and receiver for callback just prior to a transaction for the receiving process being committed. The receiving **Process** instance for the **enableTransTraceCallback** method must be the current process.

The **callbackMethod** parameter specifies the method to be invoked. The **callbackReceiver** parameter specifies the object to be used as the receiver of the invoked method. If the **enable** parameter is set to **true**, the callback is registered; if it is **false**, the callback is unregistered.

The callback method must have no parameters and no return type. It is invoked only if transaction tracing has been activated for the process by calling the **startTransactionTrace** method.

The **enableTransTraceCallback** method can be called multiple times to register additional method callbacks when a transaction commits. Calling the **enableTransTraceCallback** with a method and receiver combination that has been previously registered is ignored.

Methods are invoked in reverse order of when they were registered; that is, the most recently registered are invoked first.

Notes The invoked method should not attempt to commit the transaction. Doing so causes repeated invocations of the method leading eventually to a kernel stack overflow.

Similarly, the invoked method should not abort the current transaction. Doing so raises an exception after the method returns and an attempt to commit the transaction is made.

If an exception occurs within an invoked method and is not dealt with by an exception handler, the transaction is not committed.

The following code fragment specifies that when a transaction for the current process commits, a method **AnyClass** class **commitCallback** is to be called for the receiver **inst**, which is of type **AnyClass**.

```
process.enableTransTraceCallback(AnyClass::commitCallback, inst, true);
```

endMethodProfiling

Signature `endMethodProfiling();`

The **endMethodProfiling** method of the **Process** class stops method profiling for the target process used as the method receiver.

The profiling information is retained until the **removeMethodProfileInfo** method is called or the target process terminates.

The **endMethodProfiling** method has no effect if it is called when profiling is not in effect for the target process.

Note One process can start method profiling on a target process, and a different process can clear or end the profiling.

---

executeIOScript

Signature

```
executeIOScript(methodName:         String;
                schemaType:         Type;
                schema:             Schema;
                sourceCode:         String;
                isWorkspaceMethod:  Boolean;
                returnType:         Type;
                errorCode:          Integer output;
                errorPosition:      Integer output;
                errorLength:        Integer output;
                receiver:           Any io;
                params:             ParamListType io): Any;
```

The **executeIOScript** method of the **Process** class executes a JADE script passing parameters as **io** (input-output) and provides a wrapper method for calling the **createTransientMethod**, **executeTransientIOMethod**, and **deleteTransientMethod** methods to compile and execute JADE method source code.

This method returns the result value returned by the executed method. The method is executed as part of the current JADE process and any references to system variables (for example, **app**) reference those for the application that is currently running.

The **executeIOScript** method parameters are listed in the following table.

Parameter	Description
methodName	The name of the method to be created. For non-Workspace methods, this parameter must match the method name specified in the signature in the source code.
schemaType	The owner type, or owner root type, of the method (that is, the type of the method receiver) and it is the type of the **self** system variable.
schema	The schema against which the method is to be compiled and which is searched when resolving the names of classes referenced in the method.
sourceCode	The method source code to be compiled.
isWorkspaceMethod	Specifies whether the source code is in JADE Workspace format (that is, it has no method signature). JADE Workspace format transient methods cannot update the receiver directly.
returnType	Specifies the type of the result value returned by Workspace methods. For non-Workspace methods, this parameter is not used and the return type is specified in the method signature in the source code.
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the method was compiled successfully.
errorPosition	The position of the error in the source code. Note that the first character of the source code has a position of zero (**0**).
errorLength	The length in characters of the error in the source code.
receiver	The receiving object or primitive type, defined as an **io** parameter to allow it to be updated by an updating method specified in the **methodName** parameter.

Parameter	Description
params	Maps to a variable list of zero or more parameters of any type that are to be passed to the method that is executed. This parameter is defined as **io** to allow the passing and updating of parameters defined as **io** in the signature of the method specified in the **methodName** parameter. (The **ParamListType** pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of **ParamListType**.)

For details about the **ParamListType** pseudo type specified in the last formal parameter (**params**), see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*. If the source code does not compile successfully, the error parameters return compiler information that indicates the cause of the error.

executeScript

Signature

```
executeScript(methodName:         String;
              schemaType:         Type;
              schema:             Schema;
              sourceCode:         String;
              isWorkspaceMethod:  Boolean;
              returnType:         Type;
              errorCode:          Integer output;
              errorPosition:      Integer output;
              errorLength:        Integer output;
              receiver:           Any;
              params:             ParamListType): Any;
```

The **executeScript** method of the **Process** class executes a JADE script and provides a wrapper method for calling the **createTransientMethod**, **executeTransientMethod**, and **deleteTransientMethod** methods to compile and execute JADE method source code.

This method returns the result value returned by the executed method.

The method is executed as part of the current JADE process and any references to system variables (for example, **app**) reference those for the application that is currently running.

The **executeScript** method parameters are listed in the following table.

Parameter	Description
methodName	The name of the method to be created. For non-Workspace methods, this parameter must match the method name specified in the signature in the source code.
schemaType	The owner type, or owner root type, of the method (that is, the type of the method receiver) and it is the type of the **self** system variable.
schema	The schema against which the method is to be compiled and which is searched when resolving the names of classes referenced in the method.
sourceCode	The method source code to be compiled.
isWorkspaceMethod	Specifies whether the source code is in JADE Workspace format (that is, it has no method signature). JADE Workspace format transient methods cannot update the receiver directly.

Parameter	Description
returnType	Specifies the type of the result value returned by Workspace methods. For non-Workspace methods, this parameter is not used and the return type is specified in the method signature in the source code.
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the method was compiled successfully.
errorPosition	The position of the error in the source code. Note that the first character of the source code has a position of zero (**0**).
errorLength	The length in characters of the error in the source code.
receiver	The receiving object or primitive type.
params	Maps to a variable list of zero or more parameters of any type that are to be passed to the method that is executed. (The **ParamListType** pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of **ParamListType**.)

For details about the **ParamListType** pseudo type specified in the last formal parameter (**params**), see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

If the source code does not compile successfully, the error parameters return compiler information that indicates the cause of the error. The following example shows the use of the **executeScript** method to perform a calculation entered by the user. (For example, an input of **'2*3+1'** would display the value **'7'**.)

```
vars
    input, str    : String;
    err, pos, len : Integer;
    result        : Any;
begin
    read input;
    str    := "return " & input & ";";
    result := process.executeScript("calc", self.class, currentSchema,
                                    str, true, Any, err, pos, len, self);
    if err = 0 then
        write result;
    else
        write "Compiler error " & err.String & " - " &
              process.getErrorText(err);
    endif;
end;
```

executeTransientIOMethod

Signature

```
executeTransientIOMethod(meth:     JadeMethod;
                         receiver: Any io;
                         params:   ParamListType io): Any;
```

The **executeTransientIOMethod** method of the **Process** class executes the transient JADE method specified in the **meth** parameter. This method returns the result value returned by the executed method. The method that is executed must have been created by using the **Process** class **createTransientMethod** method.

The method is executed as part of the current JADE process and any references to system variables (for example, **app**) reference those for the application that is currently running.

The **executeTransientIOMethod** method parameters are listed in the following table.

Parameter	Description
meth	The transient method to be executed.
receiver	The receiving object or primitive type, defined as an **io** parameter to allow it to be updated by an updating method specified in the **meth** parameter.
params	Maps to a variable list of zero or more parameters of any type that are to be passed to the method that is executed. This parameter is defined as **io** to allow the passing and updating of parameters defined as **io** in the signature of the method specified in the **meth** parameter. (The **ParamListType** pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of **ParamListType**.)

For details about the **ParamListType** pseudo type specified in the last formal parameter (**params**), see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

executeTransientMethod

Signature

```
executeTransientMethod(meth:     JadeMethod;
                       receiver: Any;
                       params:   ParamListType): Any;
```

The **executeTransientMethod** method of the **Process** class executes the transient JADE method specified in the **meth** parameter. This method returns the result value returned by the executed method. The method is executed as part of the current JADE process and any references to system variables (for example, **app**) reference those for the application that is currently running.

The method that is executed must have been created by using the **Process** class **createTransientMethod** method.

The **executeTransientMethod** method parameters are listed in the following table.

Parameter	Description
meth	The transient method to be executed.
receiver	The receiving object.
params	Maps to a variable list of zero or more parameters of any type that are to be passed to the method that is executed. (The **ParamListType** pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of **ParamListType**.)

The following example shows the use of the **createTransientMethod**, **executeTransientMethod**, and **deleteTransientMethod** methods to select objects according to dynamic selection criteria. For example, an input of **'name[1] = "M"'** would display customers whose names start with the letter **M**. The selection expression needs to be compiled only once, and is more efficient than using the **executeScript** method where the compilation overhead would occur for each customer.

```
vars
    input, str    : String;
    meth          : JadeMethod;
```

```
    err, pos, len : Integer;
    cust          : Customer;
begin
    read input;
    str  := "return " & input & ";";
    meth := process.createTransientMethod("select", Customer,
                        currentSchema, str, true, Boolean, err, pos, len);
    if meth <> null then
        foreach cust in Company.firstInstance.allCustomers
            where process.executeTransientMethod(meth, cust).Boolean do
            write cust.name;
        endforeach;
    else
        write "Compiler error " & err.String & " - " &
              process.getErrorText(err);
    endif;
epilog
    if meth <> null then
        process.deleteTransientMethod(meth);
    endif;
end;
```

For details about the **ParamListType** pseudo type specified in the last formal parameter (**params**), see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

extractRequestStatistics

Signature

```
extractRequestStatistics(proc:          Process output;
                         jdo:           JadeDynamicObject input;
                         localOrRemote: Integer;
                         any:           Any);
```

The **extractRequestStatistics** method of the **Process** class extracts request statistics from the **userInfo** part of notifications sent in response to **sendRequestStatistics** method requests. The extracted statistics are inserted as attributes in a **JadeDynamicObject** instance.

The parameters for the **extractWebStatistics** method are listed in the following table.

Parameter	Description
proc	An output parameter that receives a reference to the **Process** instance to which the statistics relate
jdo	A **JadeDynamicObject** instance into which the statistics values are placed as attributes
localOrRemote	To extract local request statistics (event type **Process_Local_Stats_Event**), set to one (1), or to extract remote request statistics (event type **Process_Remote_Stats_Event**), set to two (2)
any	The **userInfo** part of the notification that was received

The calling process is responsible for creating the **JadeDynamicObject** instance that is used as the **jdo** parameter. Any existing attributes that the instance has are cleared every time the method is called.

For a list and explanations about the properties that are returned by this method, see "Process::*getRequestStatistics Method*, in Chapter 4 of the *JADE Object Manager Guide*.

If the **any** parameter is not recognized as containing encoded Web statistics values, a 1000 exception is raised (*Invalid parameter type*), a 1002 exception is raised (*Invalid parameter value*), or a 1137 exception (*An internal data packet inconsistency was detected*) is raised. This could happen if the **any** parameter is not the **userInfo** part of a notification received in response to a **sendRequestStatistics** request.

The following examples show methods for a form to obtain and display information about local and remote request statistics.

```
load() updating;
begin
    //register to receive local and remote process request statistics
    beginNotification(process, Process_Local_Stats_Event,
                      Response_Continuous, 0);
    beginNotification(process, Process_Remote_Stats_Event,
                      Response_Continuous, 0);
end;

unload() updating;
begin
    //register to receive local and remote process request statistics
    beginNotification(process, Process_Local_Stats_Event,
                               Response_Continuous, 0);
    beginNotification(process, Process_Remote_Stats_Event,
                      Response_Continuous, 0);
end;

userNotify(eventType: Integer; theObject: Object; eventTag: Integer;
           userInfo: Any) updating;
begin
    if eventType = Process_Local_Stats_Event then
        displayRequestStats(1 /*local*/, userInfo);
        return;
    elseif eventType = Process_Remote_Stats_Event then
        displayRequestStats(2 /*remote*/, userInfo);
        return;
    endif;
    //...any other notification handling goes here...
end;

displayRequestStats(localOrRemote: Integer; any: Any);
vars
    targetProcess: Process;
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    process.extractRequestStatistics(targetProcess, jdo,
                                     localOrRemote, any);
    if localOrRemote = 1 then
        write "Local Request Statistics for " & targetProcess.String;
    else
        write "Remote Request Statistics for " & targetProcess.String;
    endif;
    write jdo.display;
```

```
epilog
    delete jdo;
end;

askForLocalRequestStats(targetProc: Process);
begin
    targetProc.sendRequestStatistics(1);
end;

askForRemoteRequestStats(targetProc: Process);
begin
    targetProc.sendRequestStatistics(2);
end;
```

extractWebStatistics

Signature

```
extractWebStatistics(proc: Process output;
                     jdo:  JadeDynamicObject input;
                     any:  Any);
```

The **extractWebStatistics** method of the **Process** class extracts Web performance statistics from the **userInfo** part of notifications sent in response to **sendWebStatistics** method requests, defined in the **Process** class. The extracted statistics are inserted as attributes in a **JadeDynamicObject** instance.

The parameters for the **extractWebStatistics** method are listed in the following table.

Parameter	Description
proc	An output parameter that receives a reference to the **Process** instance to which the statistics relate
jdo	A **JadeDynamicObject** instance into which the statistics values are placed as attributes
any	The **userInfo** part of the notification that was received

The calling process is responsible for creating the **JadeDynamicObject** instance that is used as the **jdo** parameter. Any existing attributes that the instance has are cleared every time the method is called. If the process that sent the notification is not using Web services, no attributes are added to the **JadeDynamicObject** instance; otherwise the attributes listed in the following table are added.

Attribute	Type
maximumResponseTime	Integer64 (milliseconds)
minimumResponseTime	Integer64 (milliseconds)
totalRequests	Integer64
totalResponseTime	Integer64 (milliseconds)
rejectedRequests	Integer64

If the **any** parameter is not recognized as containing encoded Web statistics values, a 1000 exception is raised (*Invalid parameter type*), a 1002 exception is raised (*Invalid parameter value*), or a 1137 exception (*An internal data packet inconsistency was detected*) is raised. This could happen if the **any** parameter is not the **userInfo** part of a notification received in response to a **sendWebStatistics** request.

The following examples show methods for a form to obtain and display information about Web statistics.

```
load() updating;
begin
    //register to receive Web statistics
    beginNotification(process, Process_Web_Stats_Event,
                      Response_Continuous, 0);
end;

unload() updating;
begin
    endNotification(process, Process_Web_Stats_Event);
end;

userNotify(eventType: Integer; theObject: Object; eventTag: Integer;
           userInfo: Any) updating;
begin
    if eventType = Process_Web_Stats_Event then
        displayWebStats(userInfo);
        return;
    endif;
    //...any other notification handling goes here...
end;

displayWebStats(any: Any);
vars
    targetProcess: Process;
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    process.extractWebStatistics(targetProcess, jdo, any);
    write "Web Statistics for " & targetProcess.String;
    write jdo.display;
epilog
    delete jdo;
end;

askForWebStats(targetProc: Process);
begin
    targetProc.sendWebStatistics();
end;
```

finalizePackages

Signature `finalizePackages() updating;`

The **finalizePackages** event method of the **Process** class calls the **Application** object **finalize** event to perform any terminate function common to all applications containing packages.

Normally these are executed automatically, if you run an application from the schema. However, they are not executed when a **JadeScript** or a Workspace method is executed unless you call the **finalizePackages** event method. (This maintains these interfaces in as light a weight as possible.) JADE does not initialize any packages for **RootSchema** applications running in user schemas, including unit tests run using the JADE Unit Test framework and the default RPS **Datapump** application in the **RootSchema**. If you require this initialization, you must initialize packages in your application code (using the **Process** class **initializePackages** and **finalizePackages** methods).

See also the **Process** class **initializePackages** method.

getAllApps

Signature `getAllApps(apps: ApplicationArray input);`

The **getAllApps** method of the **Process** class populates an array with all applications that are active in the process of the receiver. Use this method, for example, when you are working with imported packages to access all active forms across the main process **Application** object and all package **Application** objects.

As the main process **Application** object is always added to the array first, **apps[1]** is always the application of the main process when you call this method (when the **apps** parameter is empty before the call).

Notes The array membership is **Application**, so your calling method can deal only with the application member objects at the **RootSchema** level. To access schema-specific **Application** subclass features, use a type guard or indirect access (for example, an **Object** class **getPropertyValue** or **sendMsg** method call). For details about type guards, see "Using Type Guards", in Chapter 1 of the *JADE Developer's Reference*.

If you are using packages and you have two packages (for example, **p1** and **p2**), the second package (**p2**) imports the first package (**p1**), and a third schema imports both the **p1** and **p2** packages, there will be two instance of **app** for the first package (**p1**); that is, one in the context of the importing schema and the other in the second **p2** package.

getBufferStatistics

Signature
```
getBufferStatistics(obj: Object;
                    jdo: JadeDynamicObject): Boolean;
```

The **getBufferStatistics** method of the **Process** class returns cache-related information about the object specified by the **obj** parameter. The cache information is returned as attributes inserted into the **JadeDynamicObject** instance specified by the **jdo** parameter.

The calling process is responsible for creating and deleting this instance. Any existing attributes in the **JadeDynamicObject** instance are cleared when the **getBufferStatistics** method is called.

The method returns **true** if the object was in cache at the time of the call, and **false** if the object was not in cache. It does not load an object into cache if it was not already present.

For details about the attributes inserted into the **JadeDynamicObject** instance, see "Process::*getBufferStatistics* Method", in Chapter 4 of the *JADE Object Manager Guide*.

The **getBufferStatistics** method can be used with persistent and transient objects. With transient objects, a process can examine only shared transient objects and its own non-shared transient objects.

A process can use only its own **Process** instance as the method receiver. Using any other **Process** instance causes a 1265 exception (*Environmental object operation is out of scope for process*) to be raised.

The following example shows the use of the **getBufferStatistics** method.

```
showAnimalsInCache();
vars
    root : Root;
    animal : Animal;
    jdo : JadeDynamicObject;
begin
    create jdo transient;
    root := Root.firstInstance;
    foreach animal in root.allAnimals do
        if process.getBufferStatistics(animal, jdo) then
            write animal.String & " is loaded in cache";
            write "  Size (bytes)   = " &
                      jdo.getAttributeValue("size").Integer.String;
            write "  Lives          = " &
                      jdo.getAttributeValue("lives").Integer.String;
            write "  Cycles         = " &
                      jdo.getAttributeValue("cycles").Integer.String;
            write "  Flag           = " &
                      jdo.getAttributeValue("flag").Integer.String;
            write "  Operations     = " &
                      jdo.getAttributeValue("operations").Integer64.String;
            write "  Age(node ticks) =" &
                      jdo.getAttributeValue("age").Integer64.String;
        endif;
    endforeach;
epilog
    delete jdo;
end;
```

The output from the **getBufferStatistics** method shown in the previous example is as follows.

```
Animal/2096.11345 is loaded in cache
  Size (bytes)   = 191
  Lives          = 1
  Cycles         = 15
  Flag           = 1
  Operations     = 2
  Age(node ticks)= 17525
Animal/2096.456 is loaded in cache
  Size (bytes)   = 191
  Lives          = 1
  Cycles         = 10
  Flag           = 1
  Operations     = 2
  Age(node ticks)= 32016
Animal/2096.987 is loaded in cache
  Size (bytes)   = 191
  Lives          = 1
  Cycles         = 10
  Flag           = 1
  Operations     = 2
  Age(node ticks)= 67374
Animal/2096.1000 is loaded in cache
  Size (bytes)   = 191
```

```
Lives          = 1
Cycles         = 10
Flag           = 1
Operations     = 2
Age(node ticks)= 68384
```

getCallStackInfo

Signature `getCallStackInfo(): String;`

The **getCallStackInfo** method of the **Process** class retrieves information about the call stack of the executing process. This method can be called only on the process instance of the current process. If you call it on an instance for another process, an exception is raised.

The return value from the **getCallStackInfo** method contains environmental details in addition to the local and remote call stacks. The location of execution for each method is signified at the end of the method line as **(C)** for client node or **(S)** for server node or single user, followed by the source code line of each method.

The following example shows the output from the **getCallStackInfo** method.

```
Method call stack information retrieval for Process: {13} commenced
Environmental details:
Node: 4068 Process: 13
Current Schema: CallStackInfo Application: CallStackInfo
Method Call Stack:
<<Process/187.03>> Process::getCallStackInfo(66) (S)
++ Source line: return self._getCallStackInfo(true);
<<JadeScript/107.0 (s)>> JadeScript::testCallStackInfo(54) (S)
++ Source line: write process.getCallStackInfo;
Method call stack information retrieval for Process: {13} finished
```

getCommandLine

Signature `getCommandLine(): String;`

The **getCommandLine** method of the **Process** class returns a string containing the current command line of the process of the receiver. A method on a specific process instance performs its action on the owning node (that is, a **process.node** instance) if the process is not associated with a presentation client.

If the process has an associated presentation client, the action is performed on the presentation client. The presentation client does not have to be the current presentation client or a presentation client attached to the same application server.

Use the **Node** class **getCommandLine** method to obtain the file from the application server.

The following example shows the use of the **getCommandLine** method.

```
vars
    cmdLine, myOption : String;
    int               : Integer;
begin
    cmdLine := process.getCommandLine;                  // get command line
    // look for my command line option ('myOption')
    int     := cmdLine.pos('myOption', 1);
    if int <> 0 then
        int := cmdLine.pos('=', int) + 1;               // look for '='
        // skip any blanks after the '='
```

```
            cmdLine.scanWhile(' ', int);
            // return input up to next blank
            myOption := cmdLine.scanUntil(' ', int);
            write myOption;
        endif;
    end;
```

getComputerName

Signature `getComputerName(): String;`

The **getComputerName** method of the **Process** class returns a string containing the computer name of the machine that owns the process (for example, **"wilbur2a"**).

If the receiving process belongs to a presentation client, the **getComputerName** method returns the computer name of the machine where the presentation client is running; otherwise, it returns the computer name of the machine where the node owning the process is running.

getDateTimeDelta

Signature
```
getDateTimeDelta(deltaDate: Integer output;
                 deltaTime: Integer output);
```

The **getDateTimeDelta** method of the **Process** class retrieves the values used to adjust initial date and time local variable values used by the receiving process. (For details about setting the initial **Date**, **Time**, and **TimeStamp** local variables, see the **setDateTimeDelta** method.)

getExceptionHandlerStack

Signature `getExceptionHandlerStack(oa: ObjectArray input);`

The **getExceptionHandlerStack** method of the **Process** class populates a transient instance of the **ObjectArray** class with transient instances of the **ExceptionHandlerDesc** class that represent the exception handlers armed by the receiving process in the current node.

In the array, locally armed exception handlers precede globally armed exception handlers. Within this grouping, the most recently armed exception handlers occur first.

The method displays only exception handlers the process has armed on the current node. For example, it does not display global exception handlers armed in a **serverExecution** method, unless executed from a **serverExecution** method.

The use of the **getExceptionHandlerStack** method is shown in the following code example. It is the responsibility of the calling method to delete the transient instances of the **ExceptionHandlerDesc** class.

```
showArmedExceptionHandlers();
vars
    oa: ObjectArray;
    o:  Object;
begin
    create oa transient;
    process.getExceptionHandlerStack(oa);
    foreach o in oa do
        write o.display;
    endforeach;
epilog
```

```
    oa.purge;
    delete oa;
end;
```

getErrorText

Signature `getErrorText(errorCode: Integer): String;`

The **getErrorText** method of the **Process** class returns the message text for a JADE-defined error code.

getIniFileName

Signature `getIniFileName(): String;`

The **getIniFileName** method of the **Process** class returns the full path and file name of the JADE initialization file; for example:

```
c:\jade\system\jade.ini
```

The name of the JADE initialization file is returned in the form that it was entered on the command line. If no initialization file name was specified, JADE looks for an initialization file with the name **jade.ini** in the default location and either finds the file or creates it. The name and full path of that *default* initialization file is returned with forward slash characters (for example, **c:/jade/system/jade.ini**).

A method on a specific process instance performs its action on the owning node (that is, a **process.node** instance) if the process is not associated with a presentation client. If the process has an associated presentation client, the action is performed on the presentation client. The presentation client does not have to be the current presentation client or a presentation client attached to the same application server.

Use the **Application** class **getIniFileNameAppServer** method or **Node** class **getIniFileName** method to obtain the file from the application server.

Note If you create a shortcut that has the **newcopy** parameter set to **false** and you specify a different JADE initialization file from the one with which the process was started, the active JADE initialization file is the one that was specified when the process started up and *not* the one specified in the **newcopy=false** shortcut.

Calling the **getIniFileName** method in the new process enables you to get the name of the initialization file that was used when the process started up.

getJadeInstallDirectory

Signature `getJadeInstallDirectory(): String;`

The **getJadeInstallDirectory** method of the **Process** class returns a string containing the JADE installation directory, from which the JADE executable program is running; for example:

```
c:\jade\bin
```

getJadeHomeDirectory

Signature `getJadeHomeDirectory(): String;`

The **getJadeHomeDirectory** method of the **Process** class returns a string containing the JADE HOME directory, which is the parent directory of the JADE installation directory; for example:

```
c:\jade      (if the installation directory was c:\jade\bin)
```

getJadeWorkDirectory

Signature `getJadeWorkDirectory(): String;`

The **getJadeWorkDirectory** method of the **Process** class returns a string containing the directory where work files are created by JADE.

When you call the **getJadeWorkDirectory** method and the directory does not exist, JADE creates it based on the value of the **JadeWorkDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file.

By default, this directory is created at the same level as the JADE installation directory (that is, the directory in which the **jade.exe** executable program is located) and is named **temp**. For example, if the JADE installation directory is **c:\jade\bin**, the working directory would be **c:\jade\temp**.

The cache file for a thin client (which contains all forms and pictures sent by logic from the application server) is stored in the work directory, unless another location is specified by the **FormCacheFile** parameter in the [JadeThinClient] section. The thin client automatic download interlock file (**thinlock.fil**) is also created in the work directory.

getLastExtFunctionCallError

Signature `getLastExtFunctionCallError(): Integer;`

The **getLastExtFunctionCallError** method of the **Process** class returns the value of the error code set by the last external function call made by the current process.

This corresponds to the value returned by a call to the **GetLastError** Windows API.

getLockCallStackFilter

Signature `getLockCallStackFilter(classList: ClassColl input);`

The **getLockCallStackFilter** method of the **Process** class returns the list of classes for which saving lock call stack information has been enabled for this process.

You can call this method for any running **Process** instance in the system.

getMethodCacheLimit

Signature `getMethodCacheLimit(): Integer64;`

The **getMethodCacheLimit** method of the **Process** class retrieves the method cache limit for the executing process

This method can be called only on the process instance of the current process. Exception 1265 (*Environmental object operation is out of scope for process*) is raised if you call the method on an instance for another process.

getMethodCacheStatistics

Signature `getMethodCacheStatistics(jdo: JadeDynamicObject input);`

The **getMethodCacheStatistics** method of the **Process** class retrieves information about the method cache of the executing process and stores it in the dynamic object instance specified by the **jdo** parameter. The **JadeDynamicObject** is updated to include properties that contain statistics about the method cache and string pool of the executing process.

This method can be called only on the process instance of the current process. Exception 1265 (*Environmental object operation is out of scope for process*) is raised if you call the method on an instance for another process.

To send a notification containing statistics about the method cache of the target process, call the **sendMethodCacheStatistics** method.

The following table lists the properties added to the dynamic object instance.

Property	Type	Description
hasCacheManager	Boolean	**true** if the executing process has an associated cache manager; otherwise **false**. This would be **false** when the executing process is a background process. If the value is **false**, none of the other properties are added to the dynamic object.
processOid	String	Oid of the executing process.
singleMethodCache	Boolean	Determines if the executing process is using single or multiple method cache structure. This reflects the value of the **MethodCache** parameter in the [JadeInterpreter] section of the JADE initialization file.
methodCacheLimit	Integer64	Method cache limit for the executing process.
maximumCacheSize	Integer64	Largest cache size hit since the start of process execution.
numberOfMethodsInCache	Integer64	Current number of JADE methods in the method cache for the executing process.
totalMethodsDiscarded	Integer64	Total number of JADE methods discarded from the method cache for the executing process since the start of process execution.
totalMethodsExecuted	Integer64	Total number of JADE methods executed on this process since the start of process execution.
totalTimeLoadingMethods	Integer64	Amount of time in milliseconds spent loading methods into the method cache.
cacheOverruns	Integer64	Number of cache overruns in the method cache since the start of process execution.
stringPoolLimit	Integer64	String pool size limit for the executing process.
maximumStringPoolSize	Integer64	Largest string pool size hit since the start of process execution.
numberOfStringsInPool	Integer64	Current number of strings in the string pool for the executing process.
stringPoolOverruns	Integer64	Number of string pool overruns since the start of process execution.
cacheOverrunXXtoYYPercent	Integer64	Number of cache overruns *XX* to *YY* percent over the method cache limit.
cacheOverrun100PlusPercent	Integer64	Number of cache overruns greater than 100 percent over the method cache limit.

getMethodProfileInfo

Signature `getMethodProfileInfo(jdo: JadeDynamicObject input; truncated: Boolean output);`

The **getMethodProfileInfo** method of the **Process** class retrieves method profiling information for the process specified as the method receiver.

The method receiver can be any current process, including the requesting process itself or a process executing on another node.

The retrieved information is presented as a set of dynamic objects in the **children** collection of the dynamic object instance specified by the **jdo** parameter; that is, the information is contained in the collection **jdo.children**.

The calling process is responsible for creating and deleting the **JadeDynamicObject** passed to the method. It is also responsible for deleting the **JadeDynamicObject** instances inserted into the **jdo.children** collection (for example, by purging the collection). If the **JadeDynamicObject** instance used as the **jdo** parameter is persistent, the **JadeDynamicObject** instances added to the **children** collection are also persistent. Similarly, if the object is transient, the child dynamic objects are also transient.

For details about the properties of the **JadeDynamicObject** instances in the **children** collection, see "Process::*getMethodProfileInfo* Method", in Chapter 4 of the *JADE Object Manager Guide*.

Time spent in recursive method calls is correctly accounted for as time spent executing the method.

The **truncated** parameter indicates if the amount of method profile information to be retrieved exceeded an internal buffer limit so was truncated. If the **truncated** parameter returns **false**, all method profiling information is present in the **JadeDynamicObject** instance specified in the **jdo** parameter. If it returns **true**, some entries have been truncated. When truncation occurs, entries with the lowest "total calls" amounts are omitted.

Notes Truncation occurs when the amount of profiling information to be returned exceeds 1,000 entries.

Truncation occurs only when method profiling information is retrieved for processes running on remote nodes.

The CPU time has a granularity of 10 or 15 milliseconds. This means that the CPU time figures for methods of short duration are subject to inaccuracy due to the large granularity. However, the clock times have a much smaller granularity and are therefore more accurate.

Note Clock times may fluctuate, depending on other activity on the same machine. The total clock times include time spent waiting; for example, to wait for a **Window** event, to lock an object, or for a user response to a modal form.

The **children** collection of the **jdo** parameter passed to **getMethodProfileInfo** is purged each time the method is called.

When displaying method profiling results, you can use the **qualifiedName** method of the corresponding **Method** instance to obtain the name of a profiled method, and the **isKindOf** method to determine if the method is an external method or a JADE method.

The **removeMethodProfileInfo** method can be called to remove profiling information.

If you call the **getMethodProfileInfo** method on a target process that has terminated, an exception is raised.

The following example shows the use of the **getMethodProfileInfo** method.

```
tryProfiling();
vars
    jdo : JadeDynamicObject;
```

```
    child : JadeDynamicObject;
    mth : Method;
    calls : Integer64;
    clockTime : Integer64;
    mthType : String;
    truncated : Boolean;
begin
    create jdo transient;
    process.profileMethod(JadeScript::updateAnAnimal, true);
    process.beginMethodProfiling(2);
    updateAnAnimal;
    process.endMethodProfiling();
    process.getMethodProfileInfo(jdo, truncated);
    foreach child in jdo.children do
        mth := child.getPropertyValue("method").Method;
        calls := child.getPropertyValue("calls").Integer64;
        clockTime := child.getPropertyValue("clockTimeInMethod").Integer64;
        if mth.isKindOf(ExternalMethod) then
            mthType := "EXTERNAL METHOD ";
        elseif mth.isKindOf(JadeMethod) then
            mthType := "JADE     METHOD ";
        else
            mthType := "OTHER    METHOD";
        endif;
        write mthType & " " & mth.qualifiedName & " Calls=" & calls.String
                & " ClockTime=" & clockTime.String;
    endforeach;
    process.removeMethodProfileInfo();
epilog
    jdo.children.purge;
    delete jdo;
end;
```

The output from the **getMethodProfileInfo** method shown in the previous example is as follows.

```
EXTERNAL METHOD  Type::getMethod Calls=8 ClockTime=84
EXTERNAL METHOD  List::_values Calls=8 ClockTime=30
EXTERNAL METHOD  Class::firstInstance Calls=1 ClockTime=570
JADE     METHOD  JadeScript::updateAnAnimal Calls=1 ClockTime=1793308
EXTERNAL METHOD  Dictionary::getAtKey Calls=84 ClockTime=911
EXTERNAL METHOD  Schema::getClassByNumber Calls=5 ClockTime=4512
EXTERNAL METHOD  ArrayBlock::_loadValues Calls=3 ClockTime=9
EXTERNAL METHOD  Btree::_values Calls=143 ClockTime=463
EXTERNAL METHOD  SetBlock::_loadValues Calls=5 ClockTime=10
EXTERNAL METHOD  DbFile::path Calls=4 ClockTime=24
EXTERNAL METHOD  DictBlock::_loadValues Calls=38 ClockTime=76
EXTERNAL METHOD  DictBlock::_search Calls=100 ClockTime=187
```

getOSDetails

Signature `getOSDetails(jdo: JadeDynamicObject input);`

The **getOSDetails** method of the **Process** class populates a **JadeDynamicObject** object with information about the operating system and architecture of the receiving process.

In JADE thin client mode, this method returns the operating system details of the presentation client. (To return the operating system details of the application server workstation that is running the JADE logic, use the **getOSDetails** method of the **Node** class.)

The method enables you to determine the various usages of JADE for a specific environment; for example, the type of binaries required for thin client downloads (for example, **x64-msoft-win64-ansi**).

The properties that are returned in the dynamic object specified in the **jdo** parameter are listed in the following table.

Property	Type	Description
version	String	Specific version of the operating system.
architecture	Integer	Internal byte ordering and alignment information relevant to JADE release. It is used by the **setByteOrderLocal** and **setByteOrderRemote** methods of the **Character**, **Date**, **Decimal**, **Integer**, **Integer64**, **Real**, **Time**, and **TimeStamp** primitive types. The architecture can be one of the values listed in the following table.

Node Class Constant	Description
Architecture_32Big_Endian	32-bit big-endian internal byte ordering and alignment
Architecture_32Little_Endian	32-bit little-endian internal byte ordering and alignment
Architecture_64Big_Endian	64-bit big-endian internal byte ordering and alignment
Architecture_64Little_Endian	64-bit little-endian internal byte ordering and alignment
Architecture_Gui	Binary data passed in the byte order of the GUI system (currently Windows 32-bit little-endian)

platformId	Integer	Operating system of the server node of the receiver object. The operating system returned by this method can be one of the values listed in the following table.

Node Class Constant	Description
OSWindowsEnterprise	Microsoft Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008
OSWindowsHome	Microsoft Windows 98 (not a supported operating system)
OSWindowsMobile	Microsoft Windows CE (not a supported operating system)

buildArchitecture	String	Details about the platform and build type for which the binaries where built (for example, **x64-msoft-win64-ansi**). Can be used to determine the type of binaries required for thin client downloads.

Property	Type	Description
currentBuildArchitectureList	String	Complete list of current **buildArchitecture** strings, separated by semicolons.
fullBuildArchitectureList	String	Complete list of past and current **buildArchitecture** strings, separated by semicolons.
isBigEndian	Boolean	Indicates if CPU for the node is running big-endian (PowerPC can switch from big-endian to little-endian, and the reverse).
characterSize	Integer	**1** for ANSI, **2** for Unicode.
addressWidth	Integer	**32** indicates 32-bit executing binaries, **64** indicates 64-bit executing binaries.
osAddressWidth	Integer	**32** indicates a 32-bit operating system, **64** indicates a 64-bit operating system.
osVersionEnum	Integer	Internal unique number representing the operating system and hardware combination
osVersionString	String	Description of the operating system in a readable format.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

The following example shows the use of the **getOSDetails** method.

```
vars
    jdoProcess : JadeDynamicObject;
    str, str2  : String;
    pos        : Integer;
begin
    create jdoProcess transient;
    process.getOSDetails(jdoProcess);
    str:=jdoProcess.getPropertyValue("currentBuildArchitectureList").String;
    pos := 1;
    while true do
        str2 := str.scanUntil(";", pos);
        write str2;
        if pos = null then
            break;
        endif;
        pos := pos + 1;
    endwhile;
epilog
    delete jdoProcess;
end;
```

The output from the method shown in the previous example is as follows.

```
i686-msoft-win32-ansi
i686-msoft-win32-unicode
armv4i-msoft-wince50-unicode
i686-msoft-x86emu-unicode
x64-msoft-win64-ansi
x64-msoft-win64-unicode
armv4i-msoft-wm60-unicode
```

getOSPlatform

Signature `getOSPlatform(version: String output;`
`architecture: Integer output): Integer;`

The **getOSPlatform** method of the **Process** class returns an integer value that indicates the operating system of the process of the receiver.

In JADE thin client mode, this method returns the operating platform of the presentation client. (To return the operating system of the application server workstation that is running the JADE logic, use the **getOSPlatform** method of the **Node** class.)

The operating system can be one of the values listed in the following table.

Node Class Constant	Operating system is …
OSWindowsEnterprise	Microsoft Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008
OSWindowsHome	Microsoft Windows 98 (not a supported operating system)
OSWindowsMobile	Microsoft Windows CE (not a supported operating system)

The **version** parameter specifies the specific version of the operating system. The **architecture** parameter is a unique number that indicates internal byte ordering and alignment information relevant to this release of JADE. It is used by the **Character**, **Date**, **Decimal**, **Integer**, **Integer64**, **Real**, **Time**, and **TimeStamp** primitive type **setByteOrderLocal** and **setByteOrderRemote** methods.

You can use the **OSWindows** constant of the **Node** class, which is a bit mask that enables you to identify a family of operating systems, as shown in the following example.

```
vars
    platform     : Integer;
    version      : String;
    architecture : Integer;
begin
    platform := process.getOSPlatform(version, architecture);
    if platform.bitAnd(Node.OSWindows) <> 0 then
        // operating system is Windows family (2012 or 2008)
        if platform = Node.OSWindowsHome then
            // version is an older version of Windows (unsupported)
            return 'Windows (unsupported) ' & version;
        endif;
        if platform = Node.OSWindowsEnterprise then
            // version is Windows 10, Windows 8, Windows 7, Windows
            // Server 2016, Windows Server 2012, or Windows Server 2008
            return 'Windows ' & version;
        endif;
        if platform = Node.OSWindowsMobile then
            // version is Windows CE
            return 'Windows CE (unsupported) ' & version;
        endif;
    endif;
    return '* Unknown platform: ' & platform.String & ' version: ' &
           version;
end;
```

getPersistentDeadlockPriority

Signature `getPersistentDeadlockPriority(): Integer;`

The **getPersistentDeadlockPriority** method of the **Process** class retrieves the priority value to be used when dealing with deadlocks involving persistent objects.

This method can be called only on the process instance of the current process.

If you call it on an instance for another process, an exception is raised.

getProcessApp

Signature `getProcessApp(): Application;`

The **getProcessApp** method of the **Process** class returns a reference to the main **Application** object of the current process. For example, to return the locale of the main application, call **process**.**getProcessApp**.**currentLocale**.

When using packages, a single process can have multiple transient **Application** objects: one for the process itself and one for each imported package.

If the process is not using any packages, this method returns the same reference as the **app** system variable.

If a process is using one or more imported packages, this method returns a reference to the **Application** object of the process, regardless of the context from which this method was called. The difference in referencing the main **Application** object when packages are involved is because the **app** system variable will be different while executing within the package. The value returned by this method is the same as **app** when the running process began execution, and before any possible changes, owing to context switches into packages.

getProfileString

Signature
```
getProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 default:  String): String;
```

The **getProfileString** method of the **Process** class retrieves a string from the specified section in an initialization file. (The **setProfileString** method copies the string into the specified section of an initialization file.)

A method on a specific process instance performs its action on the owning node (that is, a **process**.**node** instance) if the process is not associated with a presentation client. If the process has an associated presentation client, the action is performed on the presentation client. The presentation client does not have to be the current presentation client or a presentation client attached to the same application server.

Use the **Application** class **getProfileStringAppServer** method or **Node** class **getProfileString** method to obtain the file from the application server.

The parameters for the **getProfileString** method are listed in the following table.

Parameter	Specifies the …
fileName	Initialization file. If you set this parameter to **windows**, the **win.ini** file is used. If it does not contain a full path to the file, Windows searches for the file in the Windows directory.
section	Initialization file section containing the key (parameter) name.

Parameter	Specifies the …
keyName	Name of the key (parameter) whose associated string is to be retrieved.
default	Default value for the specified key if the key cannot be found in the initialization file.

You can return all initialization file sections or all parameters in a section, by using the **JadeProfileString** category global constants listed in the following table.

Global Constant	Specified in the…	Returns all…
ProfileAllKeys	**keyName** parameter	Key (parameter) strings in the initialization file section, separated by spaces
ProfileAllSections	**section** parameter	Initialization file sections, separated by spaces

You can use this method to retrieve a string from a two-level section name (prefixed with a unique identifier) within a JADE initialization file shared by multiple programs on the same host. For details, see "Two-Level Section Names" under "Format of the JADE Initialization File", in the *JADE Initialization File Reference*.

The following example shows the use of the **getProfileString** method to determine the server for the current JADE initialization file.

```
vars
    server : String;
begin
    server := process.getProfileString(process.getIniFileName, "JadeClient",
                                       "ServerName", null);
    write "server name is " & server;
end;
```

getProgramDataDirectory

Signature `getProgramDataDirectory(): String;`

The **getProgramDataDirectory** method of the **Process** class returns a string containing the path of the program data directory. The program data directory is used to share files among the users of the executables; for example, the **jommsg.log** file or shared dictionary spelling files that are updated.

If JADE is not installed under the **\Program Files** directory, the path of the JADE HOME directory is returned.

If JADE is installed under the **\Program Files** directory, the value that is returned by the **getProgramDataDirectory** method depends on the value of the **ProgramDataDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file. If the directory does not exist, JADE creates it.

The values of the **ProgramDataDirectory** parameter and the corresponding values returned by the **getProgramDataDirectory** method are shown in the following table.

ProgramDataDirectory Value	Return Value
<default>	The path of the JADE HOME directory with the **\Program Files** portion replaced with the programmatically obtained path of the common application data directory. For example, a presentation client installed into **\Program Files\Jade Software\parsys** returns **\ProgramData\Jade Software\parsys**.
<homedir>	The path of the JADE HOME directory.
<programdata>	The same as for **<default>**.

ProgramDataDirectory Value	Return Value
Directory name	Directory name.

getRequestStatistics

Signature `getRequestStatistics(dynObj: JadeDynamicObject input; localOrRemote: Integer);`

The **getRequestStatistics** method of the **Process** class retrieves node sampling values relating to the current process that is executing the method. The values are returned as properties of a **JadeDynamicObject** object.

You can use this method to directly retrieve node sampling information; for example, if you want to calculate resource usage over the duration of a transaction. The returned values include cumulative counters that are not reset by the method calls. JADE applications that use this method therefore need to compare values from one call to the next, to work out the value differences.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

You should use the **getRequestStatistics** method only when node sampling is enabled. If node sampling is not enabled, this method returns zero (**0**) for all values other than for process ticks.

If the **localOrRemote** parameter is set to **1**, the statistics for all requests invoked on the local node are returned as properties in the dynamic object specified in the **dynObj** parameter. If the **localOrRemote** parameter is set to **2**, the statistics for all requests from the local node to remote nodes are returned as properties in the specified dynamic object.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

For a list and explanations about the properties that are returned by this method, see "Process::*getMethodProfileInfo* Method", in Chapter 4 of the *JADE Object Manager Guide*.

If the dynamic object passed to the method already contains properties but they do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. This method is most efficient when the properties match those to be returned.

You can call this method only with the current process as the receiver. An exception is raised if you attempt to call this method for a different process.

getRpcServerStatistics

Signature `getRpcServerStatistics(jdo: JadeDynamicObject input; detailed: Boolean);`

The **getRpcServerStatistics** method of the **Process** class retrieves Remote Procedure Call (RPC) statistics relating to requests and replies sent to and from the database server node and the process specified by receiver object. The method receiver can be any current process, including the requesting process itself or a process executing on another node.

The values returned represent information about the connection between client node and database server node, and requests and replies between the nodes for the specified process. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

The **detailed** parameter specifies whether the values include individual totals for each request type.

For details about the attributes returned in the dynamic object, see "Process::*getRpcServerStatistics* Method", in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match properties to be returned, existing dynamic object properties are removed and replaced with appropriate properties. This method is most efficient when properties match those to be returned.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getRpcServerStatistics** method.

```
showRpcProcessStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   process.getRpcServerStatistics(jdo, false);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getRpcServerStatistics** method shown in the previous example is as follows.

```
---RPCServerStatistics(106)---
timeStarted = 27 April 2007, 12:32:07
connectionType = 1
lastInboundRequest = 27 April 2007, 14:43:25
requestsFromClients = 706
repliesToClients = 705
requestPacketsFromClients = 706
replyPacketsToClients = 705
requestBytesFromClients = 111258
replyBytesToClients = 164806
requestsToClients = 0
repliesFromClients = 0
requestPacketsToClients = 0
replyPacketsFromClients = 0
requestBytesToClients = 0
replyBytesFromClients = 0
notificationPacketsToClients = 0
notificationBytesToClients = 0
```

getSaveLockCallStack

Signature `getSaveLockCallStack(): Boolean;`

The **getSaveLockCallStack** method of the **Process** class returns **true** when an object is locked if the lock call stack is being saved for a process; otherwise, it returns **false**.

getSignOnUsage

Signature `getSignOnUsage(): Integer;`

The **getSignOnUsage** method of the **Process** class returns the way in which a currently logged on user signed on to JADE (for example, call this method, for example, when standard security is handled by the JADE start-up form of your application and you want to validate users who sign on to the JADE system from ODBC).

This method returns one of the **Process** class constants listed in the following table.

Class Constant	Integer Value	Signed on...
SignOn_Usage_NoAudit	2	In **NoAudit** mode
SignOn_Usage_OdbcLogin	3	As a JADE ODBC driver
SignOn_Usage_ReadOnly	1	In **ReadOnly** mode
SignOn_Usage_Update	0	In **Update** mode

When the current log-in is an ODBC log-in, the database mode is read-only.

getStringPoolLimit

Signature `getStringPoolLimit(): Integer64;`

The **getStringPoolLimit** method of the **Process** class retrieves the string pool limit for the executing process.

This method can be called only on the process instance of the current process. Exception 1265 (*Environmental object operation is out of scope for process*) is raised if you call the method on an instance for another process.

getTempPath

Signature `getTempPath(): String;`

The **getTempPath** method of the **Process** class returns a string containing the architecture-specific version of the directory in which temporary files are created on the node of the receiver object. For example, this method returns **TEMP** or **TMP**, as appropriate.

A method on a specific process instance performs its action on the owning node (that is, a **process.node** instance) if the process is not associated with a presentation client. If the process has an associated presentation client, the action is performed on the presentation client. The presentation client does not have to be the current presentation client or a presentation client attached to the same application server.

Use the **Node** class **getTempPath** method to obtain the temporary directory from the app server.

For details about returning the value of a specified environment variable, see the **Node** class **getEnvironmentVariable** method.

getTimers

Signature
```
getTimers(jdoa:  JadeDynamicObjectArray;
          count: Integer output);
```

The **getTimers** method of the **Process** class populates a user-supplied JADE dynamic object with timer-related information from the receiving process that is the method receiver.

The value of the **count** parameter is updated with the number of active timers. The values are returned as properties in the dynamic object array specified by the **jdoa** parameter. For details about the properties returned in the dynamic object array, see "Getting Timer Information", in Chapter 4 of the *JADE Object Manager Guide*.

The following example shows the use of the **getTimers** method and the resulting output.

```
vars
    count : Integer;
    jdoa : JadeDynamicObjectArray;
    jdo : JadeDynamicObject;
    str : String;
    int : Integer;
begin
    create jdoa transient;
    process.getTimers(jdoa,count);
    foreach jdo in jdoa do
        str := '---' & jdo.getName & '(' & jdo.type.String & ')---';
        foreach int in 1 to jdo.propertyCount do
            str := str & CrLf & jdo.getPropertyName(int) & " = " &
                   jdo.getPropertyValueByIndex(int).String;
        endforeach;
        write str;
    endforeach;
epilog
    jdoa.purge;
    delete jdoa;
end;

---CurrentTimers(117)---
receiver = Q/18536.1
eventTag = 1
delay = 20000
option = 1
fired = false
inCallBack = false
exceptionState = false
remainingTime = 11549
application = Test/18432.1
interfaceNumber = 0
serverExecution = false
```

getTrackedMethod

Signature `getTrackedMethod(): Method;`

The **getTrackedMethod** method of the **Process** class returns a reference to the **Method** instance of the tracked method that caused the current method to be invoked. (The current method is a *preamble* method called before the tracked method or a *postamble* method called after the tracked method.)

If the current method is not invoked as a result of method tracking, a **null** value is returned.

getTrackedMethodReceiver

Signature `getTrackedMethodReceiver(): Object;`

The **getTrackedMethodReceiver** method of the **Process** class returns a reference to the object that is used as the receiver for the method being tracked.

If the current method is not invoked as a result of method tracking, a **null** value is returned.

getTrackedMethodReturnValue

Signature `getTrackedMethodReturnValue(): Any;`

The **getTrackedMethodReturnValue** method of the **Process** class retrieves the return value of the method being tracked.

If the current method is not invoked as a result of a method returning execution or if the tracked method does not have a return value, a **null** value is returned.

getTrackedMethodStatus

Signature `getTrackedMethodStatus(): Integer;`

The **getTrackedMethodStatus** method of the **Process** class returns a value representing the current status of the tracked method. The values returned are listed in the following table.

Value	Description
0	No current tracked method
1	The tracked method is about to be called
2	The tracked method has just returned normally
3	The tracked method has just exited abnormally

An abnormal exit can occur when an exception has been raised causing further execution of the method to be abandoned (for example, when an exception handler returns **Ex_Abort_Action**).

getTransactionId

Signature `getTransactionId(): Decimal;`

The **getTransactionId** method of the **Process** class returns a value that represents the latest identifying number from the most recent **beginTransaction** instruction executed in the current process as a **Decimal** value (regardless of whether a **commitTransaction** or an **abortTransaction** instruction has been executed since the latest **beginTransaction** instruction).

getTransactionId64

Signature `getTransactionId64(): Integer64;`

The **getTransactionId64** method of the **Process** class returns a value that represents the latest identifying number from the most recent **beginTransaction** instruction executed in the current process as an **Integer64** value (regardless of whether a **commitTransaction** or an **abortTransaction** instruction has been executed since the latest **beginTransaction** instruction).

getTransactionTraceCallbacks

Signature `getTransactionTraceCallbacks(callbacks: JadeDynamicObject io);`

The **getTransactionTraceCallbacks** method of the **Process** class populates the **children** collection of the **JadeDynamicObject** instance passed as a parameter with further **JadeDynamicObject** instances containing the method and receiver for all currently enabled transaction trace callbacks; that is, those methods and receivers that have been enabled using the **enableTransTraceCallback** method of the **Process** class.

Each child **JadeDynamicObject** instance contains the following references, which are of type **Object**, relating to the callback method.

- **callbackReceiver**, which is the receiver
- **callbackMethod**, which is the instance of the **Method** class

The process is responsible for creating and deleting the **JadeDynamicObject** instance used as a parameter. The instances in the **children** collection are automatically deleted when the parent object is deleted.

The **JadeDynamicObject** instance is cleared every time the **getTransactionTraceCallbacks** method is called. This includes purging its children collection.

The following example shows the use of the **getTransactionTraceCallbacks** method.

```
vars
    jdo : JadeDynamicObject;
    jdoChild : JadeDynamicObject;
    o1, o2 : Object;
begin
    create jdo transient;
    process.getTransactionTraceCallbacks(jdo);
    foreach jdoChild in jdo.children do
        o1 := jdoChild.getPropertyValue
                            (JadeTransactionTrace.CallbackReceiver).Object;
        o2 := jdoChild.getPropertyValue
                            (JadeTransactionTrace.CallbackMethod).Object;
        write "Receiver " & o1.String & ", Method " & o2.Method.name;
    endforeach;
epilog
    delete jdo;
end;
```

getTransactionTraceObject

Signature `getTransactionTraceObject(): JadeTransactionTrace;`

The **getTransactionTraceObject** method of the **Process** class returns a reference to the transient instance of the **JadeTransactionTrace** class associated with the process. The object contains information gathered from the latest transaction when transaction tracing was active.

getTransientDeadlockPriority

Signature `getTransientDeadlockPriority(): Integer;`

The **getTransientDeadlockPriority** method of the **Process** class retrieves the priority value to be used when dealing with deadlocks involving shared transient objects. This method can be called only on the process instance of the current process. If you call it on an instance for another process, an exception is raised.

getTransientFileLength

Signature `getTransientFileLength(): Integer64;`

The **getTransientFileLength** method of the **Process** class returns the physical size of the transient database file in use by the executing process; that is, the current thread executing the current method.

See also "Transient Database File Analysis", in Chapter 3 of the *JADE Database Administration Guide*.

getTransientFileName

Signature `getTransientFileName(): String;`

The **getTransientFileName** method of the **Process** class returns the name of the transient database file in use by the executing process; that is, the current thread executing the current method.

Tip To obtain the path for this file, use the **getTransientDbPath** method of the **Application** class.

See also "Transient Database File Analysis", in Chapter 3 of the *JADE Database Administration Guide*.

getUserDataDirectory

Signature `getUserDataDirectory(): String;`

The **getUserDataDirectory** method of the **Process** class returns a string containing the path of the user data directory. The user data directory is used for files that are specific to each user of the JADE executables; for example, if a presentation client installation occurs on a Windows machine running Citrix or Terminal Services and all users run the same thin client binaries, any data created on the client file system should be stored under this directory (that is, in separate directories for each user).

If JADE is not installed under the **\Program Files** directory, the location of the JADE HOME directory is returned.

If JADE is installed under the **\Program Files** directory, the value that is returned depends on the value of the **UserDataDirectory** parameter in the [JadeEnvironment] section of the JADE initialization file. If the directory does not exist, JADE creates it.

The values of the **UserDataDirectory** parameter and the corresponding values returned by the **getUserDataDirectory** method are shown in the following table.

UserDataDirectory Value	Return Value
<default>	The path of the JADE HOME directory with the **\Program Files** portion replaced with the programmatically obtained path for the specific user application private data directory. For example, a presentation client installed into **\Program Files\Jade Software\parsys** and executed by user **wilbur** returns **\Users\wilbur\AppData\Local\Jade Software\parsys**.
<homedir>	The path of the JADE HOME directory.
<userdata>	The same as for **<default>**.
Directory name	Directory name.

initializePackages

Signature `initializePackages() updating;`

The **initializePackages** method of the **Process** class calls the **Application** object **initialize** event to perform any initialize function common to all applications containing packages before the application start-up form is invoked.

Normally, if you run an application from the schema, these are executed automatically. However, they are not executed when a **JadeScript** or a Workspace method is executed unless you call the **initializePackages** event method. (This maintains these interfaces in as light a weight as possible.) JADE does not initialize any packages for **RootSchema** applications running in user schemas, including unit tests run using the JADE Unit Test framework and the default RPS **Datapump** application in the **RootSchema**. If you require this initialization, you must initialize packages in your application code (using the **Process** class **initializePackages** and **finalizePackages** methods).

If your schema contains packages and you run a **JadeScript** or a Workspace method and you do *not* call this event, the imported packages application **initialize** event methods are not called and methods such as the **Process** class **getAllApps** method can therefore not return information about packages.

See also the **Process** class **finalizePackages** method.

isCommitting

Signature `isCommitting(): Boolean;`

The **isCommitting** method of the **Process** class returns **true** if the receiver is currently committing a transaction.

isInExceptionState

Signature `isInExceptionState(): Boolean;`

The **isInExceptionState** method of the **Process** class returns **true** if the process is in exception state (that is, the exception has occurred in the current transaction); for example, in the epilog code or in an exception handler.

Note The method receiver must be the current process.

The code fragment in the following example shows the use of the **isInExceptionState** method.

```
epilog
    if process.isInExceptionState then
        abort;
    endif;
end;
```

isInImportedContext

Signature `isInImportedContext(): Boolean;`

The **isInImportedContext** method of the **Process** class returns **true** if the current (executing) process has invoked the current method from a package (that is, a method in a imported package is being executed) or it returns **false** if the current method is defined in the local schema branch of the main application (process).

Use this method to write conditional code based on whether a method is being executed by a process that *imports* the package or by an application running from the schema that *implements* the package (for example, in situations in which both exported classes and local applications share common code).

isInLoadState

Signature `isInLoadState(): Boolean;`

The **isInLoadState** method of the **Process** class returns **true** if the process is currently in load state; that is, it is between a pair of **beginLoad** and **endLoad** instructions.

Note The method receiver must be the current process.

isInLockState

Signature `isInLockState(): Boolean;`

The **isInLockState** method of the **Process** class returns **true** if the process is currently in lock state; that is, it is between a pair of **beginLock** and **endLock** instructions.

Note The method receiver must be the current process.

The following example shows the use of the **isInLockState** method.

```
unload() updating;
begin
    beginTransaction;
        delete global.a;
    commitTransaction;
    if process.isInLockState then
        endLock;
    endif;
end;
```

isInTransactionState

Signature `isInTransactionState(): Boolean;`

The **isInTransactionState** method of the **Process** class returns **true** if the process is currently in persistent transaction state.

Note The method receiver must be the current process.

The following example shows the use of the **isInTransactionState** method.

```
commitButton_click(btn: Button input) updating;
begin
    if process.isInTransactionState then
        commitTransaction;
        sl1.caption := 'Not in transaction state';
    endif;
end;
```

isInTransientTransactionState

Signature `isInTransientTransactionState(): Boolean;`

The **isInTransientTransactionState** method of the **Process** class returns **true** if the process is currently in transient transaction state.

Note The method receiver must be the current process.

isRunningScript

Signature `isRunningScript(): Boolean;`

The **isRunningScript** method of the **Process** class returns **true** if the process is running a **JadeScript** method or Workspace.

Note A network message is sent to the node on which the process is running. This overhead should be taken into account when using the method.

isUserDataPump

Signature `isUserDataPump(): Boolean;`

The **isUserDataPump** method of the **Process** class returns **true** if the process is running as a user-defined **Datapump** application on an RPS node.

isUsingThinClient

Signature `isUsingThinClient(): Boolean;`

The **isUsingThinClient** method of the **Process** class returns **true** if the process is running under the JADE thin client mode.

iteratorsExcludeOfflineObjects

Signature `iteratorsExcludeOfflineObjects(enable: Boolean): Boolean;`

The **iteratorsExcludeOfflineObjects** method of the **Process** class when called with the value of the **enable** parameter set to **true**, specifies that all iterators created by the calling process are to exclude objects stored in offline partitions from the iteration and takes effect on the next call to the **next** or **back** method.

This affects explicit collection iterators and **foreach** iterations over object collections executed by the calling process, including the **Class**::**instances** virtual collection.

The method returns the prior exclusion state, which user logic can restore, if required, when calls are nested.

networkAddress

Signature `networkAddress(): String;`

The **networkAddress** method of the **Process** class returns a string.

For a thin client process, the returned string contains the IP address of the presentation client (for example, **127.0.0.1** or **::1**).

For non-thin client processes, the contents of the returned string depend on the type of transport used for the connection to the database server.

When the transport is TCP/IP, the string contains the IP address used by the client for the connection to the database server; for example, **127.0.0.1** or **::1**.

When the transport is **JadeLocal**, the returned string is empty.

When the transport is **HPSM**, the returned string contains **"proc*NNNN*"**, where the ***NNNN*** value is the decimal number of the process at the other end of the connection.

profileMethod

Signature `profileMethod(m: Method;`
`b: Boolean);`

The **profileMethod** method of the **Process** class selects or deselects a method to be profiled for the target process used as the method receiver.

The target process can be any current process, including the requesting process itself or a process executing on another node.

You would usually call this method before method profiling is started for the target process using the **beginMethodProfiling** method with the **option** parameter set to two (**2**) or three (**3**), to indicate a subset of methods are to be profiled.

The **m** parameter is a **Method** object reference, which can be a JADE method or an external method. The **b** parameter indicates if the method is to be added to or removed from the list of nominated methods. If **true**, the method is added. If **false**, the method is removed.

You can call the method when method profiling is already in effect for the target process. Changes to the list of nominated methods take immediate effect.

Note It is recommended that when investigating application performance, only one of the JADE Profiler, JADE Monitor, or method profiling is used at any one time, as the results reported when any of these are combined is undefined.

The method has no effect if called to add a method that is already in the list of nominated methods. Similarly, it has no effect if removing a method that is not in the nominated list.

The list of nominated methods is retained until the **removeMethodProfileInfo** method is called or the target process terminates.

profiler

Signature `profiler(): JadeProfiler;`

The **profiler** method of the **Process** class returns a reference to the profiler for the receiving process. If no instance of the **JadeProfiler** class exists, it is created. (You therefore do not have to create a profiler instance nor delete it in an epilog at the end of your method.)

prohibitBeginTransaction

Signature `prohibitBeginTransaction(prohibited: Boolean): Boolean updating;`

Set the **prohibited** parameter in the **prohibitBeginTransaction** method of the **Process** class to **true** if you do not want the current process to be able to enter persistent transaction state. If transactions are prohibited (or allowed) for a method executing on a client node and a **serverExecution** method is called, transactions will also be prohibited (or allowed) on the server node. This also applies when method execution switches from the server to the client.

To enable the process to enter transaction state, this method must be called with the **prohibited** parameter set to **false**.

The **prohibitBeginTransaction** method returns **true** only if the current process was already prohibiting entry into persistent transaction state.

Note You can call the **prohibitBeginTransaction** method only for the current process.

prohibitPersistentUpdates

Signature `prohibitPersistentUpdates(prohibited: Boolean): Boolean updating;`

Set the **prohibited** parameter in the **prohibitPersistentUpdates** method of the **Process** class to **true** if you do not want persistent objects in the current process to be updated. When the updating of persistent objects is prohibited in the current process, any attempt to do so raises a 1271 exception (that is, *An attempt was made to perform an operation that is prohibited in this context*). If the updating of persistent objects is prohibited (or allowed) for a method executing on a client node and a **serverExecution** method is called, the updating of persistent objects will also be prohibited (or allowed) on the server node. This also applies when method execution switches from the server to the client.

When the updating of persistent objects in the process of the receiver is prohibited, the following operations are also prohibited.

- Creating or deleting a persistent object
- Cloning or copying to a persistent object
- Executing the **beginTransaction** instruction
- Executing the **commitTransaction** instruction

When the updating of persistent objects in the process is prohibited, setting the value of the **prohibited** parameter to **false** removes the prohibition and allows persistent objects to once again be updated.

The **abortTransaction** instruction can still be executed, even if the prohibition is in place. Executing the **abortTransaction** instruction removes the prohibition.

The **prohibitPersistentUpdates** method returns **true** only if the current process was already prohibiting the updating of persistent objects.

Notes This method applies only to the current process.

The **prohibitPersistentUpdates** method is designed to prevent unexpected updates to persistent objects. It is not intended to be a comprehensive security measure.

removeMethodProfileInfo

Signature `removeMethodProfileInfo();`

The **removeMethodProfileInfo** method of the **Process** class removes all method profiling information and any list of nominated methods for the target process specified as the method receiver.

The target process can be any current process, including the requesting process itself or a process executing on another node.

If method profiling is in effect for the target process, it is stopped before the profiling information is removed. If method profiling is not active for the target process or there is no method profiling information, this method has no effect.

resumeTimers

Signature `resumeTimers();`

The **resumeTimers** method of the **Process** class resumes all timers suspended for the receiver using the **suspendTimers** method.

rpsSuppressTransactionDeletes

Signature `rpsSuppressTransactionDeletes();`

The **rpsSuppressTransactionDeletes** method of the **Process** class specifies on a primary system those transactions for which deletions are *not* to be replicated to the relational database by an RPS **Datapump** application on a secondary system. This allows selected operations on the primary system (for example, housekeeping, and archiving tasks) to be performed without affecting relational database replicas.

The **rpsSuppressTransactionDeletes** method is called after executing the **beginTransaction** instruction and before executing the **commitTransaction** instruction; it does not have to precede the first object deletion, as shown in the following code fragment.

```
beginTransaction;
    delete obj1;
    process.rpsSuppressTransactionDeletes;
    delete obj2;
commitTransaction;  // Delete obj1 and obj2 not replicated in relational db
```

When the process exits transaction state, replication of deletes is no longer suppressed. An exception is raised if the method is called outside transaction state.

sendCallStackInfo

Signature `sendCallStackInfo();`

The **sendCallStackInfo** method of the **Process** class requests a target process (the method receiver) to send one or more notifications containing information regarding its call stack.

The target process can be any current process, including the requesting process itself or a process executing on another node.

The information received is the same as that returned by the **getCallStackInfo** method of the **Process** class, although that method can be used only for the current process instance.

The target process is activated temporarily or interrupted in order to retrieve the call stack information and send the notifications, after which it resumes whatever it was doing.

Note The **sendCallStackInfo** method is asynchronous; that is, it does not wait until the information is received. The information is received through notifications some time after the method is called.

The information in the notification relating to the call stack for the target process is shown in the following table.

Parameter	Contains …
eventType	**Process_Call_Stack_Info_Event**
target	Process instance of the process that made the request
userInfo	Call stack information stored within a string value

Register to receive notifications of events of type **Process_Call_Stack_Info_Event** (a global constant in the **JadeProcessEvents** category) on the process making the request using the **beginNotification** method defined in the **Object** class before executing the **sendCallStackInfo** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Call_Stack_Info_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains call stack information, the **userNotification** method should test whether the value of the **eventType** parameter is **Process_Call_Stack_Info_Event**.

If the size of the call stack information collected is greater than the maximum allowed for a notification, the information is broken into parts and sent using multiple notifications. Each delivered notification records positional information at the end of the string, in the format [*<process oid>*:*<current notification>*:*<total number of notifications>*] (for example, **[187.5:1:3]**).

If the target process (the method receiver) is not a valid current process, an 1128 exception (*The target process is not valid*) is raised.

sendMethodCacheStatistics

Signature `sendMethodCacheStatistics() serverExecution;`

The **sendMethodCacheStatistics** method of the **Process** class requests a target process (the receiver) to send a notification containing statistics about the method cache of the target process.

The target process can be any current process, including the requesting process itself or a process executing on another node.

To retrieve the method cache statistics information and send the notifications, the target process is activated temporarily or interrupted, after which it resumes whatever it was doing.

Note The **sendMethodCacheStatistics** method is asynchronous; that is, it does not wait until the information is received. The information is received through notifications sometime after the method is called.

The information in the notification relating to the method cache for the target process is shown in the following table.

Parameter	Contains …
eventType	**Process_Method_Cache_Stats_Event** global constant

Parameter	Contains …
target	Process instance of the process that made the request
userInfo	Method cache statistics stored within a string value

The process making the request should register to receive type **Process_Method_Cache_Stats_Event** notifications on its **Process** instance, using the **beginNotification** method defined in the **Object** class before executing the **sendMethodCacheStatistics** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Method_Cache_Stats_Event,
                       Response_Continuous, 0);
```

To determine whether a notification contains method cache statistics, the **userNotification** method of its **Process** instance should test whether the value of the **eventType** parameter is **Process_Method_Cache_Stats_Event**.

The following is an example of the **userInfo** output.

```
OID: 187.2
Process method cache type: Multiple (Separate per process)
Cache limit: 524288
Maximum cache size: 524288
Number of methods in cache: 105
Total methods discarded: 2560
Total methods executed: 190586
Total time loading methods into cache (ms): 2237
Cache overruns: 0
String pool limit: 5242880
Maximum string pool size: 173568
Number of Strings in string pool: 234
String pool overruns: 0
Number of cache overruns 0-10% over limit: 0
Number of cache overruns 10-20% over limit: 0
Number of cache overruns 20-30% over limit: 0
Number of cache overruns 30-40% over limit: 0
Number of cache overruns 40-50% over limit: 0
Number of cache overruns 50-60% over limit: 0
Number of cache overruns 60-70% over limit: 0
Number of cache overruns 70-80% over limit: 0
Number of cache overruns 80-90% over limit: 0
Number of cache overruns 90-100% over limit: 0
Number of cache overruns 100%+ over limit: 0
[187.2:1:1]
```

sendRequestStatistics

Signature `sendRequestStatistics(localOrRemote: Integer);`

The **sendRequestStatistics** method of the **Process** class requests a target process to send a notification containing local or remote request statistics.

The target process can be any current process, including the requesting process itself or a process executing on another node.

The target object for the notification is the **Process** instance of the process making the request.

To request local statistics, which record information about requests made on the node on which the process is running, set the value of the **localOrRemote** parameter to one (**1**). To request remote statistics, which record information about requests made between the client node and the database server node, set the value of the **localOrRemote** parameter to two (**2**).

Most of the request statistics are collected only when node sampling is active on the node. The values that are reported independent of node sampling are the thin client statistics that are part of the local request statistics.

The target process is temporarily activated or interrupted to send the notification. After sending the notification, it resumes whatever it was previously doing.

Note This method is asynchronous; that is, the **sendRequestStatistics** method does not wait until the statistics have been received. The statistics are received as a notification some time after the **sendRequestStatistics** method has been called.

The information in the notification relating to the request statistics is shown in the following table.

Parameter	Contains …
eventType	**Process_Local_Stats_Event** for local request statistics **Process_Remote_Stats_Event** for remote request statistics
target	Process instance of the process that made the request
userInfo	Statistical values encoded within a **Binary** value

The process making the request should register to receive type **Process_Local_Stats_Event** (local request statistics) or **Process_Remote_Stats_Event** (remote request statistics) notifications on its **Process** instance using the **beginNotification** method defined in the **Object** class before executing the **sendRequestStatistics** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Local_Stats_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains local or remote request statistical information, the **userNotification** method of its **Process** instance should test whether the value of the **eventType** parameter is **Process_Local_Stats_Event** or **Process_Remote_Stats_Event**, which indicates a local or remote request statistics notification.

The **userInfo** parameter of the notification should then be passed as a parameter to the **extractRequestStatistics** method, to extract the local and remote request statistics as attributes in a **JadeDynamicObject** instance.

If the target process (the method receiver) is not a valid current process, an 1128 exception (*The target process is not valid*) is raised.

sendTransientFileAnalysis

Signature `sendTransientFileAnalysis();`

The **sendTransientFileAnalysis** method of the **Process** class requests a target process (the method receiver) to send one or more notifications containing detailed analysis of its transient database file, including counts of objects by class number plus other useful information.

The target process can be any current process, including the requesting process itself or a process executing on another node.

The information received is the same as that returned by the **analyzeTransientFileUsage** method of the **Process** class, although that method can be used only for the current process instance.

The target process is activated temporarily or interrupted in order to analyze the transient database file and send the notifications, after which it resumes whatever it was doing.

Note The **sendTransientFileAnalysis** method is asynchronous; that is, it does not wait until the information is received. The information is received through notifications some time after the method is called.

The information in the notification relating to analysis of the transient database file for the target process is shown in the following table.

Parameter	Contains …
eventType	**Process_TDB_Analysis_Event**
target	Process instance of the process that made the request
userInfo	Detailed analysis of the transient database file for the process

Register to receive notifications of events of type **Process_TDB_Analysis_Event** (a global constant in the **JadeProcessEvents** category) on the process making the request using the **beginNotification** method defined in the **Object** class before executing the **sendTransientFileAnalysis** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_TDB_Analysis_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains transient database file analysis, the **userNotification** method should test whether the value of the **eventType** parameter is **Process_TDB_Analysis_Event**.

If the size of the analyzed data collected is greater than the maximum allowed for a notification, the information is broken into parts and sent in a number of notifications.

Each delivered notification records positional information at the end of the string, in the format [<*process oid*>:<*current notification*>:<*total number of notifications*>] (for example, **[187.5:1:3]**).

If the target process (the method receiver) is not a valid current process, an 1128 exception (*The target process is not valid*) is raised.

sendTransientFileInfo

Signature `sendTransientFileInfo();`

The **sendTransientFileInfo** method of the **Process** class requests a target process (the method receiver) to send a notification containing information regarding its transient database file.

The target process can be any current process, including the requesting process itself or a process executing on another node.

The target process is activated temporarily or interrupted in order to retrieve the transient database file information and send the notifications, after which it resumes whatever it was doing.

Note This **sendTransientFileInfo** method is asynchronous; that is, the method does not wait until the information is received. The information is received as a notification some time after the method is called.

The information in the notification relating to analysis of the transient database file for the target process is shown in the following table.

Parameter	Contains…
eventType	**Process_TDB_Info_Event**
target	Process instance of the process that made the request
userInfo	A colon delimited string containing the process oid, transient database file name, and the transient database file length

Register to receive notifications of events of type **Process_TDB_Info_Event** (a global constant in the **JadeProcessEvents** category) on the process making the request using the **beginNotification** method defined in the **Object** class before executing the **sendTransientFileInfo** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_TDB_Info_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains transient database file information, the **userNotification** method should test whether the value of the **eventType** parameter is **Process_TDB_Info_Event**.

If the target process (the method receiver) is not a valid current process, an 1128 exception (*The target process is not valid*) is raised.

sendWebStatistics

Signature `sendWebStatistics();`

The **sendWebStatistics** method of the **Process** class requests a target process to send a notification containing World Wide Web performance statistics.

The target process can be any current process, including the requesting process itself or a process executing on another node.

The target object for the notification is the **Process** instance of the process making the request.

Only processes using Web services send meaningful data. Other processes send the notification, but it does not contain any values.

The target process is temporarily activated or interrupted in order to send the notification. After sending the notification, it resumes whatever it was previously doing.

Note This method is asynchronous; that is, the **sendWebStatistics** method does not wait until the statistics have been received. The statistics are received as a notification some time after the **sendWebStatistics** method has been called.

The information in the notification relating to the Web performance statistics is shown in the following table.

Parameter	Contains…
eventType	**Process_Web_Stats_Event** for Web statistics
target	Process instance of the process that made the request
userInfo	Statistical values encoded within a **Binary** value

The process making the request should register to receive type **Process_Web_Stats_Event** (a global constant in the **JadeProcessEvents** category) notifications on its **Process** instance using the **beginNotification** method defined in the **Object** class before executing the **sendWebStatistics** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Web_Stats_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains Web statistics information, the **userNotification** method of its **Process** instance should test whether the value of the **eventType** parameter is **Process_Web_Stats_Event**, which indicates a Web statistics notification.

The **userInfo** parameter of the notification should be passed as a parameter to the **extractWebStatistics** method, to extract the Web statistics as attributes in a **JadeDynamicObject** instance.

If the target process (the method receiver) is not a valid current process, an 1128 exception (*The target process is not valid*) is raised.

setDateTimeDelta

Signature `setDateTimeDelta(deltaDate: Integer;`
`deltaTime: Integer);`

The **setDateTimeDelta** method of the **Process** class sets the values used to adjust initial values of the **Date**, **Time**, and **TimeStamp** local variables.

The value of **deltaDate** parameter specifies the number of days by which to adjust the initial value of any **Date** local variable and the **Date** component of a **TimeStamp** local variable.

The value of the **deltaTime** parameter the number of milliseconds by which to adjust the initial value of any **Time** local variable and the **Time** component of a **TimeStamp** local variable.

Calling the **setDateTimeDelta** method with parameter values of zero (**0**) causes all local variables to be initialized with the current date and time. If you want to set the date or time to a value in the past, specify a negative value in the appropriate parameter.

You can define an adjustment to the initial date or time value for local variables only for the current process.

For details about retrieving the values used to adjust the initial date and time used by the receiving process, see the **getDateTimeDelta** method.

setDefaultLockTimeout

Signature `setDefaultLockTimeout(timeout: Integer): Integer;`

The **setDefaultLockTimeout** method of the **Process** class programmatically changes the default lock timeout period for the receiving process.

The value of the **timeout** parameter can be an explicit number of milliseconds (for example, specify a value of **30000** for 30 seconds), or it can be one of the predefined global constants defined in the RootSchema **LockTimeouts** category; that is, one of:

- LockTimeout_Immediate
- LockTimeout_Infinite
- LockTimeout_Server_Defined

When the **setDefaultLockTimeout** method is called, the parameter value is used for the timeout for all implicit locks and locks obtained by the **Object** class **exclusiveLock**, **sharedLock**, **reserveLock**, and **updateLock** methods.

This method returns the current process lock timeout (before the new value specified in the **timeout** parameter is saved).

By default (that is, if you do not call this method), the default lock timeout for all processes is defined by the value of the **ServerTimeout** parameter in the [JadeServer] section of the JADE initialization file.

setMethodCacheLimit

Signature `setMethodCacheLimit(limit: Integer64);`

The **setMethodCacheLimit** method of the **Process** class programmatically sets the method cache limit for the executing process. The **limit** parameter represents the number of bytes to which to set the method cache limit.

The **limit** value must be greater than the minimum allowed cache size (64K bytes); otherwise exception 1002 (*Invalid parameter value*) is raised.

This method can be called only on the process instance of the current process. Exception 1265 (*Environmental object operation is out of scope for process*) is raised if you call the methods on an instance for another process.

setObjectCachePriority

Signature
```
setObjectCachePriority(obj:      Object;
                       priority: Integer);
```

The **setObjectCachePriority** method of the **Process** class specifies, through the **priority** parameter, how long an object, specified by the **obj** parameter, is to be retained in persistent or transient object cache. The greater the value of the **priority** parameter, the longer an object remains in the object cache.

The **priority** parameter effectively gives an object a number of *lives* in cache. When an object in cache has not been used for the specified length of time, it becomes a candidate to be removed from cache. Its number of lives is examined. If equal to one (**1**), the object is removed from cache. If greater than one (**1**), the number of lives is decremented and instead of being removed from cache, the object is treated as if it had just been accessed. This results in it being retained longer in cache, instead of being removed.

Conversely, when the number of lives for an object is set to zero (**0**), it is removed from cache.

The range of values for the **priority** parameter is zero (**0**) through **255**. A negative value is treated as (**0**), and a value greater than **255** is treated as **255**.

The number of lives an object has applies only while the object is in cache. When an object is first loaded into cache, it is assigned one life only. Lives are not recorded for objects that are not in cache.

If the value of the **priority** parameter is greater than zero (**0**), the **setObjectCachePriority** method loads the object into cache if it is not already present, before setting the number of lives. For values greater than one (**1**), this results in an extension to the length of time the object is retained in cache.

If the value of the **priority** parameter is zero (**0**), the **setObjectCachePriority** method removes the object from cache immediately, if it is currently in cache. If it is not in cache, the method has no effect. When an object is removed, its subobjects are also removed, including string large objects (slobs) and binary large objects (blobs) but not *exclusive* collections, which must be removed separately, as shown in the following example.

```
/* remove obj1 from cache as well as its slobs and blobs,
   but not its // exclusive collections */
process.setObjectCachePriority(obj1, 0);
```

```
/* remove the exclusive collection obj2.allObj3s from cache
and any of its collection blocks */
process.setObjectCachePriority(obj2.allObj3s, 0);
```

Note An object is not removed from the cache if it is currently being updated by another process (that is, it contains uncommitted updates).

You can use the **setObjectCachePriority** method with persistent and transient objects; that is, it applies to persistent and transient object caches. With transient objects, a process can only affect shared transient objects and its own non-shared transient objects.

A process must use its own **Process** instance as the method receiver. Using any other **Process** instance causes a 1265 exception (*Environmental object operation is out of scope for process*) to be raised.

In the following code fragment, a report application iterates a collection and accesses objects in the collection once only. The **setObjectCachePriority** method is used to remove each object from cache immediately after it has been accessed.

```
foreach cust in root.allCusts do
    totalSales := totalSales + cust.purchases;
    process.setObjectCachePriority(cust, 0);
endforeach;
```

setPersistentDeadlockPriority

Signature `setPersistentDeadlockPriority(priority: Integer);`

The **setPersistentDeadlockPriority** method of the **Process** class sets the priority value to be used when dealing with deadlocks involving persistent objects. Negative and positive values are allowed.

The default deadlock priority is zero (**0**).

In a deadlock situation, the process with the lower value is given the deadlock exception. This method can be called only on the process instance of the current process. If you call it on an instance for another process, an exception is raised.

See also the **DoubleDeadlockException** parameter in the [JadeServer] section of the JADE initialization file, in your *JADE Initialization File Reference*.

setProfileString

Signature
```
setProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 string:   String): Boolean;
```

The **setProfileString** method of the **Process** class copies a parameter (key name) string into the specified section of the initialization file of the process.

A method on a specific process instance performs its action on the owning node (that is, a **process.node** instance) if the process is not associated with a presentation client. If the process has an associated presentation client, the action is performed on the presentation client. The presentation client does not have to be the current presentation client or a presentation client attached to the same application server.

Use the **Application** class **setProfileStringAppServer** method or **Node** class **setProfileString** method to set the string on the application server.

This method returns **true** if it succeeds in storing the specified string. Conversely, if the value of the **section** or **keyName** parameter is **null** (**""**) or empty, this method returns **false**, to indicate that the JADE initialization file has not been updated. Use the respective **ProfileRemoveSection** or **ProfileRemoveKey** global constant in the **JadeProfileString** category to delete a section or key, rather than passing a **null** or empty string in the appropriate parameter of this method.

To retrieve a stored string, use the **getProfileString** method.

The parameters for the **setProfileString** method are listed in the following table.

Parameter	Description
fileName	Specifies the initialization file. If you set this parameter to **windows**, the **win.ini** file is used. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory.
section	Specifies the initialization file section containing the key (parameter) name.
keyName	Specifies the name of the key (parameter) whose associated string is to be stored.
string	Specifies the string that is to be written to the file.

You can use this method to copy a string to a two-level section name (prefixed with a unique identifier) within a JADE initialization file shared by multiple programs on the host. For details, see "Two-Level Section Names" under "Format of the JADE Initialization File", in the *JADE Initialization File Reference*. However, you cannot use this method to update JADE initialization file parameter values specified on the command line. Attempts to do so return a value of **false** and the parameter values are unchanged.

The following example shows the use of this method to remove an entire [mySection] section and the **WindowPos** parameter in the [InternalAS.JadeAppServer] section from the JADE initialization file.

```
begin
    process.setProfileString(process.getIniFileName, "mySection",
                             ProfileRemoveSection, "");
    // If the user has moved the window, reset it to the default values
    process.setProfileString(process.getIniFileName, "JadeAppServer",
                             "WindowPos", ProfileRemoveKey);
end;
```

setSaveLockCallStack

Signature `setSaveLockCallStack(saveLockCallStack: Boolean): Boolean;`

The **setSaveLockCallStack** method of the **Process** class specifies whether JADE saves the current call stack information when a process locks an object.

If the value of the **saveLockCallStack** parameter is **true**, the call stack is saved when the receiving process locks an object. A parameter value of **false** stops saving this information. The method returns **true** if the call stack was being saved before calling this method; otherwise it returns **false**.

Calling this method overrides the value of the **DefaultProcessSaveLockCallStack** parameter in the [JadeClient] section and the value of the **DefaultProcessSaveLockCallStack** parameter in the [JadeServer] section of the initialization file.

setStringPoolLimit

Signature `setStringPoolLimit(limit: Integer64);`

The **setStringPoolLimit** method of the **Process** class programmatically sets the string pool limit for the executing process. The **limit** parameter represents the number of bytes to which to set the string pool limit.

The **limit** value must be greater than the minimum allowed string pool size (64K bytes); otherwise exception 1002 (*Invalid parameter value*) is raised.

This method can be called only on the process instance of the current process. Exception 1265 (*Environmental object operation is out of scope for process*) is raised if you call the methods on an instance for another process.

setTransientDeadlockPriority

Signature `setTransientDeadlockPriority(priority: Integer);`

The **setTransientDeadlockPriority** method of the **Process** class sets the priority value to be used when dealing with deadlocks involving shared transient objects. Negative and positive values are allowed. The default deadlock priority is zero (**0**).

In a deadlock situation, the process with the lower value is given the deadlock exception.

This method can be called only on the process instance of the current process. If you call it on an instance for another process, an exception is raised.

See also the **DoubleDeadlockException** parameter in the [JadeServer] section of the JADE initialization file, in the *JADE Initialization File Reference*.

sleep

Signature `sleep(sleepTime: Integer);`

The **sleep** method of the **Process** class suspends the execution of the thread of the receiver process for the specified time interval.

The following example shows the use of the **sleep** method.

```
defaultInitialize();
vars
    ini  : String;
    bin  : String;
    size : Integer;
begin
    size := system.nodes.size;
    ini  := app.getIniFileName;
    bin  := app.getJadeInstallDir;
    app.createExternalProcess(bin & '\jade.exe path = ' & app.dbPath &
                              ' app = PrintTest schema = PrintTest', false);
    while size = system.nodes.size do
        process.sleep(2000);
    endwhile;
end;
```

Use the **sleepTime** parameter to specify the time (in milliseconds) that the thread is to be suspended. A value of zero (**0**) causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run in the system. If there are no other threads that are ready to run, the receiver process continues execution immediately.

Note Unlike the **Application** class **doWindowEvents** method, the **sleep** method does not allow Window events to be processed while the thread is suspended.

startMethodTracking

Signature

```
startMethodTracking(targetMethod:    Method;
                    preambleMethod:  Method;
                    postambleMethod: Method;
                    receiver:        Object);
```

The **startMethodTracking** method of the **Process** class initiates method tracking for the receiving process by automatically invoking a specified method just before the specified target method is called and invoking another method just after the target method has returned from execution.

The receiving **Process** instance can be any current process including the current process. The parameters of the **startMethodTracking** method are listed in the following table.

Parameter	Description
targetMethod	Method to be tracked
preambleMethod	Method to be invoked just before calling the target method
postambleMethod	Method to be invoked just after returning from executing the target method (that is, after any epilog code in the target method has been executed)
receiver	Receiver for the preamble and postamble methods

The **preambleMethod** and **postambleMethod** methods must have the following signature:

```
method(paramList: ParamListType);
```

When invoked, the **paramList** parameter contains a list of parameters matching those of the method being tracked.

The following methods cannot be tracked.

- **getAndValidateUser** in the **Global** class or a reimplementation in a subclass
- **isUserValid** in the **Global** class or a reimplementation in a subclass

Method tracking is not currently supported for **serverExecution** methods.

Note To avoid repeated calls and kernel stack overflow exceptions, the tracking method should not track itself or any of the methods it calls.

For more details about method tracking, see Chapter 17, "Tracking Methods", in the *JADE Developer's Reference*.

startTransactionTrace

Signature `startTransactionTrace();`

The **startTransactionTrace** method of the **Process** class initiates transaction tracing for transactions carried out by the receiving **Process** instance, which must be the current process. A transient instance of the **JadeTransactionTrace** class is created for the process, if one did not already exist, to store transaction information.

Transaction tracing can be started regardless of the current transaction state. If started while a transaction is active, only objects updated, created, and deleted after tracing is initiated are recorded. If the **startTransactionTrace** method is called when transaction tracing is already active, an exception is raised.

Existing trace information in the current transaction trace object is removed if the **startTransactionTrace** method is called when not in transaction state, or when the information does not relate to the current transaction. Stopping and starting transaction tracing within a transaction does not remove tracing information.

stopMethodTracking

Signature `stopMethodTracking(targetMethod: Method);`

The **stopMethodTracking** method of the **Process** class turns off method tracking of the method specified in the **targetMethod** parameter by the receiving process, which can be any process including the current process. The instruction is ignored if the method specified by the **targetMethod** parameter is not being tracked.

For more details about method tracking, see Chapter 17, "Tracking Methods", in the *JADE Developer's Reference*.

stopTransactionTrace

Signature `stopTransactionTrace();`

The **stopTransactionTrace** method of the **Process** class turns off transaction tracing for the receiving **Process** instance, which must be the current process.

Transaction tracing can be stopped regardless of the current transaction state. You can stop and start transaction tracing multiple times within a transaction.

Calling the **stopTransactionTrace** method does not remove trace information from the current transaction trace object. The information is cleared automatically when tracing is next started outside of transaction state or within a subsequent transaction. The information can also be cleared by calling the **clear** for the **JadeTransactionTrace** object.

If the **stopTransactionTrace** method is called when transaction tracing is not active, an exception is raised.

suspendTimers

Signature `suspendTimers();`

The **suspendTimers** method of the **Process** class suspends all timers registered for the receiver. Use the **resumeTimers** method of the **Process** class to resume suspended timers.

transactionTraceStarted

Signature `transactionTraceStarted(): Boolean;`

The **transactionTraceStarted** method of the **Process** class enables an application to determine whether transaction tracing is currently enabled. The receiving **Process** instance must be the current process.

If transaction tracing has been started by calling the **startTransactionTrace** method of the **Process** class and not yet stopped by calling the **startTransactionTrace** method, **true** is returned. If transaction tracing has not been started or has been stopped and not yet restarted, **false** is returned.

transientPersistentInvsEnabled

Signature `transientPersistentInvsEnabled(): Boolean;`

The **transientPersistentInvsEnabled** method of the **Process** class returns the current state of the **Boolean** value set by calls to **allowTransientToPersistentInvs** on the process.

transientSharedTranInvsEnabled

Signature `transientSharedTranInvsEnabled(): Boolean;`

The **transientSharedTranInvsEnabled** method of the **Process** class returns the current state of the **Boolean** value set by calls to **allowTransientToSharedTranInvs** on the process.

truncateOnDecimalOverflow

Signature `truncateOnDecimalOverflow(bool: Boolean);`

The **truncateOnDecimalOverflow** method of the **Process** class specifies whether an exception is raised when a decimal overflow occurs.

When you set the **bool** parameter to **true** and an exception overflow occurs, the exception that is raised (exception 4043) is continuable.

The following example shows the use of the **truncateOnDecimalOverflow** method.

```
vars
    d : Decimal[4,2];
begin
    on Exception do e0(exception);
    process.truncateOnDecimalOverflow(true);
    d := 123.456;
    write d;
    d := -123.456;
    write d;
end;
```

The following example shows the use of the **extendedErrorText** property in an exception handler that deals with decimal truncation

```
e0(e: Exception): Integer;
begin
    // exception 4043 is a 'continuable' decimal overflow
    if e.errorCode = 4043 then
        write e.extendedErrorText;
```

```
            // continuing here will result in the value being truncated
            return Ex_Continue;
        endif;
        return Ex_Resume_Next;
    end;
```

This method results in the output of the two **write** instructions in the first of these examples being **123.456**, **23.46**, **-123.456**, and **23.46**, respectively.

Note It is the integral part that is truncated. The fractional part is rounded.

If an exception handler continues this exception, the decimal number is truncated and execution continues; for example, **123.456** is truncated to **23.46** when assigned to a decimal **4,2** (the integral digits are truncated and the fractional digits are rounded). The **Exception** class **extendedErrorText** property contains the value of the decimal before it is truncated.

useUpdateLocks

Signature `useUpdateLocks(b: Boolean);`

The **useUpdateLocks** method of the **Process** class automatically enables **update** locks on objects. Set the value of the **b** parameter to **true** if you want the automatic lock applied when an object is first updated to be an **update** lock rather than an **exclusive** lock. Set the parameter to **false** if you want to disable the use of the **update** lock.

Note If you do not enable **update** locks, it is still possible to apply an **update** lock explicitly to an object by using the **updateLock** method of the **Object** class or the **lock** method specifying **Update_Lock** for the **lockType** parameter.

waitForMethods

Signature `waitForMethods(methodContextList: ParamListType): JadeMethodContext;`

The **waitForMethods** method of the **Process** class suspends the process until one of the method contexts specified by the **methodContextList** parameter completes or times out. The **waitForMethods** method returns a reference to the method context that completes or times out. If all method contexts have completed or timed out, the **waitForMethods** method returns a **null** value.

The **methodContextList** parameter value consists of one or more references to instances of the **JadeMethodContext** or **ObjectArray** class. An **ObjectArray** instance must contain **JadeMethodContext** references only. The combined context list must not have more than 64 active entries.

For more details, see Chapter 16, "Using Asynchronous Method Calls", in the *JADE Developer's Reference*.

ProcessDict Class

The **ProcessDict** class is the persistent class that encapsulates the behavior required to access process objects in a dictionary.

The key of the **ProcessDict** class is the user code of the user. The user code strings are sorted in binary order.

Inherits From: MemberKeyDictionary

Inherited By: (None)

ProcessStackArray Class

The **ProcessStackArray** class is the transient class that encapsulates behavior required to access method calls in the process stack array.

The process stack array is populated with references to method call descriptor objects by the **currentStack** method of the **Process** class and represents a snapshot of the current execution history of the application thread of the current process.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a process stack array.

Inherits From: ObjectArray

Inherited By: (None)

RealArray Class

The **RealArray** class is an ordered collection of **Real** values in which the values are referenced by their position in the collection.

Real arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a **Real** array.

Inherits From: Array

Inherited By: (None)

Rectangle Class

The **Rectangle** class encapsulates the behavior required to store the dimensions of a rectangle.

For details about the properties and methods defined in the **Rectangle** class, see "Rectangle Properties" and "Rectangle Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

Rectangle Properties

The properties defined in the **Rectangle** class are summarized in the following table.

Property	Description
bottom	Contains the bottom coordinate of the rectangle
left	Contains the left coordinate of the rectangle
right	Contains the right coordinate of the rectangle
top	Contains the top coordinate of the rectangle

Use the **Rectangle** class **display** method if you want to return a string containing these coordinate values.

bottom

Type: Real

The **bottom** property of the **Rectangle** class contains the bottom coordinate of the rectangle, in units.

left

Type: Real

The **left** property of the **Rectangle** class contains the left coordinate of the rectangle, in units.

right

Type: Real

The **right** property of the **Rectangle** class contains the right coordinate of the rectangle, in units.

top

Type: Real

The **top** property of the **Rectangle** class contains the top coordinate of the rectangle, in units.

Rectangle Methods

The methods defined in the **Rectangle** class are summarized in the following table.

Method	Description
copy	Copies the coordinates of the specified rectangle
display	Returns a string containing the left, top, right, and bottom coordinates of the rectangle
isEmpty	Specifies if the rectangle is empty
set	Sets the coordinates of the rectangle

copy

Signature `copy(rect: Rectangle) updating;`

The **copy** method of the **Rectangle** class copies the coordinates of the rectangle specified in the **rect** parameter to **self**.

display

Signature `display(): String;`

The **display** method of the **Rectangle** class returns a string containing the left, top, right, and bottom coordinates of the rectangle, respectively.

isEmpty

Signature `isEmpty(): Boolean;`

The **isEmpty** method of the **Rectangle** class returns **true** if the width or height of the rectangle is zero (**0**).

set

Signature
```
set(lft:  Real;
    tp:   Real;
    rght: Real;
    bttm: Real) updating;
```

The **set** method of the **Rectangle** class sets the values of the receiver to the coordinates specified in the parameters listed in the following table.

Parameter	Description
lft	Left point of the rectangle
tp	Top point of the rectangle
rght	Right point of the rectangle
bttm	Bottom point of the rectangle

RelationalView Class

The **RelationalView** class represents an RPS mapping or an ODBC relational view. For details about RPS mappings, see Chapter 2, "Relational Population Service (RPS) Support", in the *JADE Synchronized Database Service (SDS) Administration Guide*. For details about relational views, see Chapter 9, "Defining ODBC Inquiry Relational Views and Ad Hoc Indexes", in the *JADE Development Environment User's Guide*.

For details about the constants, properties, and methods defined in the **RelationalView** class, see "RelationalView Class Constants", "RelationalView Properties", and "RelationalView Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

RelationalView Class Constants

The constants provided by the **RelationalView** class are listed in the following table.

Class Constant	Integer Value
DatabaseType_RelationalView	0
DatabaseType_RpsNodeUseDefault	0
DatabaseType_SqlServer2000	1
DatabaseType_SqlServer2005	2
DatabaseType_SqlServer2008	3
ExceptionLogging_Default	0
ExceptionLogging_Invalid	2
ExceptionLogging_RPSTable	1
Exception_Alternate	1
Exception_Halt	0
ExtractOrderClassInstances	1
ExtractOrderDefault	0
ExtractOrderSelectedFirst	2
Load_ClientExecute	1
Load_ServerExecute	0
RpsScript_SelectedTables	2
RpsScript_UserAndJadeTables	1
RpsScript_UserTablesOnly	0

RelationalView Properties

The properties defined in the **RelationalView** class are summarized in the following table.

Property	Description
creator	Contains the user name of the process that created the RPS mapping or relational view
name	Contains the name of the RPS mapping or relational view
rpsDatabaseName	Contains the default name of the RDBMS database
rpsDatabaseType	Contains the type of the RDBMS database
rpsDefaultConnectionString	Contains the default RDBMS connection string
rpsDefaultPassword	Contains the default RDBMS log-in password
rpsDefaultUserName	Contains the default RDBMS log-in user name
rpsExceptionCreate	Specifies what happens when an exception is raised during an RDBMS create
rpsExceptionDelete	Specifies what happens when an exception is raised during an RDBMS delete
rpsExceptionUpdate	Specifies what happens when an exception is raised during an RDBMS update
rpsLoggingOptions	Specifies the logging option for the RDBMS database
rpsShowMethods	Specifies whether column mapping methods were displayed when the mapping was defined
rpsShowVirtualProperties	Specifies whether virtual properties are displayed when the mapping was defined
rpsTopSchemaName	Specifies the highest-level superschema for the RPS mapping
rpsUseOidClassInstMap	Specifies whether oids are mapped to two columns: class number and instance identifier
schema	Contains a reference to the schema in which the RPS mapping or relational view is defined
timeCreated	Contains the timestamp of the creation of the relational view or RPS mapping

creator

Type: String[30]

The **creator** property of the **RelationalView** class contains the name of the user who created the RPS mapping or relational view.

name

Type: String[100]

The **name** property of the **RelationalView** class contains the name of the RPS mapping or relational view.

rpsDatabaseName

Type: String[128]

The **rpsDatabaseName** property of the **RelationalView** class contains the default name of the RDBMS database.

rpsDatabaseType

Type: Integer

The **rpsDatabaseType** property of the **RelationalView** class contains the intended use of the relational view; that is, whether it is used by the JADE ODBC driver for third-party relation access to the JADE database or whether it is used by the JADE RPS to replicate data in an RDBMS database.

The **rpsDatabaseType** property values are described in the following table:

RelationalView Class Constant	Value	Description
DatabaseType_RelationalView	0	Not an RPS mapping. Used to access a JADE database using the ODBC driver.
DatabaseType_RpsNodeUseDefault	0	Uses the default RPS node.
DatabaseType_SqlServer2000	1	RPS mapping used to replicate data to a SQL Server 2000 database.
DatabaseType_SqlServer2005	2	RPS mapping used to replicate data to a SQL Server 2005 database.
DatabaseType_SqlServer2008	3	RPS mapping used to replicate data to a SQL Server 2008 or later database.

rpsDefaultConnectionString

Type: String

The **rpsDefaultConnectionString** property of the **RelationalView** class contains the default RDBMS connection string. The connection string can be specified or overridden on the RPS node.

An example of the connection string for an RPS node to connect to an RDBMS is as follows:

```
DSN=SqlServerODBC; Database=MyDatabase
```

rpsDefaultPassword

Type: String[128]

The **rpsDefaultPassword** property of the **RelationalView** class contains the default RDBMS log-in password.

rpsDefaultUserName

Type: String[30]

The **rpsDefaultUserName** property of the **RelationalView** class contains the default RDBMS log-in user name.

rpsExceptionCreate

Type: Integer

The **rpsExceptionCreate** property of the **RelationalView** class contains a value that indicates what happens when an exception is raised during an RDBMS create operation.

The values for the **rpsExceptionCreate** property are listed in the following table.

RelationalView Class Constant	Value	Description
Exception_Halt	0	Create operation is aborted. Database tracking is stopped.
Exception_Alternate	1	Create operation is attempted as an update. Database tracking is not stopped.

rpsExceptionDelete

Type: Integer

The **rpsExceptionDelete** property of the **RelationalView** class contains a value that indicates what happens when an exception is raised during an RDBMS delete operation.

The values for the **rpsExceptionDelete** property are listed in the following table.

RelationalView Class Constant	Value	Description
Exception_Halt	0	Delete operation is aborted. Database tracking is stopped.
Exception_Alternate	1	Delete operation errors are ignored. Database tracking is not stopped.

rpsExceptionUpdate

Type: Integer

The **rpsExceptionUpdate** property of the **RelationalView** class contains a value that indicates what happens when an exception is raised during an RDBMS update operation.

The values for the **rpsExceptionUpdate** property are listed in the following table.

RelationalView Class Constant	Value	Description
Exception_Halt	0	Update operation is aborted. Database tracking is stopped.
Exception_Alternate	1	Update operation attempted as an insert. Database tracking is not stopped.

rpsLoggingOptions

Type: Integer

The **rpsLoggingOptions** property of the **RelationalView** class contains a value that indicates whether exception information for create, update, and delete statements is recorded in a table in the relational database in addition to being recorded in the **jommsg.log** file.

The values for the **rpsLoggingOptions** property are listed in the following table.

RelationalView Class Constant	Value	Description
ExceptionLogging_Default	0	Exception information recorded in **jommsg.log** file.
ExceptionLogging_RPSTable	1	Exception information recorded in **jommsg.log** file and in a table in the RDBMS.

rpsShowMethods

Type: Boolean

The read-only **rpsShowMethods** property of the **RelationalView** class contains **true** if the mapping was defined with an option set to display column mapping methods in the RPS wizard.

rpsShowVirtualProperties

Type: Integer

The read-only **rpsShowVirtualProperties** property of the **RelationalView** class contains a value that indicates whether virtual properties are displayed for selection in the RPS wizard.

The values for the **rpsShowVirtualProperties** property are listed in the following table.

Value	Description
0	No virtual properties are shown
1	Only condition-safe virtual properties are shown
2	All virtual properties are shown

rpsTopSchemaName

Type: String

The read-only **rpsTopSchemaName** property of the **RelationalView** class contains the highest-level superschema whose objects are available for selection in the RPS wizard.

rpsUseOidClassInstMap

Type: Boolean

The read-only **rpsUseOidClassInstMap** property of the **RelationalView** class specifies whether oids are mapped to two integer columns: class number and instance identifier.

schema

Type: Schema

The **schema** property of the **RelationalView** class contains a reference to the schema in which the RPS mapping or relational view is defined.

timeCreated

Type: TimeStamp

The **timeCreated** property of the **RelationalView** class contains the timestamp of the creation of the RPS mapping or relational view.

If you make changes to the RPS mapping or relational view, the value of the **timeCreated** property is updated.

RelationalView Methods

The methods defined in the **RelationalView** class are summarized in the following table.

Method	Description
addUserAttribute	Adds a specified user-defined attribute to the specified user-defined table
addUserTable	Adds a user-defined entity or a real JADE class type with soft attributes
changeColumnName	Changes the name of a column in a relational view table
columnExists	Returns **true** if the specified column exists
createExcludedJcfFile	Creates a command file to exclude tables and columns that are currently excluded
excludeTableColumnName	Excludes the specified column in the specified table
excludeTableName	Excludes the specified table
extractData	Extracts a specified table or all tables, using specified parameter values
extractDataAll	Extracts all tables using specified parameter values
extractDataUsingIniFileOptions	Extracts a specified table or all tables using values stored in the [**JadeRps**] section of the JADE initialization file
generateRpsTableCreationScript	Generates a script that creates the tables for an RPS mapping
getColumnFeature	Returns the feature (method or property) associated with a table column
getExcludedTableColumnNames	Adds the names of excluded columns in the specified table to the specified array
getExcludedTableNames	Adds the names of excluded tables to the specified array
getRpsMappedClasses	Adds classes that are involved in the RPS mapping to the specified set
getTableColumnNames	Adds the names of non-excluded columns for the specified table to the specified array
getTableNames	Adds the names of non-excluded tables to the specified array
isODBCRelationalView	Returns **true** if the receiver is being used as an ODBC relational view
isRpsMapping	Returns **true** if the receiver is being used as an RPS mapping
removeColumn	Removes the specified column from the specified table
removeTable	Removes the specified table
tableExists	Returns **true** if the specified table exists
versionRpsMapping	Versions the RPS mapping and returns the latest version

addUserAttribute

Signature
```
addUserAttribute(entityDesc: JadeRelationalEntityIF;
                 attrDesc:   JadeRelationalAttributeIF);
```

The **addUserAttribute** method of the **RelationalView** class adds a user-defined (*soft*) attribute to the user-defined (*soft*) table specified in the **entityDesc** parameter, if it exists in the relational view.

Note This method does not apply to RPS mappings.

The name returned by the **entityDesc.getSQLName** method is called to obtain the table name. If a table by that name is not found or if the table is not a user-defined table, an exception is raised.

The value of the **entityDesc** parameter specifies an implementation of the **JadeRelationalAttributeIF** interface that correctly describes the attribute being added.

Calls to this method can raise the following exceptions.

- JErr_Table_Not_Found : Relational Table not found.
- JErr_Attribute_Name_Conflict : Attribute name null or already used as column in the selected table.
- JErr_SQL_Type_Not_Mapped : JADE Type specified does not have a supported SQL type mapping.
- String_Too_Long : Attribute name exceeds maximum of 80 characters.
- JErr_Invalid_For_RpsMapping : May only be called for ODBC Relational Views.
- JErr_No_Jade_Type : No JADE Type defined for this attribute. entityDesc.getJadeType() returned null.
- JErr_Not_Soft_Table : The table exists in the View, but it is not a user defined table.

addUserTable

Signature
```
addUserTable(entityDesc:           JadeRelationalEntityIF;
             includeRealProperties: Boolean;
             includeMethods:        Boolean);
```

The **addUserTable** method of the **RelationalView** class adds a user-defined (*soft*) entity or a real JADE class type with soft attributes to the receiver (that is, to the relational view). The **entityDesc** parameter specifies an implementation of the **JadeRelationalEntityIF** interface that correctly describes the entity being added.

Note This method does not apply to RPS mappings.

If the value of the **includeRealProperties** parameter is **true** and **entityDesc.getJadeClass** returns a valid class, the properties of this class that can be mapped to valid SQL types are mapped to columns in the table. If the value of the **includeMethods** parameter is **true** and **entityDesc.getJadeClass** returns a valid class, the methods of this class that are not updating, have no parameters, and return a value that can be mapped to a valid SQL types are mapped to columns in the table.

Calls to this method can raise the following exceptions.

- JErr_Table_Name_Conflict : Table name already used.
- String_Too_Long : Table name exceeds maximum of 80 characters.
- JErr_Invalid_For_RpsMapping : May only be called for ODBC Relational Views.

changeColumnName

Signature `changeColumnName(tableName: String; oldColumnName: String; newColumnName: String);`

The **changeColumnName** method of the **RelationalView** class changes the name of the column specified by the value of the **oldColumnName** parameter in the table specified by the **tableName** parameter to the value specified by the **newColumnName** parameter.

Note This method does not apply to RPS mappings.

Calls to this method can raise the following exceptions.

- JErr_Attribute_Name_Conflict : Attribute name null or already used as column in the selected table.
- JErr_Invalid_For_RpsMapping : May only be called for ODBC Relational Views.
- JErr_Table_Not_Found : Relational Table not found.
- JErr_Column_Not_Found : Column not found in Relational Table.
- JErr_ColumnName_Cannot_Change : column name cannot be changed (for example, oid or index)

columnExists

Signature `columnExists(tableName: String; columnName: String): Boolean;`

The **columnExists** method of the **RelationalView** class returns **true** if the column specified in the **columnName** parameter exists in the table specified in the **tableName** parameter; otherwise it returns **false**. This method returns **false** if the specified table does not exist.

Note This method applies to RPS mappings and ODBC relational views.

createExcludedJcfFile

Signature `createExcludedJcfFile(file: File): Boolean;`

The **createExcludedJcfFile** method of the **RelationalView** class creates a JADE command file to exclude all tables and columns currently excluded in the RPS mapping.

Note This method does not apply to ODBC relational views.

The file represented by the **file** parameter must be initialized for output to a valid file name before the **createExcludedJcfFile** method is called. The method returns **true** if the command file is created successfully.

When the RPS mapping is loaded, which happens during a normal schema load, the exclusions must be reapplied. To do this, load the file created by the **createExcludedJcfFile** method using the batch Schema Load utility (**jadloadb**). For details, see "Site-Specific RPS Mapping Customization", in Chapter 2 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

excludeTableColumnName

Signature
```
excludeTableColumnName(tableName:  String;
                       columnName: String): Boolean updating;
```

The **excludeTableColumnName** method of the **RelationalView** class excludes the column specified by the **columnName** parameter in the table specified by the **tableName** parameter from the RPS mapping.

This method returns **true** if the column is successfully excluded.

Note This method applies to RPS mappings and ODBC relational views.

The method returns **false** if any of the following applies.

- The table specified by the **tableName** parameter is not found in the RPS mapping
- The column specified by the **columnName** parameter is not found in the RPS mapping
- The column specified by the **columnName** parameter is a primary key
- The RPS mapping has not been versioned
- The RPS mapping is not the latest version

excludeTableName

Signature
```
excludeTableName(tableName: String): Boolean updating;
```

The **excludeTableName** method of the **RelationalView** class excludes the table specified by the **tableName** parameter from the RPS mapping. The method returns **true** if the table is successfully excluded.

Note This method applies to RPS mappings and ODBC relational views.

The method returns **false** if any of the following applies.

- The table specified by the **tableName** parameter is not found in the RPS mapping
- The RPS mapping has not been versioned
- The RPS mapping is not the latest version

extractData

Signature
```
extractData(tableName:               String;
            executionLocation:       Integer;
            scriptFilePath:          String;
            dataFilesPath:           String;
            rdbDataFilesPath:        String;
            rdbName:                 String;
            extractHistoricalTables: Boolean;
            serverName:              String;
            extractWorkers:          Integer): Process;
```

The **extractData** method of the **RelationalView** class starts the RPS Datapump application on the server node to extract data for the table specified by the **tableName** parameter or for all tables on an SDS secondary or RPS node.

Notes This method does not apply to ODBC relational views.

You can execute this method on an SDS secondary *or* an RPS node, but not on the primary node. Running an RPS extract on an SDS node causes tracking to be stopped during the extract.

The **extractData** method parameters are listed in the following table.

Parameter	Specifies the …
tableName	The name of the table for which data is extracted. If null or an empty string, data for all tables is extracted.
executionLocation	The location used for loading the extracted data. Allowed values can be specified using the **Load_ServerExecute** (0) and **Load_ClientExecute** (1) **RelationalView** class constants.
scriptFilePath	The output directory for the script files.
dataFilesPath	The output directory for the data files.
rdbDataFilesPath	The path of the data files directory from the perspective of the RDBMS database.
rdbname	The name of the RDBMS database.
extractHistoricalTables	If historical table data is to be extracted.
serverName	The name of the RDBMS server.
extractWorkers	The number of extract worker processes to run.

The method returns the process of the application that extracts the table data. You can register to receive notifications for events occurring for the process that carries out the data extraction in the following table.

Process Class Constant	Value
RPS_EXTRACT_FAILED_EVENT	202
RPS_EXTRACT_FINISHED_EVENT	203

extractDataAll

Signature

```
extractDataAll(executionLocation:       Integer;
               scriptFilePath:          String;
               dataFilesPath:           String;
               rdbDataFilesPath:        String;
               rdbName:                 String;
               extractHistoricalTables: Boolean;
               serverName:              String;
               extractWorkers:          Integer;
               extractOrder:            Integer;
               extractFirst:            String;
               userDataPumpSchema:      String;
               userDataPumpApp:         String): Process;
```

The **extractDataAll** method of the **RelationalView** class starts the user-defined RPS Datapump application specified by the **userDataPumpApp** and **userDataPumpSchema** parameters on the server node to extract data for all tables on an SDS secondary or RPS node.

Notes This method does not apply to ODBC relational views.

You can execute this method on an SDS secondary *or* an RPS node, but not on the primary node. Running an RPS extract on an SDS node causes tracking to be stopped during the extract.

The **extractDataAll** method parameters are listed in the following table.

Parameter	Specifies the …
executionLocation	The location that will be used for loading the extracted data. Allowed values can be specified using the **Load_ServerExecute** (0) and **Load_ClientExecute** (1) **RelationalView** class constants.
scriptFilePath	The output directory for the script files.
dataFilesPath	The output directory for the data files.
rdbDataFilesPath	The path of the data files directory from the perspective of the RDBMS database.
rdbname	The name of the RDBMS database.
extractHistoricalTables	If historical table data is to be extracted.
serverName	The name of the RDBMS server.
extractWorkers	The number of extract worker processes to run.
extractOrder	The order in which the tables are to be extracted; possible values specified by the **RelationalView** class constants listed in the following table.
extractFirst	The names of the tables to be extracted first, if any, delimited by semicolons.
userDataPumpSchema	The name of the schema for the user-defined **Datapump** application. If null, the default **Datapump** application is used.
userDataPumpApp	The name of the user-defined data pump application. If executed on the primary, the user-defined data pump may not be used. The user-defined data pump may be used in an RPS or SDS node. The value of the user-defined **Datapump** application (or **<default>**) is written out to the **DataPumpApplication** parameter in the [JadeRps] section of the JADE initialization file.

Class Constant	Value Order of Output Tables
ExtractOrderDefault	0 - No order specified
ExtractOrderClassInstances	1 - Number of instances of the class from highest to lowest. Note that determining the number of instances may delay the start of extractions.
ExtractOrderSelectedFirst	2 - As specified in the extractFirst parameter, then in default order.

The method returns the process of the application that extracts the table data. You can register to receive notifications for events occurring for the process that carries out the data extraction in the following table.

Process Class Constant	Value
RPS_EXTRACT_FAILED_EVENT	202
RPS_EXTRACT_FINISHED_EVENT	203

Calls to this method can raise the following exception.

- JErr_RpsExtractRequestError : Error in parameters. See extended error text for details.

extractDataUsingIniFileOptions

Signature `extractDataUsingIniFileOptions(tableName: String): Process;`

The **extractDataUsingIniFileOptions** method of the **RelationalView** class starts the RPS Datapump application on the server node to extract data for the table specified by the **tableName** parameter or for all tables if the value of the **tableName** parameter is an empty string.

Notes This method does not apply to ODBC relational views.

You can execute this method on an SDS secondary *or* an RPS node, but not on the primary node. Running an RPS extract on an SDS node causes tracking to be stopped during the extract.

The method uses applicable settings from parameters in the [JadeRps] section of the JADE initialization file.

This method returns the process of the application that extracts the table data. You can register to receive notifications for events occurring for the process that carries out the data extraction in the following table.

Process Class Constant	Value
RPS_EXTRACT_FAILED_EVENT	202
RPS_EXTRACT_FINISHED_EVENT	203

generateRpsTableCreationScript

Signature
```
generateRpsTableCreationScript(relDataBaseName:    String;
                               scriptFile:         File;
                               tablesOption:       Integer;
                               selectedTableNames: HugeStringArray);
```

The **generateRpsTableCreationScript** method of the **RelationalView** class enables you to programmatically generate a script that creates the tables for an RPS mapping.

The method parameters are listed in the following table.

Parameter	Description
relDataBaseName	Name of the relational database
scriptFile	Script file that was set up by the caller
tablesOption	One of the **RelationalView** class **RpsScript_UserTablesOnly** (0), **RpsScript_UserAndJadeTables** (1), or **RpsScript_SelectedTables** (2) constant values
selectedTableNames	List of tables to output if the value of the **tablesOption** parameter is set to **RpsScript_SelectedTables**; otherwise null

The SQL script to create the tables is output using the **scriptFile** parameter. The caller must insure that the file can be created when called. The SQL script must always be created as an ANSI file for use with SQL Server.

Doing the initial write will do an implicit open if it is not open already.

Note As the file represented by the **scriptFile** parameter could be open before the **generateRpsTableCreationScript** method is called, the file is not closed when the method returns.

You can explicitly close the file using the **close** method of the **File** class.

The **tablesOption** parameter controls what tables are created in the SQL script, by specifying one of the following **RelationalView** class constants.

- **RpsScript_SelectedTables**, which generates a creation script for the user table names specified in the **selectedTableNames** parameter
- **RpsScript_UserAndJadeTables**, which generates a creation script for all user tables and all JADE RPS tables (that is, **JADE_TRANSACTIONS**, **JADE_CONTROL_INFO**, and **JADE_EXCEPTION_LOG**)
- **RpsScript_UserTablesOnly**, which generates a creation script for all user tables in the RPS mapping

getColumnFeature

Signature
```
getColumnFeature(tableName:  String;
                 columnName: String): Feature;
```

The **getColumnFeature** method of the **RelationalView** class returns the feature (that is, method or property) associated with a column specified by the **columnName** parameter in the table specified by the **tableName** parameter.

Note This method applies to RPS mappings and ODBC relational views.

Calls to this method can raise the following exceptions.

- JErr_Table_Not_Found : Relational Table not found.
- JErr_Column_Not_Found : Column not found in Relational Table.

getExcludedTableColumnNames

Signature
```
getExcludedTableColumnNames(tableName:   String;
                            columnNames: HugeStringArray input);
```

The **getExcludedTableColumnNames** method of the **RelationalView** class returns the names of columns in the table specified by the **tableName** parameter in the RPS mapping that have been specifically excluded in the array specified by the **columnNames** parameter.

The **tableName** array is not cleared before names are added.

Note This method does not apply to ODBC relational views.

If the table specified by the **tableName** parameter is not found in the RPS mapping, the **columnNames** array is empty.

getExcludedTableNames

Signature
```
getExcludedTableNames(tableNames: HugeStringArray input);
```

The **getExcludedTableNames** method of the **RelationalView** class returns the names of tables in the RPS mapping that have been specifically excluded in the array specified by the **tableNames** parameter.

Note This method does not apply to ODBC relational views.

The **tableNames** array is not cleared before names are added.

getRpsMappedClasses

Signature `getRpsMappedClasses(rpsMappedClasses: ClassSet input);`

The **getRpsMappedClasses** method of the **RelationalView** class adds classes that are involved in the RPS mapping (that is, the receiver) to the input **rpsMappedClasses** parameter.

Note This method does not apply to ODBC relational views.

The **rpsMappedClasses** collection is not cleared before classes are added.

getTableColumnNames

Signature
```
getTableColumnNames(tableName:   String;
                    columnNames: HugeStringArray input);
```

The **getTableColumnNames** method of the **RelationalView** class returns the names of the columns in the table specified by the **tableName** parameter in the array specified by the **columnNames** parameter.

Note This method applies to RPS mappings and ODBC relational views.

The **columnNames** array is not cleared before names are added.

The names of columns in the RPS mapping that have been specifically excluded are not added to the **columnNames** array. If the table specified by the **tableName** parameter is not found in the relational view or RPS mapping or if it has been specifically excluded, the **columnNames** array is empty.

getTableNames

Signature `getTableNames(tableNames: HugeStringArray input);`

The **getTableNames** method of the **RelationalView** class returns the names of the tables in the relational view or RPS mapping in the array specified by the **tableNames** parameter.

The **tableNames** array is not cleared before names are added.

Note This method applies to RPS mappings and ODBC relational views.

The names of tables in the RPS mapping that have been specifically excluded are not added to the **tableNames** array.

isODBCRelationalView

Signature `isODBCRelationalView(): Boolean;`

The **isODBCRelationalView** method of the **RelationalView** class returns **true** if the receiver is being used as an ODBC relational view and **false** if it is being used as an RPS mapping.

Note This method applies to RPS mappings and ODBC relational views.

isRpsMapping

Signature `isRpsMapping(): Boolean;`

The **isRpsMapping** method of the **RelationalView** class returns **true** if the receiver is being used as an RPS mapping and **false** if it is being used as an ODBC relational view.

Note This method applies to RPS mappings and ODBC relational views.

removeColumn

Signature
```
removeColumn(tableName:  String;
             columnName: String) final, updating;
```

The **removeColumn** method of the **RelationalView** class removes the column specified by the **columnName** parameter in the table specified by the **tableName** parameter.

Note This method applies to RPS mappings and ODBC relational views.

Calls to this method can raise the following exceptions.

- JErr_Table_Not_Found : Relational Table not found.
- JErr_Column_Not_Found : Column not found in Relational Table.

removeTable

Signature `removeTable(tableName:  String) final, updating;`

The **removeTable** method of the **RelationalView** class removes the table specified in the **tableName** parameter.

Note This method applies to RPS mappings and ODBC relational views.

Calls to this method can raise the following exception.

- JErr_Table_Not_Found : Relational Table not found.

tableExists

Signature `tableExists(tableName: String): Boolean;`

The **tableExists** method of the **RelationalView** class returns **true** if the table specified in the **tableName** parameter exists; otherwise it returns **false**.

Note This method applies to RPS mappings and ODBC relational views.

versionRpsMapping

Signature `versionRpsMapping(): RelationalView updating;`

The **versionRpsMapping** method of the **RelationalView** class versions the RPS mapping if it is not already versioned, and returns the latest version of the RPS mapping.

Note This method does not apply to ODBC relational views.

This method must be called before any changes are made to the RPS mapping using the **excludeTableName** method or the **excludeTableColumnName** method.

All changes must be made to the latest version of the RPS mapping.

RootSchemaSession Class

The **RootSchemaSession** class is the transient class that provides a superclass for Web session classes in subschemas.

For details about the properties defined in the **RootSchemaSession** class, see "RootSchemaSession Properties", in the following subsection.

Inherits From: WebSession

Inherited By: (None)

RootSchemaSession Properties

The properties defined in the **RootSchemaSession** class are summarized in the following table.

Property	Description
allowHiddenControlEvents	Specifies whether hidden controls can invoke event methods
userSecurityLevel	Contains the numeric security level for the current user in the Web session

allowHiddenControlEvents

Type: Boolean

Default Value: False

The **allowHiddenControlEvents** property of the **RootSchemaSession** class is a protected property that specifies whether hidden controls on Web pages can invoke event methods.

Set this property to **true** if you want to invoke event methods for hidden controls on Web pages.

userSecurityLevel

Type: Integer

Availability: Read or write at run time

The **userSecurityLevel** property of the **RootSchemaSession** class contains the security level for the current user in the Web session. The default value is the user security level of the application (that is, the value of **Application**::**userSecurityLevel**).

It is your responsibility to assign a value to this property. The **userSecurityLevel** property is used in conjunction with the **securityLevelEnabled** and **securityLevelVisible** properties for **Window** classes.

When a form is loaded, the following rules determine the state of controls and menu items on the Web form.

- If **securityLevelEnabled > currentSession.userSecurityLevel** for a Web control or menu item, it is automatically disabled, regardless of the value of the **Application** class **enabled** property.
- If **securityLevelVisible > currentSession.userSecurityLevel** for a control or menu item, its **visible** property is set to **false**.
- If **securityLevelVisible > currentSession.userSecurityLevel** for a Web form when the **Application** class **show** or **showModal** method is called, the following message is displayed.

```
Requested form form-name is not valid
```

You can subsequently override the values of the control or menu item **enabled** and **visible** properties.

You should set the **userSecurityLevel** property during the **initialize** method or **getAndValidateUser** process before the creation of a Web form, as the setting of this property is actioned when forms and controls are created.

Changing the value of **app.userSecurityLevel** or the **RootSchemaSession** class **userSecurityLevel** method does not change the behavior of Web forms that have already been loaded.

Changing the value of the **Window** class **securityLevelEnabled** property of a Web form, control, or menu item causes a re-evaluation of its enabled status, based on the above rules. Changing the value of the **Window** class **securityLevelVisible** property of a Web form, control, or menu item causes a re-evaluation of its visible status, based on the above rules.

Schema Class

The **Schema** class represents the object model for a specific application domain. Instances of the **Schema** class contain the classes and primitive types that define the object model. For more details, see Chapter 1, "JADE Concepts and Terminology", in the *JADE Development Environment User's Guide*.

For details about the constants, properties, and methods defined in the **Schema** class, see "Schema Constants", "Schema Properties", and "Schema Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

Schema Class Constants

The constants provided by the **Schema** class are listed in the following table.

Constant	Integer Value	Description
FormsMngmt_Default	FormsMngmt_Multi_Multi	Default value
FormsMngmt_Multi_Multi	0	Multiple form definition and multiple translation
FormsMngmt_Single_Multi	2	Single form definition and multiple translations
FormsMngmt_Single_Single	1	Single form definition and single translation

Schema Properties

The properties defined in the **Schema** class are summarized in the following table.

Property	Description
externalDatabases	Contains a list of all external databases by name
formsManagement	Contains the style of forms management
jomVersion	Contains a string of the JADE Object Manager version
name	Contains a string of the schema name
needsReorg	Specifies if any class in the schema requires a reorganization
patchVersion	Contains the patch version number
superschema	Contains a reference to the superschema (parent) of the schema
text	Contains a string of the schema descriptive text

All properties are read-only.

externalDatabases

Type: ExternalDatabaseByNameDict

The read-only **externalDatabases** property of the **Schema** class contains a reference to the names of all external databases in the schema.

formsManagement

Type: Byte

The read-only **formsManagement** property of the **Schema** class contains the style of forms management used by the schema and its subschemas. For more details, see "Forms Translation Styles", in Chapter 11 of the *JADE Development Environment User's Guide*.

jomVersion

Type: String[9]

The read-only **jomVersion** property of the **Schema** class contains the JADE Object Manager version as a string.

Note This property is set only for the **RootSchema**, and it contains the version of the internal JADE Object Manager format.

name

Type: String[100]

The read-only **name** property of the **Schema** class contains the schema name as a string.

The code fragment in the following example shows the use of the **name** property.

```
if global.appCount = 0 then
    beginTransaction;
    global.appCount := 1;
    commitTransaction;
    app.startApplication(currentSchema.name, app.name);
endif;
```

needsReorg

Type: Boolean

The read-only **needsReorg** property of the **Schema** class contains **true** if any class in the schema requires reorganization.

patchVersion

Type: Integer

The read-only **patchVersion** property of the **Schema** class contains the change patch version as an integer.

superschema

Type: Schema

The read-only **superschema** property of the **Schema** class contains a reference to the superschema (parent) of the schema.

relationalViews

The read-only **relationalViews** property of the **Schema** class contains a reference to the names of all relational views in the schema.

rpsDatabases

The read-only **rpsDatabases** property of the **Schema** class contains a reference to the names of all RPS mappings in the schema.

text

Type: String

The read-only **text** property of the **Schema** class contains the schema descriptive text as a string.

Schema Methods

The methods defined in the **Schema** class are summarized in the following table.

Method	Description
addCompileTranslatableString	Adds a translatable string to all base locales of the receiving schema
addUserCollectionSubclass	Creates a user collection class in the receiving schema
addUserSubclass	Creates a user class in the receiving schema
allClasses	Returns all classes in the schema and its superschemas
allDatabases	Returns all databases in the schema and its superschemas
allJadeInterfaces	Returns all interfaces defined the receiver and its superschemas
allLibraries	Adds all libraries in the schema to the library collection
allPrimitives	Returns all primitive types in the schema and its superschemas
allSubschemas	Recursively adds all subschemas in the schema to the schema collection
buildFormData	Checks that the form build data for every form in the schema and subschemas is up-to-date
constantNames	Returns a concatenated string of all constants in the schema
createWebServiceApplication	Creates a definition for a Web service provider application
deleteUserSubclass	Deletes a user class in the receiving schema
extractControlIdsCSV	Creates a CSV file containing generated Windows control id for controls in all schemas
extractControlIdsCSVforSchema	Creates a CSV file containing generated Windows control id for controls in the current schema
findClassInBranch	Returns the type of the specified class
findClassInSubschema	Returns the specified class
findFormForLocale	Returns the specified form for the specified locale

Method	Description
findFormForLocaleInAllSchemas	Returns the specified form for the specified locale from all schemas
findFormForLocaleInSupers	Returns the specified form for the specified locale from superschemas
findGlobalConstantInBranch	Returns the specified global constant instance
findMeForm	Returns the specified form from the base locale
findName	Returns the type of the specified class or primitive type
findProperty	Returns the property of the specified name
findType	Returns the type of the specified integer
generateWSDL	Creates a WSDL file for an existing JADE Web service provider application
getAllBaseLocales	Returns all base (non-clone) locales in the schema
getAllClasses	Returns a reference to all classes in the current schema and its superschemas
getAllFormTranslations	Returns all form translations of the specified form
getAllInheritedLocales	Gets all locales inherited by the current schema
getAllLocales	Gets all locales in the current schema and superschemas
getAllLocalLocales	Gets all locales defined in the current schema
getAllRpsMappings	Returns an array of all RPS mappings defined in the current schema
getAllSystemLocales	Adds an instance of the **LocaleNameInfo** class for each locale known to the operating system to the specified object array
getAppliedPatches	Returns information about the patches applied to system schemas in the JADE database
getBaseLocalesLocal	Populates the specified collection with the base locales defined in the receiving schema
getCategory	Returns the specified global constant category from the receiver and superschemas
getClass	Gets the specified class
getClassByNumber	Returns the class specified by number
getConstant	Returns the specified constant
getConstantCategory	Returns the specified global constant category from the receiver
getControlClasses	Adds all subclasses of the **Control** class to the specified class collection
getCurrentLocaleId	Returns the current locale identifier (LCID) of the application process when enhanced locale support is enabled; otherwise it returns the identifier of the current locale
getDefaultLocale	Returns a reference to the default locale for the schema
getExternalDatabase	Returns a reference to the shared transient instance of the external database
getFormatAnywhereInPath	Returns the specified locale format from any locale in the current version of a schema in the schema path

Method	Description
getFormatAnywhereInPathLatest	Returns the specified locale format from any locale in the latest version of a schema in the schema path
getFormatAnywhereInSubs	Returns the specified locale format from any locale in the current version of the receiver or its subschemas
getFormatAnywhereInSubsLatest	Returns the specified locale format from any locale in the latest version of the receiver or its subschemas
getFunction	Returns the specified function
getGlobalClass	Returns the **Global** class object for the schema
getGlobalConstant	Returns the specified global constant
getHtmlDocumentSource	Returns the HTML source of the specified HTML document
getImportedClass	Returns a reference to a class imported as part of an imported package
getImportedJadeInterface	Returns a reference to an interface imported as part of an imported package
getInheritedFormats	Gets a collection of all formats inherited from superschemas
getInheritedXlatableStrings	Gets a collection of all translatable strings defined for the specified locale
getJadeInterface	Returns a reference to the specified interface
getLibrary	Gets the library with the specified name
getLocalClass	Returns the specified class from the current schema
getLocale	Returns the specified locale from the current schema and all of its subschemas
getLocaleCurrencyInfo	Gets the currency formatting information for the specified locale
getLocaleDateInfo	Gets the date formatting information for the specified locale
getLocaleFullInfo	Gets the full formatting information for the specified locale
getLocaleInSubschemas	Returns the specified locale from subschemas
getLocaleLocal	Returns the specified locale from the current schema
getLocaleNameInfo	Gets the name formatting information for the specified locale
getLocaleNumericInfo	Gets the numeric formatting information for the specified locale
getLocaleTimeInfo	Gets the time formatting information for the specified locale
getLocalFormats	Adds all user-defined formats of the receiver to the specified array
getLocalLocaleInSubschemas	Returns the specified locale from subschemas
getLocalPrimitive	Returns the specified primitive type from the current schema
getName	Returns the name of the receiver schema as a string
getOidForObject	Returns the object identifier (oid) of the specified object
getPrimitive	Returns the primitive type with the specified name
getRelationalView	Returns the relational view in this schema with the specified name
getRpsMapping	Returns the RPS mapping in this schema with the specified name

Method	Description
getSchema	Returns the receiver or a subschema of the receiver with the specified name
getSubschema	Returns a subschema with the specified name
getSubschemas	Adds all subschemas of the receiving schema to the schema name dictionary
getUserAppliedPatches	Returns information about the patches applied to user schemas in the JADE database
getUserFormat	Returns the user format with the specified name
getWebServiceConsumerNames	Populates a string array with the names of Web Service consumers in the receiving schema
globalException	Returns the number of the specified global exception
importWSDL	Creates a Web service consumer by importing a specified WSDL file
isLocalLocale	Returns **true** if the specified locale is local to the current schema
loadHTMLDocuments	Processes all HTML document files in the specified folder
makeLocaleNameFromId	Returns a string of the name of the specified locale
nonGUIGlobalExceptionHandler	Logs exception details to the application exception log file, aborts any transaction, and then returns **Ex_Abort_Action**
regenerateRelationalView	Dynamically rebuilds the specified relational view, deleting an existing relational view, if applicable, or creating the relational view if it does not exist
removeWebConsumer	Removes a Web service consumer with the specified name
resetUserAppliedPatches	Resets information about patches applied to the specified user schema to null
reorgInProgress	Returns **true** if a reorganization is currently "actively" in progress
reorgIsWaitingForTransition	Returns **true** if a reorganization is in dual-update mode and waiting for the transition
setHtmlDocumentSource	Sets the HTML source for the specified HTML document
withAllSubschemas	Adds the current schema to the collection of its subschemas
withAllSuperschemas	Adds the current schema to the collection of its superschemas and returns the superschema collection

addCompileTranslatableString

Signature

```
addCompileTranslatableString(source:      String;
                             errorCode:   Integer output;
                             errorOffset: Integer output;
                             errorLength: Integer output): Boolean;
```

The **addCompileTranslatableString** method of the **Schema** class compiles and adds a translatable string to all base locales of the receiving schema. If the compilation fails, the method returns **true**, the translatable string is not added, the current transaction is aborted, and the **errorCode** parameter contains the error number.

The **addCompileTranslatableString** method parameters are listed in the following table.

Parameter	Description
source	The text to be displayed for the translatable string.
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the translatable string compiled successfully.
errorOffset	The position of the error in the translatable string. Note that the first character of the translatable string has a position of zero (**0**).
errorLength	The length in characters of the error in the translatable string.

addUserCollectionSubclass

Signature
```
addUserCollectionSubclass(superclass : CollClass input;
                          className  : String;
                          mapFileName: String): JadeUserCollClass updating;
```

The **addUserCollectionSubclass** method of the **Schema** class creates and returns a user collection class in the receiving schema with a name specified by the **className** parameter. The naming rules for user collection classes are the same as for classes added in the Class Browser. The new class is a subclass of the class specified by the **superclass** parameter, which must be **Array**, **ExtKeyDictionary**, **MemberKeyDictionary**, **Set**, or a subclass of these collection classes.

The **mapFileName** parameter must be the name of an existing map file in the receiving schema.

For more details about user classes, see "Adding User Classes at Run Time", in Chapter 21 of the *JADE Developer's Reference*.

addUserSubclass

Signature
```
addUserSubclass(superclass : Class input;
                className  : String;
                mapFileName: String): JadeUserClass updating;
```

The **addUserSubclass** method of the **Schema** class creates and returns a user class in the receiving schema with a name specified by the **className** parameter. The naming rules for user classes are the same as for classes added in the Class Browser. The new class is a subclass of the class specified by the **superclass** parameter, which must be a class defined in the receiving schema or a superschema.

The **mapFileName** parameter must be the name of an existing map file in the receiving schema.

For more details about user classes, see "Adding User Classes at Run Time", in Chapter 21 of the *JADE Developer's Reference*.

allClasses

Signature
```
allClasses(): ClassColl;
```

The **allClasses** method of the **Schema** class returns a reference to all classes in the schema and its superschemas.

allDatabases

Signature `allDatabases(): DatabaseNDict;`

The **allDatabases** method of the **Schema** class returns a reference to all databases in the receiver.

allJadeInterfaces

Signature `allJadeInterfaces(): JadeInterfaceColl;`

The **implementsInterface** method of the **Schema** class returns a reference to all interfaces in the schema and its superschemas.

allLibraries

Signature `allLibraries(libs: ObjectArray input);`

The **allLibraries** method of the **Schema** class adds all libraries in the schema to the array specified in the **libs** parameter.

Note The object array is not cleared before instances are added.

allPrimitives

Signature `allPrimitives(): TypeColl;`

The **allPrimitives** method of the **Schema** class returns a reference to all primitive types in the schema and its superschemas.

allSubschemas

Signature `allSubschemas(subs: SchemaColl input);`

The **allSubschemas** method of the **Schema** class recursively adds all subschemas of the receiving schema to the collection specified in the **subs** parameter. The collection is not cleared before instances are added.

As this method is recursive, all schemas below the receiving schema (both direct and indirect descendants) are returned.

buildFormData

Signature `buildFormData() updating;`

The **buildFormData** method of the **Schema** class examines the form build data for every form in the receiver and all of its subschemas. If the data is not up-to-date, it is constructed. This method enters transaction state if the process is not already in that state. Similarly, a **commitTransaction** instruction is called at the end of the method if the process was not initially in transaction state.

An exception is raised if the schema is currently versioned.

The following code fragment runs the **buildFormData** method for the current schema and its subschemas.

```
currentSchema.buildFormData;
```

Form build data is normally constructed when a form is saved in the JADE Painter. If that data is not up-to-date at run time, the form build construction occurs the first time a form is created from logic. In some situations, form build data can become invalid; for example, after upgrading to a new JADE version when new features have been added and following a schema and forms definition file load that involved a reorganization.

Note Run the **buildFormData** method before turning on the system production mode flag; otherwise any form where the form build data is not up-to-date must perform this step every time a form is created because the data cannot be stored persistently. This would be more expensive than creating the form from its build data and it also results in more network traffic being generated in thin client mode.

constantNames

Signature `constantNames(): String;`

The **constantNames** method of the **Schema** class returns the names of all constants in the schema and concatenates them into a string. Each constant name is separated by a space.

createWebServiceApplication

Signature
```
createWebServiceApplication(applicationName:      String;
                            applicationType:      String;
                            applicationVersion:   String;
                            localeName:           String;
                            initializeMethodName: String;
                            finalizeMethodName:   String;
                            connectionName:       String;
                            numberOfCopies:       Integer;
                            sessionTimeout:       Integer;
                            minimumResponseTime:  Integer;
                            disableMessages:      Boolean;
                            urlScheme:            String;
                            urlMachineName:       String;
                            urlVirtualDirectory:  String;
                            urlSupportLibrary:    String;
                            exposureNames:        String);
```

The **createWebServiceApplication** method of the **Schema** class creates a definition for a Web service provider application.

The **createWebServiceApplication** method parameters are listed in the following table.

Parameter	Description
applicationName	The name of the application to be created. The name must not conflict with the name of an exposure list. **Note** If an application with this name already exists, the application is updated with the new values.
applicationType	Valid values for this parameter are **GUI** and **NON-GUI**. When the value is **GUI**, the Web application monitor is displayed at application startup. The monitor is not displayed when the value is **NON-GUI**.

Parameter	Description
applicationVersion	The application version number. The value of this parameter can be null (**""**). The version number cannot exceed 30 characters.
localeName	The name of the locale. The value of this parameter can be null (**""**). If a locale is specified, it must be one of the locales defined for the schema.
initializeMethodName	The method to be invoked when the application starts. The value of this parameter can be null (**""**). The format is: `method-name or class-name::method-name` If a method name only is specified, the method had to exist on the **Application** subclass (or one of its superclasses) for the schema. If a class name and method name are specified, the class has to be the **Application** class or one of it subclasses that exist in this schema or one of its superschemas and the method has to exist on the class. In addition, the method cannot have parameters or a return type.
finalizeMethodName	The method to be invoked when the application terminates. The value of this parameter can be null (**""**). The format is: `method-name or class-name::method-name` If a method name only is entered, the method had to exist on the **Application** subclass (or one of its superclasses) for the schema. If a class name and method name are specified, the class has to be the **Application** class or one of it subclasses that exist in this schema or one of its superschemas and the method has to exist on the class. In addition, the method cannot have parameters or a return type.
connectionName	The named pipe name or a TCP/IP connection. The value of this parameter can be null (**""**). For a TCP/IP connection, the format is *machine name or ip-address:port-number*. **Note** This value is used only if the URL scheme is **http**
numberOfCopies	The number of copies of the application to be started in the node. This parameter must have a value of one (**1**) or greater.
sessionTimeout	The session timeout in minutes. The value of this parameter can be null (**""**). A value of zero (**0**) means that there is no session timeout.
minimumResponseTime	The minimum response time in seconds for the Web service to respond. The value of this parameter can be null (**""**). A value of zero (**0**) means that there is no minimum response time.
disableMessages	Set this parameter to **true** to disable messages from appearing when the Web application monitor is running. The default value is **false**.
urlScheme	Valid values for this parameter are **tcp** or **http**. The value of this parameter can be null (**""**).
urlMachineName	When using the **tcp** protocol the format is *machine-name:port-number*. The value of this parameter can be null (**""**).
urlVirtualDirectory	The name of a virtual directory on the target Web server. The value of this parameter can be null (**""**).
urlSupportLibrary	**jadehttp.dll** must be entered if the target Web server is IIS. The value of this parameter can be null (**""**).

Parameter	Description
exposureNames	A list of exposures that the Web service can use with each exposure name separated from the next by a space. The exposures must already exist and must not require regeneration. You must specify at least one exposure name.

If the validation or update fails, an exception is raised. The **extendedErrorText** property contains details of the exception.

The following example shows the use of the **createWebServiceApplication** method.

```
currentSchema.createWebServiceApplication("CustomerServicesApp", "NON-GUI",
    "", "5129", "initialiseWebService","CustomerGlobal::finaliseWebService",
    "localhost:54000", 1, 10, 0, true, "http", "localhost", "jade",
    "jadehttp.dll","ErewhonCustomer PrivateCustomer");
```

You can override some parameters at run time by runtime configuration settings. For details, see "Configuring Web Applications", in Chapter 3 of the *JADE Web Application Guide*.

deleteUserSubclass

Signature `deleteUserSubclass(superclass: Class;`
`className : String);`

The **deleteUserSubclass** method of the **Schema** class deletes the user class with the specified name and superclass in the receiving schema. The class cannot be deleted if instances of the class exist or if the class is being used by another process.

For more details about user classes, see "Adding User Classes at Run Time", in Chapter 21 of the *JADE Developer's Reference*.

extractControlIdsCSV

Signature `extractControlIdsCSV();`

The **extractControlIdsCSV** method of the **Schema** class creates a comma-separated values (CSV) file containing the generated Windows control id for each JADE control.

The control ids are used by testing tools to identify required elements on a form. The identifier (id) for a control on a form created in the JADE Painter is retained for the lifetime of the form (unless the control is deleted and re-added using the JADE Painter).

There is an entry for each control on every form in the schema (the receiver) and in its subschemas, in the following format.

```
schema_name, form_name, control_name, control_id
```

The method displays the common file open dialog, enabling you to specify the name and directory of the CSV file to be created. The default name is **controlIds.csv**.

The source for the **extractControlIdsCSV** method is provided, should you want to vary the calling approach.

```
vars
    file : File;
    cmd  : CMDFileSave;
    subs : SchemaNDict;
    scm  : Schema;
```

```
begin
    create cmd;
    cmd.fileName := "controlIds.csv";
    if cmd.open = 0 then
        create file transient;
        file.openOutput(cmd.fileName);
        create subs transient;
        subs.add(currentSchema);
        while subs.size() > 0 do
            scm := subs.first;
            scm.extractControlIdsCSVforSchema(file);
            subs.remove(scm);
            scm.getSubschemas(subs);
        endwhile;
        file.close;
    endif;
epilog
    delete cmd;
    delete file;
    delete subs;
end;
```

extractControlIdsCSVforSchema

Signature `extractControlIdsCSVforSchema(file: File) updating;`

The **extractControlIdsCSVforSchema** method of the **Schema** class writes a comma-separated value (CSV) entry containing the generated Windows control id for each JADE control into the file specified by the **file** parameter.

The control ids are used by testing tools to identify required elements on a form. The id for a control on a form created in the JADE Painter is retained for the lifetime of the form (unless the control is deleted and re-added using the JADE Painter).

There is an entry for each control on every form in the schema (the receiver) but not in its subschemas, in the following format.

```
schema_name, form_name, control_name, control_id
```

The following example shows the use of the **extractControlIdsCSVforSchema** method.

```
vars
    file : File;
begin
    create file;
    file.fileName := "C:\controlIds.csv";
    rootSchema.extractControlIdsCSVforSchema(file);
epilog
    delete file;
end;
```

findClassInBranch

Signature `findClassInBranch(clsName: String): Type;`

The **findClassInBranch** method of the **Schema** class returns a reference to the type of the class specified in the **clsName** parameter.

findClassInSubschema

Signature `findClassInSubschema(clsName: String): Class;`

The **findClassInSubschema** method of the **Schema** class returns a reference to the class specified in the **clsName** parameter.

findFormForLocale

Signature
```
findFormForLocale(formName: String;
                  lcid:     Integer): Form;
```

The **findFormForLocale** method of the **Schema** class returns a reference to the instance of the form specified in the **formName** parameter for the locale specified in the **lcid** parameter from the current schema and all of its subschemas.

findFormForLocaleInAllSchemas

Signature
```
findFormForLocaleInAllSchemas(formName: String;
                              lcid:     Integer): Form;
```

The **findFormForLocaleInAllSchemas** method of the **Schema** class returns a reference to the instance of the form specified in the **formName** parameter for the locale specified in the **lcid** parameter from all schemas, starting from the Root Schema.

findFormForLocaleInSupers

Signature
```
findFormForLocaleInSupers(formName: String;
                          lcid:     Integer): Form;
```

The **findFormForLocaleInSupers** method of the **Schema** class returns a reference to the instance of the form specified in the **formName** parameter for the locale specified in the **lcid** parameter from the current schema and all of its superschemas.

findGlobalConstantInBranch

Signature `findGlobalConstantInBranch(conName: String): Constant;`

The **findGlobalConstantInBranch** method of the **Schema** class returns a reference to the instance of the global constant specified in the **conName** parameter.

findMeForm

Signature `findMeForm(formName: String): Form updating;`

The **findMeForm** method of the **Schema** class returns a reference to the form specified in the **formName** parameter from the base locale.

An exception is raised if this method is invoked from a server method.

findName

Signature `findName(str: String): Type;`

The **findName** method of the **Schema** class returns a reference to the type of the class or primitive type specified in the **str** parameter.

findProperty

Signature `findProperty(str: String): Property;`

The **findProperty** method of the **Schema** class returns a reference to the property specified in the **str** parameter.

findType

Signature `findType(aNumber: Integer): Type;`

The **findType** method of the **Schema** class returns a reference to the primitive type that corresponds to the number specified in the **aNumber** parameter.

generateWSDL

Signature
```
generateWSDL(applicationName:     String;
             exposureName:        String;
             fileName:            String;
             urlScheme:           String;
             urlMachineName:      String;
             urlVirtualDirectory: String;
             urlSupportLibrary:   String) updating;
```

The **generateWSDL** method of the **Schema** class creates a WSDL file for an existing JADE Web service provider application.

The **generateWSDL** method parameters are listed in the following table.

Parameter	Description
applicationName	The name of an existing Web service application.
exposureName	The name of an existing Web service exposure definition.
fileName	The name of the WSDL file to be created, which must be a valid file name.
urlScheme	The Web service protocol; **http** by default and **tcp** for direct Web services.
urlMachineName	The machine name or Internet Protocol (IP) address to which the Web service requests should be directed.
urlVirtualDirectory	The virtual directory where the support library resides (as defined in IIS or Apache).
urlSupportLibrary	The support library (**jadehttp**) for communication with IIS or Apache.

If the validation or update fails, an exception is raised. The **extendedErrorText** property contains details of the exception.

The following example shows the use of the **generateWSDL** method.

```
currentSchema.generateWSDL("CustomerServicesApp","ErewhonCustomer",
                           "d:\wsdls\erewhoncustomer.wsdl", "http",
                           "localhost", "jade", "jadehttp.dll");
```

getAllBaseLocales

Signature `getAllBaseLocales(returnColl: ObjectArray input);`

The **getAllBaseLocales** method of the **Schema** class adds all base (non-clone) locales in the current schema to the array specified in the **returnColl** parameter.

The object array is not cleared before instances are added.

getAllClasses

Signature `getAllClasses(includeSystemClasses: Boolean): ClassColl;`

The **getAllClasses** method of the **Schema** class returns a collection containing a reference to all classes in the current schema and its superschemas.

Set the **includeSystemClasses** parameter to **true** to specify that references to system classes are included in the returned collection.

getAllFormTranslations

Signature
```
getAllFormTranslations(form:     Form;
                       formList: FormOrdList input);
```

The **getAllFormTranslations** method of the **Schema** class gets a collection of all translations of the form specified in the **form** parameter from the collection specified in the **formList** parameter, excluding the specified form itself.

getAllInheritedLocales

Signature `getAllInheritedLocales(returnColl: ObjectArray input);`

The **getAllInheritedLocales** method of the **Schema** class adds all locales inherited by the current schema to the array specified in the **returnColl** parameter. The object array is not cleared before instances are added.

getAllLocales

Signature `getAllLocales(returnColl: ObjectArray input);`

The **getAllLocales** method of the **Schema** class adds all locales in the current schema and superschemas to the array specified in the **returnColl** parameter. The object array is not cleared before instances are added.

getAllLocalLocales

Signature `getAllLocalLocales(returnColl: ObjectArray input);`

The **getAllLocalLocales** method of the **Schema** class adds only those locales defined in the current schema to the array specified in the **returnColl** parameter. The object array is not cleared before instances are added.

getAllSystemLocales

Signature `getAllSystemLocales(oa: ObjectArray);`

The **getAllSystemLocales** method of the **Schema** class adds an instance of the **LocaleNameInfo** class for each locale supported by the operating system to the object array specified in the **oa** parameter.

The object array is not cleared before instances are added; that is, it is the responsibility of the caller to delete the **LocaleNameInfo** objects.

getAllRpsMappings

Signature `getAllRpsMappings(returnColl: ObjectArray input);`

The **getAllRpsMappings** method of the **Schema** class adds all RPS mappings that are instances of the **RelationalView** class defined in the current schema to the array specified in the **returnColl** parameter.

The object array is not cleared before instances are added.

getAppliedPatches

Signature `getAppliedPatches(): String;`

The **getAppliedPatches** method of the **Schema** class returns a string containing all patches applied by a schema load to system schemas in your database. (See also the **getUserAppliedPatches** method.)

The format of the string returned by this method is ***schema-name* Patches:** followed by one or more lines in the following format.

file-type,*file-name*,*jade-version*,*timestamp*[,*file-version-tag*]

In the format of the returned string:

- *schema-name* is the name of the system schema to which the patch was applied
- *file-type* is **Schema** or **DDB** (or .**DDX**, depending on the value of the **Use DDX style (xml format) as Default instead of DDB** check box on the **Schema** sheet of the Preferences dialog)
- *file-name* is the full (absolute) path name of the file that was loaded
- *jade-version* is the JADE version number (for example, **7.1.03**)
- *timestamp* is the date and time of the file load
- *file-version-tag* is the optional version tag value obtained from the **JadeFiletypeVersiontag** line in the patchset file (for example, **JadeFiletypeVersiontag SCM "7.1.03.024.001";**).

This is repeated for each patch that was applied.

Note All commas in the *file-name* and *file-version-tag* are replaced by underscore characters.

An example of a string returned by this method is as follows.

```
RootSchema Patches:
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:22,7.1.03.024.001
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:30,7.1.03.025.001
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:31,7.1.03.043.001
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:32,7.1.03.051.001
```

```
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:34,7.1.03.061.001
Schema,C:\Jade7103test\bin\hotfix76.scm,7.1.03,2015/02/23 09:47:22,7.1.03.082.001
Schema,C:\Jade7103test\bin\hotfix76.scm,7.1.03,2015/02/23 09:47:25,7.1.03.083.001
Schema,C:\Jade7103test\bin\hotfix76.scm,7.1.03,2015/02/23 09:47:25,7.1.03.089.001
JadeSchema Patches:
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:17,7.1.03.032.001
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:34,7.1.03.052.001
Schema,C:\Jade7103test\bin\hotfix#42.scm,7.1.03,2014/12/20 11:06:37,7.1.03.061.001
Schema,C:\Transfer\hotfix#42.scm,7.1.03,2014/12/21 10:24:43,7.1.03.061.002
Schema,C:\Jade7103test\bin\hotfix76.scm,7.1.03,2015/02/23 09:47:21,7.1.03.069.001
```

getBaseLocalesLocal

Signature `getBaseLocalesLocal(returnColl: ObjectArray input);`

The **getBaseLocalesLocal** method of the **Schema** class populates the collection specified by the **returnColl** parameter with the base locales defined in the receiving schema.

getCategory

Signature `getCategory(catName: String): ConstantCategory;`

The **getCategory** method of the **Schema** class returns a reference to the global constant category specified in the **catName** parameter.

The search for the specified category is performed in the receiver and all of its superschemas.

getClass

Signature `getClass(name: String): Class;`

The **getClass** method of the **Schema** class returns a reference to the class specified in the **name** parameter.

getClassByNumber

Signature `getClassByNumber(classNo: Integer): Class;`

The **getClassByNumber** method of the **Schema** class returns a reference to the class specified by the class number in the **classNo** parameter.

getConstant

Signature `getConstant(name: String): GlobalConstant;`

The **getConstant** method of the **Schema** class returns a reference to the global constant specified in the **name** parameter.

getConstantCategory

Signature `getConstantCategory(name: String): ConstantCategory;`

The **getConstantCategory** method of the **Schema** class returns a reference to the global constant category specified in the **name** parameter from the receiver. (Use the **getCategory** method if you also want superschemas of the receiver to be searched for the specified global constant category.)

getControlClasses

Signature `getControlClasses(clsNDict: ClassColl input): ClassColl;`

The **getControlClasses** method of the **Schema** class returns a reference to all subclasses of the **Control** class for the schema.

This method appends the **Control** subclasses to the collection specified in the **clsNDict** parameter and returns the full collection. The collection is not cleared before instances are added.

getCurrentLocaleId

Signature `getCurrentLocaleId(): Integer;`

The **getCurrentLocaleId** method of the **Schema** class returns the current locale identifier (LCID) of the application process when enhanced locale support is enabled; otherwise it returns the identifier of the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true** and this method is called after the **Application** class **setJadeLocale** method, the returned value is the new value returned in the **setJadeLocale** method **requestedLcid** parameter. In addition, when enhanced locale support is enabled, the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant enables you to retrieve information from the session locale without having to save the initial locale for the call.

Note The current **localeID** value that is returned may be different from the **number** property in the **Application** class **currentLocale** property, if the **setJadeLocale** method has been called with a locale not found in the schema.

---

getDefaultLocale

Signature `getDefaultLocale(): Locale;`

The **getDefaultLocale** method of the **Schema** class returns a reference to the default locale for the schema used as the receiver.

getExternalDatabase

Signature `getExternalDatabase(dbName: String): ExternalDatabase;`

The **getExternalDatabase** method of the **Schema** class returns a reference to the shared transient instance of the external database specified in the **dbName** parameter or it returns **null** if there is no active external database with the specified name.

getFormatAnywhereInPath

Signature `getFormatAnywhereInPath(formatName: String): LocaleFormat;`

The **getFormatAnywhereInPath** method of the **Schema** class returns a reference to the locale format specified in the **formatName** parameter for the current version of schemas in the schema path; that is, the receiver, its superschemas, and subschemas.

getFormatAnywhereInPathLatest

Signature `getFormatAnywhereInPathLatest(formatName: String): LocaleFormat;`

The **getFormatAnywhereInPathLatest** method of the **Schema** class returns a reference to the locale format specified in the **formatName** parameter for the latest version of schemas in the schema path; that is, the receiver, its superschemas, and subschemas.

getFormatAnywhereInSubs

Signature `getFormatAnywhereInSubs(formatName: String): LocaleFormat;`

The **getFormatAnywhereInSubs** method of the **Schema** class returns a reference to the locale format specified in the **formatName** parameter from any locale in the current version of the receiver or any of its subschemas.

getFormatAnywhereInSubsLatest

Signature `getFormatAnywhereInSubsLatest(formatName: String): LocaleFormat;`

The **getFormatAnywhereInSubsLatest** method of the **Schema** class returns a reference to the locale format specified in the **formatName** parameter from any locale in the latest version of the receiver or any of its subschemas.

getFunction

Signature `getFunction(name: String): Function;`

The **getFunction** method of the **Schema** class returns a reference to the function specified in the **name** parameter.

getGlobalClass

Signature `getGlobalClass(): Class;`

The **getGlobalClass** method of the **Schema** class returns a reference to the **Global** class object for the schema.

getGlobalConstant

Signature `getGlobalConstant(constName: String): GlobalConstant;`

The **getGlobalConstant** method of the **Schema** class returns a reference to the global constant specified in the **constName** parameter.

getHtmlDocumentSource

Signature `getHtmlDocumentSource(htmlDocumentName: String): String;`

The **getHtmlDocumentSource** method of the **Schema** class returns the HTML source of the HTML document specified in the **htmlDocumentName** parameter.

This method returns null (**""**) if an HTML document with the specified name does not exist.

getImportedClass

Signature `getImportedClass(clsName: String): JadeImportedClass;`

The **getImportedClass** method of the **Schema** class returns a reference to a class imported as part of an imported package.

The imported class object has a reference to the exported class that was exported as part of the exported package. Finally, the exported class object has a reference to the original instance of the **Class** class, as shown in the following code example.

```
vars
    impClass : JadeImportedClass;
    expClass : JadeExportedClass;
    origClass : Class;
begin
    impClass := currentSchema.getImportedClass("Diary");
    expClass := impClass.exportedClass;
    origClass := expClass.originalClass;
    // Processing of the original Class instance for the Diary class
end;
```

getImportedJadeinterface

Signature `getImportedJadeInterface(infName: String): JadeImportedInterface;`

The **getImportedJadeInterface** method of the **Schema** class returns a reference to an interface imported as part of an imported package.

The imported interface object has a reference to the exported interface that was exported as part of the exported package. Finally, the exported interface object has a reference to the original instance of the **JadeInterface** class, as shown in the following code example.

```
vars
    impInterface : JadeImportedInterface;
    expInterface : JadeExportedInterface;
    origInterface : JadeInterface;
begin
    impInterface := currentSchema.getImportedJadeInterface("MeetingIF");
    expInterface := impInterface.exportedInterface;
    origInterface := expInterface.originalInterface;
    // Processing of the original instance for the MeetingIF interface
end;
```

getInheritedFormats

Signature `getInheritedFormats(returnColl: ObjectArray input);`

The **getInheritedFormats** method of the **Schema** class adds all formats inherited from superschemas of the current schema to the array specified in the **returnColl** parameter.

The object array is not cleared before instances are added.

getInheritedXlatableStrings

Signature `getInheritedXlatableStrings(locName: String; returnColl: ObjectArray input);`

The **getInheritedXlatableStrings** method of the **Schema** class adds all translatable strings from all superschemas of the current schema defined for the locale specified in the **locName** parameter to the array specified in the **returnColl** parameter.

The object array is not cleared before instances are added.

getJadeInterface

Signature `getJadeInterface(name: String): JadeInterface;`

The **getJadeInterface** method of the **Schema** class returns a reference to the interface specified in the **name** parameter.

getLibrary

Signature `getLibrary(name: String): Library;`

The **getLibrary** method of the **Schema** class returns a reference to the library with the name that is specified in the **name** parameter string.

getLocalClass

Signature `getLocalClass(className: String): Class;`

The **getLocalClass** method of the **Schema** class returns a reference to the class specified in the **className** parameter from the current schema.

getLocale

Signature `getLocale(localeName: String): Locale;`

The **getLocale** method of the **Schema** class returns a reference to the locale (either local or inherited) specified in the **localeName** parameter from the current schema and all of its subschemas.

getLocaleCurrencyInfo

Signature `getLocaleCurrencyInfo(lcid: Integer; info: CurrencyFormat input);`

The **getLocaleCurrencyInfo** method of the **Schema** class gets the currency formatting information from the **CurrencyFormat** class for the locale specified in the **lcid** parameter.

Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When the **EnhancedLocaleSupport** parameter on the database node is set to **true**, this method returns the currency information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocaleDateInfo

Signature

```
getLocaleDateInfo(lcid: Integer;
                  info: DateFormat input);
```

The **getLocaleDateInfo** method of the **Schema** class gets the date formatting information from the **DateFormat** class for the locale specified in the **lcid** parameter.

Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client. For example, if the locale of your application server is set to English (United Kingdom), which has a default short date format of **dd/mm/yyyy**, and it has been overridden with a short date format of ***yyyy-mm-dd***, this is returned in the default **dd/mm/yyyy** format.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When the **EnhancedLocaleSupport** parameter on the database node is set to **true**, this method returns the date information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocaleFullInfo

Signature

```
getLocaleFullInfo(lcid: Integer;
                  info: LocaleFullInfo input);
```

The **getLocaleFullInfo** method of the **Schema** class gets the full formatting information from the **LocaleFullInfo** class for the locale specified in the **lcid** parameter.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When you set the **EnhancedLocaleSupport** parameter on the database node to **true**, the **getLocaleFullInfo** method returns full locale information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocaleInSubschemas

Signature `getLocaleInSubschemas(localeName: String): Locale;`

The **getLocaleInSubschemas** method of the **Schema** class returns a reference to the local or inherited locale specified in the **localeName** parameter from the subschemas of the current schema.

getLocaleLocal

Signature `getLocaleLocal(localeName: String): Locale;`

The **getLocaleLocal** method of the **Schema** class returns a reference to the locale specified in the **localeName** parameter from the current schema.

Note This method returns both base and clone locales.

getLocaleNameInfo

Signature
```
getLocaleNameInfo(lcid: Integer;
                  info: LocaleNameInfo input);
```

The **getLocaleNameInfo** method of the **Schema** class gets the name formatting information from the **LocaleNameInfo** class for the locale specified in the **lcid** parameter. Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant enables you to retrieve information from the session locale without having to save the initial locale for the call.

When the **EnhancedLocaleSupport** parameter on the database node is set to **true**, this method returns the locale name information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocaleNumericInfo

Signature
```
getLocaleNumericInfo(lcid: Integer;
                     info: NumberFormat input);
```

The **getLocaleNumericInfo** method of the **Schema** class gets the numeric formatting information from the **NumberFormat** class for the locale specified in the **lcid** parameter. Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When the **EnhancedLocaleSupport** parameter on the database node is set to **true**, this method returns the numeric formatting information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocaleTimeInfo

Signature `getLocaleTimeInfo(lcid: Integer;`
`info: TimeFormat input);`

The **getLocaleTimeInfo** method of the **Schema** class gets the time formatting information from the **TimeFormat** class for the locale specified in the **lcid** parameter. Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When the **EnhancedLocaleSupport** parameter on the database node is set to **true**, this method returns the time information with regional overrides suppressed, unless the **lcid** parameter is set to zero (**0**) when the current thread locale is the session locale or the **lcid** parameter is set to the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant (which enables you to easily code calls for information about the session locale without having to change the current locale).

Regardless of the value of the **EnhancedLocaleSupport** parameter, if you set the **lcid** parameter to zero (**0**), the information is returned for the current thread locale.

getLocalFormats

Signature `getLocalFormats(returnColl: ObjectArray input);`

The **getLocalFormats** method of the **Schema** class adds all of the user-defined formats for the receiving schema (that is, the schema that is the receiver of the method) to the array specified in the **returnColl** parameter.

The object array is not cleared before instances are added.

getLocalLocaleInSubschemas

Signature `getLocalLocaleInSubschemas(localeName: String): Locale;`

The **getLocalLocaleInSubschemas** method of the **Schema** class returns a reference to the locale specified in the **localeName** parameter from the subschemas of the current schema.

getLocalPrimitive

Signature `getLocalPrimitive(primName: String): PrimType;`

The **getLocalPrimitive** method of the **Schema** class returns a reference to the primitive type specified in the **primName** parameter from the current schema.

getName

Signature `getName(): String;`

The **getName** method of the **Schema** class returns the name of the receiver schema as a string.

getOidForObject

Signature `getOidForObject(obj: Any): String;`

The **getOidForObject** method of the **Schema** class returns the object identifier (oid) of the object specified in the **obj** parameter. If the **obj** parameter is not an object reference, an exception is raised.

This method can be called for object references obtained by the **getPropertyValue** method of the **Object** class when the class of the object is not visible in the current schema.

getPrimitive

Signature `getPrimitive(name: String): PrimType;`

The **getPrimitive** method of the **Schema** class returns a reference to the primitive type with the name specified in the **name** parameter.

getRelationalView

Signature `getRelationalView(name: String): RelationalView;`

The **getRelationalView** method of the **Schema** class returns a reference to the relational view in the current schema with the name specified in the **name** parameter.

getRpsMapping

Signature `getRpsMapping(name: String): RelationalView;`

The **getRpsMapping** method of the **Schema** class returns a reference that is an instance of the **RelationalView** class, to the RPS mapping in the current schema with the name specified in the **name** parameter.

getSchema

Signature `getSchema(name: String): Schema;`

The **getSchema** method of the **Schema** class returns a schema with the name specified in the **name** parameter.

The schema can be any user-defined schema or system schema; for example, the **RootSchema** or **JadeReportWriterSchema**.

getSubschema

Signature `getSubschema(name: String): Schema;`

The **getSubschema** method of the **Schema** class returns a reference to a user-defined subschema with the name specified in the **name** parameter.

getSubschemas

Signature `getSubschemas(subs: SchemaNDict input);`

The **getSubschemas** method of the **Schema** class adds all subschemas of the receiving schema to the dictionary specified in the **subs** parameter.

As this method is not recursive, only the immediate (that is, the direct descendant) subschemas of the receiver are returned.

getUserAppliedPatches

Signature `getUserAppliedPatches(): String;`

The **getUserAppliedPatches** method of the **Schema** class returns a string containing all patches applied to user schemas in your database. (See also the **getAppliedPatches** method, which returns a string of the patches applied to system schemas.)

The format of the string returned by this method is ***schema-name* Patches:** followed by one or more lines in the following format.

file-type,*file-name*,*jade-version*,*timestamp*[,*file-version-tag*]

In the format of the returned string:

- *schema-name* is the name of the user schema to which the patch was applied
- *file-type* is **Schema** or **DDB** (or .**DDX**, depending on the value of the **Use DDX style (xml format) as Default instead of DDB** check box on the **Schema** sheet of the Preferences dialog)
- *file-name* is the full (absolute) path name of the file that was loaded
- *jade-version* is the JADE version number (for example, **7.1.03**)
- *timestamp* is the date and time of the file load
- *file-version-tag* is the optional version tag value obtained from the **JadeFiletypeVersiontag** line in the patchset file (for example, **JadeFiletypeVersiontag SCM "7.1.03.024.001";**).

This is repeated for each patch that was applied.

Note All commas in the *file-name* and *file-version-tag* are replaced by underscore characters.

You can call the **getUserAppliedPatches** method at any time to determine the files (for example, hot fixes) that have been loaded into user schemas in your JADE database. Alternatively, call the **resetUserAppliedPatches** method to reset the information about patches applied to a specified user schema to null (**""**).

getUserFormat

Signature `getUserFormat(name: String): LocaleFormat;`

The **getUserFormat** method of the **Schema** class returns a reference to the user format with the name specified in the **name** parameter.

getWebServiceConsumerNames

Signature `getWebServiceConsumerNames(names: JadeIdentifierArray input);`

The **getWebServiceConsumerNames** method of the **Schema** class populates the array specified by the **names** parameter with the names of Web Service consumers defined in the receiving schema.

globalException

Signature `globalException(exception: Exception): Integer;`

The **globalException** method of the **Schema** class is the default global exception handler for GUI applications and passes control to the **defaultHandler** method on the exception instance; that is, it calls **exception.defaultHandler**.

Note Do not call this method directly, as it is automatically armed by JADE for GUI processes.

See also "Handling Exceptions", in Chapter 3 of the *JADE Developer's Reference* and the **Schema** class **nonGUIGlobalExceptionHandler** and **Exception** class **defaultHandler** methods.

importWSDL

Signature
```
importWSDL(wsdlFileName:                String;
           consumerName:                String;
           generateAsynchronousMethods: Boolean;
           useNewPrimitiveTypes:        Boolean;
           superclassName:              String;
           classNamePrefix:             String;
           methodNamePrefix:            String;
           propertyNamePrefix:          String;
           renameFileName:              String): String;
```

The **importWSDL** method of the **Schema** class creates a Web service consumer by importing a specified WSDL file.

This method returns an empty string if it executes successfully or an error message if it fails.

The **importWSDL** method parameters are listed in the following table.

Parameter	Description
wsdlFileName	The name of the WSDL file on which to base the Web service consumer. The name can be a URL if the WSDL is available on the Web.
consumerName	The name of the Web service consumer, which must begin with an uppercase letter.

Parameter	Description
generateAsynchronousMethods	**true** if methods for consuming the Web service asynchronously are generated in addition to the methods for synchronous execution, and **false** otherwise.
useNewPrimitiveTypes	**true** if methods generated from the WSDL use the primitive types **Integer64**, **Byte**, and **TimeStampInterval** where appropriate, and **false** otherwise.
superclassName	The name of the superclass of the classes created for the Web service consumer. If an empty string is specified, the superclass is **Object**.
classNamePrefix	The prefix for the classes created for the Web service consumer. If an empty string is specified, the class names do not have a prefix.
methodNamePrefix	The prefix for the methods created for the Web service consumer. If an empty string is specified, the method names do not have a prefix.
propertyNamePrefix	The prefix for the properties created for the Web service consumer. If an empty string is specified, the property names do not have a prefix.
renameFileName	Reserved for future use. (This parameter will enable generated JADE classes to be renamed.)

You can import a JADE or external WSDL file that is in **Document/Literal Bare** or **Document/Literal Wrapped** SOAP message style format.

isLocalLocale

Signature `isLocalLocale(l: Locale): Boolean;`

The **isLocalLocale** method of the **Schema** class returns **true** if the locale specified in the **l** parameter is local to the current schema.

loadHTMLDocuments

Signature
```
loadHTMLDocuments(pathName: String;
                  reload:   Boolean): String;
```

The **loadHTMLDocuments** method of the **Schema** class processes all files in the folder specified in the **pathName** parameter, to enable you to load or reload multiple HTML documents.

The names of the HTML documents are obtained from the file names, excluding the extension and the path. For example, if the file name is **c:\documents\header.htm**, the document name is **Header** (with an uppercase first character). If this results in the name being longer than 100 characters, it is truncated to 100 characters.

If a document already exists in the schema and the **reload** parameter is set to **true**, the document is updated. If a document does not exist in the schema, the document is created and updated. An exception is raised if the **pathName** parameter contains a null value or the specified directory cannot be located.

Note As no validation is done to determine if the file is a valid HTML file, it is your responsibility to ensure that the files in the specified folder are valid HTML documents.

makeLocaleNameFromId

Signature `makeLocaleNameFromId(lcid: Integer): String;`

The **makeLocaleNameFromId** method of the **Schema** class returns the name of the locale specified in the **lcid** parameter. Set the value of the **lcid** parameter to zero (**0**) if you want to return information for the current locale.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file is not defined or it is set to **false**, inconsistent results could be returned to the application server when running in JADE thin client mode and there are regional overrides, as all overrides on the application server are suppressed.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the **JadeLocaleIdNumbers** category **LCID_SessionWithOverrides** global constant enables you to retrieve information from the session locale without having to save the initial locale for the call.

nonGUIGlobalExceptionHandler

Signature `nonGUIGlobalExceptionHandler(e: Exception): Integer;`

The **nonGUIGlobalExceptionHandler** method of the **Schema** class is the default global exception handler for non-GUI applications.

This method logs exception details from non-GUI application to the exception log file of the current application (for example, **MyApp.log**), aborts any persistent or transient transaction, and then returns **Ex_Abort_Action**.

Note Do not call this method directly, as it is automatically armed by JADE for non-GUI processes.

See also "Handling Exceptions", in Chapter 3 of the *JADE Developer's Reference* and the **Schema** class **globalException** method and **Exception** class **defaultHandler** method.

regenerateRelationalView

Signature
```
regenerateRelationalView(viewName:        String;
                         excludedClasses: ClassColl;
                         rootClass:       Class);
```

The **regenerateRelationalView** method of the **Schema** class builds the relational view specified in the **viewName** parameter. Call this method in your application logic to dynamically build a relational view at run time after changes to the schema, instead of using the Relational Views Wizard.

Use the **excludedClasses** parameter to specify the collection of classes that you want excluded from the relational view and the **rootClass** parameter to specify the root class of the view. If the view does not exist, a new view is created. If a view of the specified name already exists, it is deleted and replaced with the new view.

The following example shows the use of the **regenerateRelationalView** method.

```
makeNewView();
vars
    clsColl : ClassColl;
begin
    create clsColl transient;
    Object.allSubclassesUpToSchema(rootSchema, clsColl);
    currentSchema.regenerateRelationalView("MyView", clsColl, Company);
    delete clsColl;
end;
```

The method in this example creates or regenerates a relational view identical to that generated by using the Relational View Wizard (assuming there is a **Company** class) and accepting all default values in the wizard steps summarized in the following table, unless specified in the following list.

Step	Comment
Naming Your Relational View	**MyView** name is specified
Specifying Relational View Options	**Include Super Schema Classes** check box is checked
Selecting Classes and User-Defined Tables for Your Relational View	
Setting the Root Class	**Company** class is selected as the root class
Setting the Default Included Object Features	
Setting the Visibility of Protected Features	**Include All** option is selected
Setting the Visibility of Derived Tables	
Refining the Visibility of Class Features	
Building Your Relational View	The **Finish** button is clicked, to generate the relational view

For details about creating a relational view and the steps provided by the Relational View Wizard, see "Adding a Relational View", in Chapter 9 of the *JADE Development Environment User's Guide*.

removeWebConsumer

Signature `removeWebConsumer(consumerName: String): String;`

The **removeWebConsumer** method of the **Schema** class removes the Web service consumer with the name specified in the **consumerName** parameter. The **removeWebConsumer** method returns an empty string if it executes successfully or it returns an error message if it fails.

reorgInProgress

Signature `reorgInProgress(): Boolean;`

The **reorgInProgress** method of the **Schema** class returns **true** if a reorganization is currently "actively" in progress; otherwise it returns **false** (that is, the reorganization is in dual-update mode and waiting for the transition).

reorgIsWaitingForTransition

Signature `reorgIsWaitingForTransition(): Boolean;`

The **reorgIsWaitingForTransition** method of the **Schema** class returns **true** if a reorganization is in dual-update mode and waiting for the transition; otherwise it returns **false**.

resetUserAppliedPatches

Signature `resetUserAppliedPatches(schemaName: String): Boolean updating;`

The **resetUserAppliedPatches** method of the **Schema** class resets the information about all patches applied to user schema specified in the **schemaName** parameter to null (**""**).

This method returns **true** if the reset operation was successful or it returns **false** if the operation fails (that is, a schema that matches the value specified in the **schemaName** parameter does not exist).

See also the **getUserAppliedPatches** method, which returns information about the patches applied to all user schemas in the JADE database.

setHtmlDocumentSource

Signature `setHtmlDocumentSource(htmlDocumentName: String;`
`documentSource: String): String;`

The **setHtmlDocumentSource** method of the **Schema** class sets the HTML source to the value specified in the **documentSource** parameter for the HTML document specified in the **htmlDocumentName** parameter.

This method returns null (**""**) if the update action was successful or it returns an error message if the HTML document specified in the **htmlDocumentName** parameter does not exist, if there are errors in the structure of the source, or if the **JadeHTMLClass** class of the document is in use.

Tip If your application uses the same schema but with different HTML source, you can store the HTML in another object before loading a schema and then replacing it following the schema load process. This method enables you to set the HTML source to the required value following a schema load.

Call the **getHtmlDocumentSource** method to obtain the HTML source of a specified HTML document.

withAllSubschemas

Signature `withAllSubschemas(subs: SchemaColl input);`

The **withAllSubschemas** method of the **Schema** class adds the current schema and all of its subschemas to the collection specified in the **subs** parameter. The collection is not cleared before instances are added.

withAllSuperschemas

Signature `withAllSuperschemas(): SchemaColl;`

The **withAllSuperschemas** method of the **Schema** class returns a reference to a collection of the current schema and all of its superschemas.

SchemaEntity Class

The **SchemaEntity** class is the superclass of a number of classes that participate in the definition of a schema.

For details about the constants, properties, and methods defined in the **SchemaEntity** class, see "SchemaEntity Class Constants", "SchemaEntity Properties", and "SchemaEntity Method", in the following subsections.

Inherits From: Object

Inherited By: Database, DbFile, DbServer, Feature, Library, Locale, Type

SchemaEntity Class Constants

The constants provided by the **SchemaEntity** class are listed in the following table.

Constant	Integer Value	Description
Access_Protected	#'2'	Protected access
Access_Public	#'0'	Public access
Access_ReadOnly	#'1'	Read-only access
SubAccess_Public	#'0'	Public subschema access
SubAccess_SubschemaHidden	#'3'	No subschema access

SchemaEntity Properties

The properties defined in the **SchemaEntity** class are summarized in the following table.

Property	Description
abstract	Specifies whether the schema entity is abstract
access	Contains the type of access to the schema entity
helpKeyword	Contains text used to access the help file while the schema entity is selected
name	Contains the name of the schema entity
number	Contains the unique number of the schema entity
subAccess	Contains the level of accessibility of the schema entity from a subschema
text	Contains the descriptive text for a schema entity

abstract

Type: Boolean

The read-only **abstract** property of the **SchemaEntity** class specifies whether the entity is abstract.

Abstract classes factor out behavior that is common to a number of classes; for example, the **Btree** and the **Dictionary** collection classes. Abstract classes can have abstract methods defined for them. (Abstract methods are those that have no logic associated with them but must be implemented in the subclasses of the abstract class.)

access

Type: Character[1]

The read-only **access** property of the **SchemaEntity** class contains the type of access to the schema entity. The access for an entity is specified in the Class Definition, Define Attribute, Jade Method Definition, External Method Definition, or Add Condition dialog when the entity is defined.

The types of access are listed in the following table.

Class Constant	Value	Description	Applies to...
Access_Protected	#'2'	Protected access	Classes, methods, and properties
Access_Public	#'0'	Public access	Classes, methods, and properties
Access_ReadOnly	#'1'	Read-only access	Properties only

name

Type: String[100]

The read-only **name** property of the **SchemaEntity** class contains the name of the schema entity. The code fragment in the following example shows the use of the **name** property.

```
foreach obj in coll do
    if not (obj = self or obj = caller) then
        count.bump;
        found := true;
        if count = 1 then
            write 'Class - ' & cls.name;
        endif;
        write '  Transient - ' & obj.String;
    endif;
endforeach;
```

number

Type: Integer

The read-only **number** property of the **SchemaEntity** class contains the unique number of the schema entity.

subAccess

Type: Character[1]

The read-only **subAccess** property of the **SchemaEntity** class contains the level of accessibility of the schema entity from a subschema. The types of access are listed in the following table.

Class Constant	Value	Description
SubAccess_Public	#'0'	Public subschema access
SubAccess_SubschemaHidden	#'3'	No subschema access

For details about the **subschemaHidden** option, see "subschemaHidden Option" under "Controlling the User of Elements in Other Schemas", in Chapter 1 of the *JADE Development Environment User's Guide*.

text

Type: String

The read-only **text** property of the **SchemaEntity** class contains the descriptive text of the receiver that was entered in the text editor window (accessed by using the **Text** command from the Classes or Schema menu) for the receiver, if applicable. For example, the text can be a description of the object, the date created or amended, and by whom.

SchemaEntity Methods

The methods defined in the **SchemaEntity** class are summarized in the following table.

Method	Description
getName	Returns the name of the receiver schema as a string
getPatchNumber	Returns the patch version number of the receiver

getName

Signature `getName(): String;`

The **getName** method of the **SchemaEntity** class returns the name of the receiver schema entity as a string.

getPatchNumber

Signature `getPatchNumber(): Integer;`

The **getPatchNumber** method of the **SchemaEntity** class returns the patch version number of the receiver.

This method returns zero (**0**) if there is no open patch number.

SchemaEntityNumberDict Class

The **SchemaEntityNumberDict** class is used to store references to instances of subclasses of the **SchemaEntity** class.

The key of the **SchemaEntityNumberDict** class is the **number** property of the **SchemaEntity** class.

Inherits From: MemberKeyDictionary

Inherited By: (None)

Script Class

The **Script** class is the superclass of a number of classes that represent schema entities that have source code, which must be compiled successfully for the entity to function correctly. The most obvious example of a schema entity that has source code is a method. However, there are many other entities such as translatable strings and global constants with source code definitions.

For details about the properties and methods defined in the **Script** class, see "Script Properties" and "Script Methods", in the following subsections.

Inherits From: Feature

Inherited By: Constant, Routine

Script Properties

The properties defined in the **Script** class are summarized in the following table.

Property	Contains the …
compiledOK	Value **false**
errorCode	Compiler error code of the most recent compilation
errorLength	Length of the token found in error in the most recent compilation
errorPosition	Offset position of the token found in error in the most recent compilation
status	Compilation status of the receiver
warningCount	Number of warnings generated during the most recent compile

compiledOK

Type: Boolean

The **compiledOK** property of the **Script** class always has the value **false**.

Use the **inError** and **notCompiled** methods to determine whether a method is in error or has not been compiled.

errorCode

Type: Integer

The public **errorCode** property of the **Script** class contains the compiler error code of the most-recent attempt to compile the source code.

The value of the **errorCode** property is zero (**0**) if no attempt to compile the source code is made or the source code has been compiled successfully.

In the following example, a JADE method fails to compile because a semicolon (**;**) is required before the token **end** in the final line. The resulting value of the **errorCode** property is 7052.

```
test();
vars
begin
    write "Hello World"
end;
```

errorLength

Type: Integer

The public **errorLength** property of the **Script** class contains the length of the token found to be in error in most-recent attempt to compile the source code. The value of the **errorLength** property is zero (**0**) if no attempt to compile the source code is made or the source code has been compiled successfully.

In the following example, a JADE method fails to compile because a semicolon (**;**) is required before the token **end** in the final line. The resulting value of the **errorLength** property is 3 (the length of the token in error).

```
test();
vars
begin
    write "Hello World"
end;
```

errorPosition

Type: Integer

The public **errorPosition** property of the **Script** class contains the offset position within the code of the token found to be in error in most-recent attempt to compile the source code. Note that the first character in the source has a position of zero (**0**).

The value of the **errorPosition** property is zero (**0**) if no attempt to compile the source code is made or the source code has been compiled successfully.

In the following example, a JADE method fails to compile because a semicolon (**;**) is required before the token **end** in the final line. The resulting value of the **errorPosition** property is 44 (the offset position of the token in error).

```
test();
vars
begin
    write "Hello World"
end;
```

status

Type: Integer

The **status** property of the **Script** class contains an integer that indicates the compilation status of the receiver.

A value of zero (**0**) for the **status** property indicates that no attempt to compile the script object has been made since the source was changed, or that an unsuccessful attempt to compile the source code was made.

A value of **1** for the **status** property indicates that the script object has been successfully compiled.

warningCount

Type: Integer

The **warningCount** property of the **Script** class contains the number of warnings generated during the most recent compilation.

Script Methods

The methods defined in the **Script** class are summarized in the following table.

Method	Description
getSource	Returns the source code for the receiver
inError	Returns **true** if the most recent attempt to compile the receiver failed
notCompiled	Returns **true** if the receiver has not been compiled after the most recent source code change

getSource

Signature `getSource(): String;`

The **getSource** method of the **Script** class returns the source code for the receiver.

inError

Signature `inError(): Boolean;`

The **inError** method of the **Script** class returns **true** if the most-recent attempt to compile the receiver failed, and **false** otherwise.

notCompiled

Signature `notCompiled(): Boolean;`

The **notCompiled** method of the **Script** class returns **true** if no attempt to compile the receiver has been made since the source code was changed; otherwise it returns **false**.

Set Class

The **Set** class encapsulates the behavior of a set collection. A set is an unordered collection of objects. An object cannot be referenced in a set more than once. For details about the methods defined in the **Set** class, see "Set Methods", in the following subsection.

The **Set** class also inherits the **getStatistics** method from the **Collection** class, which analyzes the collection and returns structural statistics.

Inherits From: Btree

Inherited By: ObjectSet

Set Methods

The methods defined in the **Set** class are summarized in the following table.

Method	Description
add	Adds a specified object to a set
copy	Copies entries from the receiver to a compatible collection
createIterator	Creates an iterator for the set
getStatistics	Analyzes the collection and returns structural statistics
includes	Returns **true** if the specified object is contained in the set
indexNear	Returns an approximate index of an object in a collection
indexNear64	Returns an approximate index of an object in a collection as an **Integer64** value
remove	Removes a specified object from a set

add

Signature `add(value: MemberType);`

The **add** method of the **Set** class adds the object specified in the **value** parameter to a set; for example:

```
displayHierarchy() updating;
vars
    es           : EmployeeSet;
    emp          : Employee;
    count, level : Integer;
begin
    listOrg.clear;
    create es transient;
    foreach emp in app.myCompany.allEmployees do
        if emp.myManager = null then
            es.add(emp);
        endif;
    endforeach;
    if es.size = 0 then
        listOrg.addItem ("No root for chart");
    else
        foreach emp in es do
```

```
            displayEmployees(emp, 1);
        endforeach;
    endif;
    count := 1;
    while count < listOrg.listCount do
        listOrg.itemExpanded [count] := true;
        count := count + 1;
    endwhile;
end;

foreach emp in app.myCompany.allEmployees do
    if emp.myManager = null then
        empSet.add(emp);
    endif;
endforeach;
```

If the set already contains the object, an exception is raised.

copy

Signature `copy(toColl: Collection input);`

The **copy** method of the **Set** class copies entries from the receiver collection to a compatible collection passed as the **toColl** parameter. In this case, compatible means that the memberships of the receiver and destination collections are type-compatible.

Note Entries copied from the receiver collection are *added* to entries that already exist in the collection to which you copy them.

The following example of the **copy** method returns all regions for the company.

```
getAllRegions(regionSet: RegionSet input);
vars
    country : Country;
begin
    // For each country in the company, copy the country's regions into the set
    foreach country in allCountries do
        country.allRegions.copy(regionSet);
    endforeach;
end;
```

The following example of the **copy** method returns a snapshot of all sales for the company.

```
getAllSales(saleSet: SaleSet input);
begin
    // If the company has some sales, copy them into the supplied set
    if not allSalesByItem.isEmpty then
        allSalesByItem.copy(saleSet);
    endif;
end;
```

createIterator

Signature `createIterator(): Iterator;`

The **createIterator** method creates an iterator for the **Set** class.

Use an iterator associated with the set to remember the current position in the set. For details about iterators, see the **Iterator** class.

getStatistics

Signature `getStatistics(statistics: JadeDynamicObject input);`

The **getStatistics** method of the **Collection** class analyzes the collection and returns structural statistics in the attributes of a **JadeDynamicObject**, representing collection statistics.

The attributes of a collection statistics dynamic object are defined and interpreted as follows.

Attribute	Description
blockSize	Entries per block
keyLength	Size of the key in bytes
entrySize	Size of each **Set** entry in bytes
size	Number of entries that is, the size of the **Set** itself)
blockCount	Total number of blocks in the set
height	Number of levels in the set
minEntries	Minimum number of entries found in any block
maxEntries	Maximum number of entries found in any block
avgEntries	Average number of entries in collection blocks
loadFactor	Actual average percent loading of collection blocks (entries for each block)

To compute the block size in bytes, multiply the **blockSize** attribute by the **entrySize** attribute. The maximum collection block size for a collection is 256K bytes (that is, the value defined by the **MaximumCollectionBlockSize** global constant in the **SystemLimits** category).

The **JadeDynamicObjectNames** category global constants for collection statistics are listed in the following table, where the **name** of the dynamic object represents the collection type of the receiver.

Global Constant	String Value
JStats_ArrayName	"JStatsArray"
JStats_DictionaryName	"JStatsDictionary"
JStats_JadeBytesName	"JStatsJadeBytes"
JStats_SetName	"JStatsName"

The **JadeDynamicObjectTypes** category global constants for collection statistics are listed in the following table, where the **type** of the dynamic object represents the collection type of the receiver.

Global Constant	Integer Value
JStats_ArrayType	101
JStats_DictionaryType	102
JStats_JadeBytesType	104
JStats_SetType	103

The following example shows the use of the **getStatistics** method.

```
vars
    jdo : JadeDynamicObject;
begin
    create jdo;
    node.processes.getStatistics(jdo);
    write jdo.display;
epilog
    delete jdo;
end;
```

For details about the behavior of and tuning collections, see Chapter 4 of your *JADE Developer's Reference*.

includes

Signature `includes(value: MemberType): Boolean;`

The **includes** method of the **Set** class returns **true** if the object specified in the **value** parameter is contained in the set; for example:

```
isEmployee(emp: Employee): Boolean;
// returns true if an emp is below self in the organization hierarchy
vars
    child : Employee;
    bool  : Boolean;
begin
    if myEmployees.size <> 0 then
        if myEmployees.includes(emp) then
            bool := true;
        else
            foreach child in myEmployees do
                if child.isEmployee(emp) = true then
                    bool := true;
                    break;
                endif;
            endforeach;
        endif;
    endif;
    bool := false;
    return bool;
end;

if not goodCustomers.includes(cust) then
    goodCustomers.add(cust);
endif;
```

indexNear

Signature `indexNear(value: MemberType): Integer;`

The **indexNear** method of the **Set** class returns an approximate index for the entry specified in the **value** parameter if it exists in the collection or it returns zero (**0**) if it does not exist. (See also the **Iterator** class **startNearIndex** method.)

If the specified value occurs more than once in the collection, the approximate index of the first occurrence is returned.

Notes This method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf** method.

Use the **indexNear64** method instead of the **indexNear** method, if the number of entries in the set could exceed the maximum integer value of 2,147,483,647.

indexNear64

Signature `indexNear64(value: MemberType): Integer64;`

The **indexNear64** method of the **Set** class returns an approximate index for the entry specified in the **value** parameter if it exists in the collection as an **Integer64** value or it returns zero (**0**) if it does not exist. (See also the **Iterator** class **startNearIndex** method.)

If the specified value occurs more than once in the collection, the approximate index of the first occurrence is returned.

Note This method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf64** method.

remove

Signature `remove(value: MemberType) updating;`

The **remove** method of the **Set** class removes the object specified in the **value** parameter from a set; for example:

```
goodCustomers.remove(cust);
```

If the set does not contain the specified object, an exception is raised.

SetMergeIterator Class

The **SetMergeIterator** class encapsulates the behavior required to sequentially access objects from a merged view of two or more set instances. Set instances need not have the same membership.

When iterating multiple sets, the merged iterator returns objects in a sequence based on their object identifier (oid) value.

To iterate a single collection, the iterator is created and associated with the collection by using the **createIterator** method on the collection object. To iterate a merged view of more than one collection, first create the iterator and then use the **addCollection** method for each set to be attached to the iterator, as shown in the following example.

```
vars
    iter : SetMergeIterator;
    set1, set2 : CustomerSet;
    cust : Customer;
begin
    // Assign set1 and set2
    create iter transient;
    iter.addCollection(set1);
    iter.addCollection(set2);
    while iter.next(cust) do
```

```
            write cust.name;
        endwhile;
    end;
```

For details about the property and methods defined in the **SetMergeIterator** class, see "SetMergeIterator Property" and "SetMergeIterator Methods", in the following subsections.

Inherits From: Iterator

Inherited By: (None)

SetMergeIterator Property

The property defined in the **SetMergeIterator** class is summarized in the following table.

Property	Description
ignoreDuplicates	Skips duplicate entries in the merged iterator view

ignoreDuplicates

Type: Boolean

Default Value: True

The **ignoreDuplicates** property of the **SetMergeIterator** class specifies whether duplicate entries in the merged view should be skipped when iterating using the **next** and **back** methods.

Duplicate entries can occur in the merged view when an object is included in more than one of the attached sets.

SetMergeIterator Methods

The methods defined in the **SetMergeIterator** class are summarized in the following table.

Method	Description
addCollection	Adds the specified set to the merged iterator view
back	Accesses entries in reverse order in the merged iterator view
current	Returns the last value iterated by the **back** or **next** method
getCollectionAt	Returns the set at the specified index in the collection of sets making up the merged iterator view
getCollectionCount	Returns the number of sets
getCurrentCollection	Returns the set containing the last value iterated by the **back** or **next** method
isValid	Returns **true** if the receiver is a valid iterator
next	Accesses successive entries in the merged iterator view
removeCollection	Removes the specified set from the merged iterator view
reset	Initializes the iterator
startAtObject	Sets the starting position of the iterator at the position of the specified object

addCollection

Signature `addCollection(set: Set);`

The **addCollection** method of the **SetMergeIterator** class adds the set specified by the value of the **set** parameter to the merged iterator view.

The parameter value must be a **Set** type. The **Set** instance being added does not need to have the same membership type as existing sets associated with the iterator.

An exception is raised if you attempt to add a set that is already attached to the iterator and therefore part of the merged iterator view.

back

Signature `back(value: Any output): Boolean;`

The **back** method of the **SetMergeIterator** class accesses entries in reverse order one at a time in the sets comprising the merged iterator view.

This method returns **true** when a prior entry is found, and the entry is assigned to the **value** parameter. It returns **false** when a prior entry is not found because the iterator is positioned before the first entry in the merged view, and the **value** parameter becomes a **null** reference.

When the **back** method is used with an iterator where that iterator has been passed to a method as a method parameter, the iterator must be defined as a usage input; that is, the iterator cannot be modified by the called method.

The following example shows the use of the **back** method.

```
getReversedPosition(pObj: Object; pIter: SetMergeIterator input): Integer;
vars
    pos : Integer;
    obj : Object;
begin
    while pIter.back(obj) do
        pos := pos - 1;
        if obj = pObj then
            return pos;
        endif;
    endwhile;
    return 0;
end;
```

current

Signature `current(value: Any output): Boolean;`

The **current** method of the **SetMergeIterator** class returns the last value iterated by using the **back** or **next** method.

This method returns **true** if the iterator is positioned on an entry in the merged view, or it returns **false** if the iterator is reset or it is positioned beyond the start or end of the merged view. The **value** parameter receives the entry of the current iterator position in the merged view.

getCollectionAt

Signature `getCollectionAt(index: Integer): Set;`

The **getCollectionAt** method of the **SetMergeIterator** class returns the set at the index position specified by the **index** parameter in the array of collections attached to the iterator.

getCollectionCount

Signature `getCollectionCount(): Integer;`

The **getCollectionCount** method of the **SetMergeIterator** class returns the number of sets that have been attached to the iterator by using the **addCollection** method.

getCurrentCollection

Signature `getCurrentCollection(): Set;`

The **getCurrentCollection** method of the **SetMergeIterator** class returns the set containing the last value iterated by using the **back** or **next** method.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **SetMergeIterator** class returns **true** if the receiver is a valid iterator for all of the sets in the merged view.

next

Signature `next(value: Any output): Boolean;`

The **next** method of the **SetMergeIterator** class accesses successive entries in the sets comprising the merged iterator view.

This method returns **true** when a next entry is found, and the entry is assigned to the **value** parameter. It returns **false** when a next entry is not found because the iterator is positioned after the last entry in the merged view, and the **value** parameter becomes a **null** reference.

When the **next** method is used with an iterator where that iterator has been passed to a method as a method parameter, the iterator must be defined as a usage input; that is, the iterator cannot be modified by the called method.

The following example shows the use of the **next** method.

```
getPosition(pObj: Object; pIter: SetMergeIterator input): Integer;
vars
    pos : Integer;
    obj : Object;
begin
    while pIter.next(obj) do
        pos := pos + 1;
        if obj = pObj then
            return pos;
        endif;
    endwhile;
```

```
    return 0;
end;
```

removeCollection

Signature `removeCollection(set: Set)`

The **removeCollection** method of the **SetMergeIterator** class removes the set specified by the **set** parameter from the array of sets associated with the iterator.

An exception is raised if you attempt to remove a set that is not attached to the iterator and therefore is not part of the merged iterator view.

reset

Signature `reset() updating;`

The **reset** method of the **SetMergeIterator** class restarts an iteration. After executing this method, the following **next** method would start from the first entry in the merged view; that is, it would apply to all sets. Similarly, the **back** method would start from the last entry in the merged view.

startAtObject

Signature `startAtObject(object: Object) updating;`

The **startAtObject** method of the **SetMergeIterator** class sets the starting position of the iterator for the merged view at the position of the object specified in the **object** parameter.

An exception is raised if this object is not compatible with the membership of the collection being iterated.

SortActor Class

The **SortActor** class contains properties that enable you to specify the precedence of records in the **File** class. The following example shows the use of sort actors.

```
buttonSortFile1_click(btn: Button input) updating;
vars
    sortActor           : SortActorArray;
    sort1, sort2, sort3 : SortActor;
begin
    // Creates transient instances of SortActor and SortActorArray classes.
    create sortActor transient;
    create sort1 transient;
    create sort2 transient;
    create sort3 transient;
    // Sets the recordSize property of the file to 0, indicating the sorted
    // file has variable records.  The records will be delimited by the
    // standard carriage return and line feed endOfLine characters.
    self.file1.recordSize := 0;
    self.file1.endOfLine  := #"0D" & #"0A";
    // Sets the endOfField property to a comma, indicating the file has
    // variable fields within each record that will be delimited by a comma.
    self.file1.endOfField := ",";
    // Sets the first SortActor instance to sort using the string from the
    // 13th character of the first field through to the end of the field.
    // The sort will be done in ascending order.
    sort1.fieldNo       := 1;
    sort1.startPosition := 13;
    sort1.ascending     := true;
    // Sets the second SortActor instance to sort the duplicates from the
    // first sort using the string from the 14th character of the second
    // field through to the end of the field.  The sort will be numeric and
    // will be done in descending order.
    sort2.fieldNo       := 2;
    sort2.startPosition := 14;
    sort2.ascending     := false;
    sort2.sortType      := SortActor.SortType_Integer;
    // Sets the third SortActor instance to sort the duplicates from the
    // first and second sorts using the string from the 16th character of
    // the third field through to the end of the field.  The sort will be
    // done in ascending order.
    sort3.fieldNo       := 3;
    sort3.startPosition := 16;
    sort3.ascending     := true;
    // Adds SortActor instances to the SortActorArray for use in the sort.
    sortActor.add(sort1);
    sortActor.add(sort2);
    sortActor.add(sort3);
    // Uses the File class extractSort method to sort the file and write
    // the output to the text box.
    self.file1.extractSort(sortActor, outputFile1);
    textBox3.text := outputFile1.readString(400);
    self.resetOutputFile1;
    self.file1.close;
```

```
epilog                  // Delete the transient instances.
    delete sortActor;
    delete sort1;
    delete sort2;
    delete sort3;
end;
```

The **SortActorArray** class contains the sort actors used to sort an external file.

For details about the class constants and properties defined in the **SortActor** class, see "SortActor Class Constants" and "SortActor Properties", in the following subsection. For more sort actor examples, see "Example of Sorted Fixed Fields or Records" and "Example of Sorted Variable Fields or Records" under the **File** class **extractSort** method, earlier in this chapter.

Inherits From: Object

Inherited By: (None)

SortActor Class Constants

The constants provided by the **SortActor** class for use in the **sortType** property are listed in the following table.

Constant	Integer Value	Constant	Integer Value
SortType_Binary	3	SortType_Decimal	2
SortType_Integer	1	SortType_String	0

SortActor Properties

The properties defined in the **SortActor** class are summarized in the following table.

Property	Description
ascending	Specifies whether records or fields in a file are sorted in ascending order
fieldNo	Contains the field number in a file that is to be compared
length	Contains the length from the offset of the field or record in a file that is to be compared
numeric	Specifies whether records or fields in a file are to be compared numerically
random	Specifies whether records or fields in a file are to be sorted according to a random order
sortType	Specifies the sort type of the records or fields in a file
startPosition	Contains the current position in the record or field in a file for comparison

ascending

Type: Boolean

The **ascending** property of the **SortActor** class specifies whether the records or fields in a file are sorted in ascending order.

If the value of this property is set to **false**, a descending sort is performed.

The default values for the **ascending** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
true	true	true	true

fieldNo

Type: Integer

The **fieldNo** property of the **SortActor** class contains the field number in a file that is to be compared.

The default values for the **fieldNo** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
1	1	1	1

The code fragment in the following example sets the **SortActor** instance to sort the file randomly.

```
sort1.fieldNo := 1;
sort1.random  := true;
```

length

Type: Integer

The **length** property of the **SortActor** class contains the length from the start position of the field or record in a file that is to be compared.

Note In fixed-length files, any carriage return (**CR**) or line feed (**LF**) character is included in the length of the field or record. When sorting fixed-length records, the entire length of the file must be divisible by the record size (that is, the value contained in the **File** class **recordSize** property), with no remainder.

The default values for the **length** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
To end of field	To end of record	To end of field	To end of record

When sorting fixed-length records, the entire length of the file must be divisible by the value contained in the **File** class **recordSize** property, with no remainder.

numeric

Type: Boolean

The **numeric** property of the **SortActor** class specifies whether the records or fields in a file are to be compared numerically.

Note From JADE release 6.0, the **numeric** property has been replaced by the **sortType** property. If you set the **numeric** property to **true**, the **sortType** property is set to **SortType_Integer** (**1**).

If the value of this property is set to **false** (the default), records or fields are compared alphanumerically.

The default values for the **numeric** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
false	false	false	false

random

Type: Boolean

The **random** property of the **SortActor** class specifies whether the records or fields in a file are to be sorted in a random order.

Tip Use this property, for example, to randomly sort a file for testing purposes.

The default values for the **random** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
false	false	false	false

The code fragment in the following example shows the use of the **random** property to set the **SortActor** instance to sort the file randomly.

```
sort1.fieldNo       := 1;
sort1.startPosition := 1;
sort1.length        := 4;
sort1.random        := true;
```

sortType

Type: Integer

The **sortType** property of the **SortActor** class specifies the sort type of the records or fields in a file.

The **SortActor** class constants for this property are summarized in the following table.

Constant	Value	Description
SortType_Binary	3	Sort based on bit values.
SortType_Decimal	2	Sort as a JADE decimal (up to 23 characters of optional sign, decimal point, and digits).
SortType_Integer	1	Sort as an integer. (Equivalent to **numeric** = **true**.)
SortType_String	0 (default)	Sort as a string, based on locale. (Equivalent to **numeric** = **false**.)

If the value of this property is set to the default **SortType_String** (**0**) value, records or fields are compared alphanumerically.

Note From JADE release 6.0, the **sortType** property replaces the **numeric** property. If you set the **numeric** property to **true**, the **sortType** property is set to **SortType_Integer** (**1**).

The default values for the **sortType** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
SortType_String	SortType_String	SortType_String	SortType_String

startPosition

Type: Integer

The **startPosition** property of the **SortActor** class contains the start position from the beginning of the record or field in a file that is to be compared.

Note In fixed-length files, any carriage return (**CR**) or line feed (**LF**) character is included in the length of the field or record.

The default values for the **startPosition** property are listed in the following table.

Variable Records, Variable Fields	Variable Records, Fixed Fields	Fixed Records, Variable Fields	Fixed Records, Fixed Fields
1	1	1	1

SortActorArray Class

The **SortActorArray** class contains **SortActor** objects and has properties that enable you to specify the precedence of records in an external file.

For details about the properties defined in the **SortActorArray** class, see "SortActorArray Properties", in the following subsection. For sort actor array examples, see "Example of Sorted Fixed Fields or Records" and "Example of Sorted Variable Fields or Records" under the **File** class **extractSort** method, earlier in this chapter.

Inherits From: ObjectArray

Inherited By: (None)

SortActorArray Properties

The properties defined in the **SortActorArray** class are summarized in the following table.

Property	Contains the …
kway	Maximum number of sort files that are merged in a single pass
lcid	Locale over which non-numeric fields and records are sorted
maxMem	Maximum percentage of physical memory that the sorting method can use to sort the external file

kway

Type: Integer

The **kway** property of the **SortActorArray** class contains the maximum number of sort files that are merged in a single pass.

lcid

Type: Integer

The **lcid** property of the **SortActorArray** class contains the locale over which non-numeric fields and records are sorted.

The default value of **768** specifies an invalid locale identifier. If the default **lcid** value is passed to the **extractSort** method of the **File** class, the default system locale is used for sorting.

maxMem

Type: Integer

The **maxMem** property of the **SortActorArray** class contains the maximum percentage of physical memory that the sorting method can use to sort the external file.

The specified value represents a percentage and has a valid range of **1** through **50**. The default value of **10** is used if an invalid value is specified.

Sound Class

The **Sound** class is the abstract subclass of the **MultiMediaType** class that contains the properties and methods for the sound multimedia type. The internal speaker or sound card is initiated when the **Sound** object invokes the **play** method.

As you cannot create an instance of an abstract class, define your own **Sound** subclass with the appropriate user data file mapping and then create the instance of your new subclass that you require. You can create persistent subclass instances.

Notes Sound files are recorded and edited by using the Windows sound recorder. This standard Windows utility is normally installed in the **Accessories** program folder. You can also purchase libraries of sound files from third-party vendors.

Sound files can be played on a workstation only if a compatible sound card is installed.

For details about the properties and methods defined in the **Sound** class, see "Sound Properties" and "Sound Methods", in the following subsections.

Inherits From: MultiMediaType

Inherited By: (None)

Sound Properties

The properties defined in the **Sound** class are summarized in the following table.

Property	Description
data	Contains the sound binary data
format	Contains the format of the sound data
name	Contains a string of the sound name

data

Type: Binary

Availability: Public

The **data** property of the **Sound** class contains the sound binary data.

format

Type: Binary

Availability: Public

The **format** property of the **Sound** class contains the format of the sound data.

name

Type: String[29]

Availability: Public

The **name** property of the **Sound** class contains the sound name.

Sound Methods

The methods defined in the **Sound** class are summarized in the following table.

Method	Description
isPlayable	Specifies if a sound device is capable of playing the sound stream in the object
loadFromFile	Loads the specified .**wav** file
play	Plays the .**wav** file associated with the receiver **Sound** object

isPlayable

Signature `isPlayable(): Boolean;`

The **isPlayable** method of the **Sound** class returns **true** if a sound device capable of playing the sound stream contained in the receiver is configured and available. In JADE thin client mode, the **isPlayable** method always executes on the presentation client. An exception is raised if this method is invoked from a server method.

The device must have the capabilities required to play back the sound stream specified in the **format** property.

loadFromFile

Signature `loadFromFile(fileName: String);`

The **loadFromFile** method of the **Sound** class loads the .**wav** file specified in the **fileName** parameter string and updates the format and binary data for the **Sound** object. In JADE thin client mode, this method by default attempts to load the specified file from the presentation client.

The code fragment in the following example shows the use of the **loadFromFile** method.

```
beginTransaction;
    create sound;
    sound.loadFromFile("c:\jade\pics\heat.wav");
    sound.play;
commitTransaction;
```

An exception is raised if this method is invoked from a server method. (For details about the processing of this method when the application is running in JADE thin client mode, see the **MultiMediaType** class **usePresentationFileSystem** property, earlier in this chapter.)

play

Signature `play();`

The **play** method of the **Sound** class plays the sound stream contained in the receiver **Sound** object. In JADE thin client mode, this method always executes on the presentation client.

This method transfers the sound wave image from the receiver object to memory and creates a thread to play the sound asynchronously, allowing control to be returned to the application as soon as the sound has started playing. If the sound cannot be played, a message is logged in your JADE log file.

An exception is raised if this method is invoked from a server method.

See also the **Application** class **playSound** method, which plays the specified .**wav** file and returns when the sound file has been played, or the **playSoundAsync** method, which starts playing the specified .**wav** file and returns immediately.

StringArray Class

The **StringArray** class is an ordered collection of **String** values with a length less than or equal to 62 characters. However, you can subclass the **StringArray** class and specify a different length for the strings in the array.

Note The **StringArray** class is unsuitable for storing JADE identifiers (for example, the names of schemas, classes, forms, and methods), which have lengths up to 100 characters. The **JadeIdentifierArray** class can be used for this purpose.

The values are referenced by their position in the collection.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a **String** array.

Inherits From: Array

Inherited By: JadeIdentifierArray

StringUtf8Array Class

The **StringUtf8Array** class is an ordered collection of **StringUtf8** values with a length less than or equal to 30 UTF8 characters. However, you can subclass the **StringUtf8Array** class and specify a different length for the UTF8 strings in the array.

The values are referenced by their position in the collection.

StringUtf8 arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a **StringUtf8** array.

Inherits From: Array

Inherited By: (None)

System Class

The **System** class defines the behavior of the JADE system. There is one instance only of the **System** class for each JADE environment (installation).

For details about the properties and methods defined in the **System** class, see "System Properties" and "System Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

System Properties

The properties defined in the **System** class are summarized in the following table.

Property	Contains…
name	The name of the system that is assigned internally by JADE
nodes	All of the nodes currently attached to the system

name

Type: String[100]

The read-only **name** property of the **System** class contains an internally assigned name for the executing system.

nodes

Type: NodeDict

The read-only **nodes** property of the **System** class contains a reference to all of the nodes currently attached to the system.

Caution Lock environmental object collections with extreme caution, as this can cause hold-ups when processes sign off and on and when nodes initiate and terminate; for example, you should *never* use the **foreach** instruction to iterate through an environmental object collection. Instead, create a transient clone of the collection to iterate through.

The code fragment in the following example shows the use of the **nodes** property.

```
while size = system.nodes.size do
    process.sleep(2000);
endwhile;
```

System Methods

The methods defined in the **System** class are summarized in the following table.

Method	Description
activateDeltaDatabase	Activate or deactivate delta database mode

Method	Description
beginIndividualRequestsLogging	Starts sampling the individual remote requests of all processes in the node
beginLockContentionStats	Starts recording lock contentions for persistent objects
beginObjectTracking	Starts recording in a file all reads and writes of persistent objects to the database
beginSample	Opens a new sampling context for each of the nodes in the sample definition group and begins the accumulation of sampling statistics for those nodes
beginSampleGroupDefinition	Opens a new remote sampling context for a group of nodes
clearLockContentionStats	Removes all existing lock contention data and restarts recording of lock contentions
createSystemSequenceNumber	Initializes a named *system sequence number* to a specified value
disableRemoteSampling	Disables sampling of statistics on the specified node
dumpCharacterEntityTable	Lists the supported character entity names and values
enableRemoteSampling	Enables sampling of statistics on the specified node
endIndividualRequestsLogging	Terminates the sampling of individual requests of all processes in enabled nodes
endLockContentionStats	Stops recording lock contentions and removes all lock contention data
endObjectTracking	Ends recording in a file all reads and writes of persistent objects to the database
endSample	Terminates sampling of statistics on all currently enabled nodes and releases the file
endSampleGroupDefinition	Terminates the sampling context identified in the **samplingHandle** parameter for the group of nodes
findCharacterEntityByName	Locates the Unicode code point for a character entity
findCharacterEntityByNumber	Locates the character entity for a Unicode code point
forceOffUser	Forces (signs) a user off a process
getAllUsers	Returns an array of all users in all nodes in the system
getClassAccessFrequencies	Returns access counts for specified classes
getDatabaseRole	Returns the database role of the server node on which the JADE system is running
getDatabaseStats	Returns statistics relating to persistent database activity
getDatabaseSubrole	Returns the subrole of the server node on which the JADE system is running
getDbDiskCacheStats	Returns statistics relating to the persistent database disk cache
getDeltaDatabaseStatus	Returns the delta database status
getEnvironmentServerIdentity	Returns the environment and server identities
getLockContentionInfo	Retrieves lock contention information for the specified object

Method	Description
getLockContentionStats	Retrieves lock contention information
getLocks	Fills an array with a specified number of instances of current locks in the system
getMostAccessedClasses	Returns access counts for the classes that have been most frequently accessed
getNotes	Fills an array with a specified number of instances of current notification requests in the system
getObjectLockProcesses	Populates the **processes** parameter with all processes that have locks on the specified object
getObjectPartitionID	Returns the identifier of the partition in which the specified object is located
getQueuedLocks	Fills an array with the specified number of instances of lock requests in the system that are waiting for a locked object
getRequestStats	Returns system statistics relating to requests carried out by the database server node
getRpcServerStatistics	Retrieves RPC statistics relating to activity between the database server node and all client nodes
getStatistics	Loads the specified **Integer** values with system statistics
getStatistics64	Loads the specified **Integer64** values with system statistics
getSystemSequenceNumberNext	Increments a specified *system sequence number* and returns the new value
getTimeInTransactionState	Returns the number of milliseconds a process is in transaction state
interruptUser	Causes a conditional interruption of the specified process
isDatabaseEncryptionEnabled	Specifies whether database encryption is enabled
isDbArchival	Specifies whether database archival recovery is enabled for the JADE system
isRemoteSamplingEnabled	Specifies whether remote statistics sampling is enabled for the specified node
isValidProcess	Returns **true** if the process represents a signed-on application
logObjectCaches	Specifies the object cache statistics for nodes enabled for sampling
logRequestStatistics	Specifies the request statistics that are logged for all processes in enabled nodes
logUserCommand	Invokes the **NodeSampleUserCommandCallBack** entry point in the user library
processDumpAllNodes	Invokes process dumps of the database server node and all nodes attached to it
queryLockContentionStats	Retrieves information about the current recording of lock contentions
removeNode	Forces (signs) off all users on the node from the system
sdsAuditEnableSecondaryApps	When invoked on an SDS primary, restarts applications and enables sign-on on an SDS secondary

Method	Description
verifyDbEncryptionMasterKey	Specifies whether the database encryption master key is present and correct

activateDeltaDatabase

Signature `activateDeltaDatabase(activate: Boolean; timeout: Integer): Boolean;`

The **activateDeltaDatabase** method of the **System** class is used to perform one of the following actions.

- Create a delta database and activate delta mode.
- Deactivate delta mode and delete the delta database.

Use the **activate** parameter to specify whether the delta database is being activated (**true**) or deactivated (**false**).

Note The **DeltaDatabaseCapable** parameter in the [JadeServer] section of the JADE initialization file on the database server node must be set to **true** for an activation request to succeed.

Use the **timeout** parameter to specify a maximum number of seconds to allow for deactivation to take place. A default timeout of 60 seconds is used if the specified value is zero (**0**). The **timeout** parameter is ignored when the **activate** parameter is set to **true**.

You cannot deactivate the delta database until all current processes apart from JADE tools (for example, the JADE Monitor and the SDS Administration application) and the process making the request are idle. The deactivation attempt is abandoned if this does not happen within the time specified in the **timeout** parameter, the delta database remains activated, and system exception 1163 is raised.

The return value indicates the delta database status prior to the method call. If the delta database is active at the time of the call, **true** is returned, or if it was inactive, **false** is returned.

The following code fragment is an example of activating a delta database.

```
system.activateDeltaDatabase(true, 0);
```

The following code fragment is an example of deactivating a delta database.

```
//Deactivate the Delta Database. Allow up to 2 minutes for deactivation.
system.activateDeltaDatabase(false, 120);
```

The following system exceptions can be raised from an **activateDeltaDatabase** method call.

- 1162 - *The system is not delta database capable*

  This exception is raised if the database server node is not specified to be delta database-capable.

- 1163 - *Could not change delta database mode because not all processes are idle*

  This exception is raised and the deactivation attempt is abandoned if a deactivation request cannot be completed because not all applications have become idle within the time specified in the **timeout** parameter.

- 1165 - *A delta database transition request is already in progress*

  This exception is raised if a deactivation or activation request is currently in progress.

beginIndividualRequestsLogging

Signature

```
beginIndividualRequestsLogging(samplingHandle:              Integer;
                               localRequests:               Boolean;
                               remoteRequests:              Boolean;
                               persistentCacheBuffers:      Boolean;
                               transientCacheBuffers:       Boolean;
                               remoteTransientCacheBuffers: Boolean;
                               userNumber:                  Integer;
                               userText:                    String);
```

The **beginIndividualRequestsLogging** method of the **System** class starts sampling individual requests or cache activities, or both, of all processes in each of the remote nodes in the sample definition group and invokes the **NodeSampleIndividualRequestCallBack** or the **NodeSampleObjectBufferCallBack** entry point, or both of these entry points, in the user library specified in the **libraryName** parameter of the **beginSample** method.

The **NodeSampleIntervalCallBack** entry point is invoked once only before these entry points, with the **eventType** parameter in the entry point set to **1**.

The **beginIndividualRequestsLogging** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSampleGroupDefinition** method when sampling started
localRequests	Logs individual requests to the database of the node
remoteRequests	Logs individual requests to remote nodes
persistentCacheBuffers	Logs activities in the persistent object cache
transientCacheBuffers	Logs activities in the transient object cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the sampling of the statistics that you require, set the appropriate Boolean parameters to **true**.

The following code fragment shows an example of the **beginIndividualRequestsLogging** method and its parameters.

```
system.beginIndividualRequestsLogging(samplingHandle, false, true, false,
          false, false, 4, "Start logging of remote requests");
```

The JADE sampling libraries produce the following record types.

- Begin process record (type **6**)
- **BeginInterval** record (type **11**), containing your specified user number and text to the output file immediately, followed by one **IndividualRequest** record for each of the subsequent remote requests or one cache buffer activity record for each of the subsequent buffer cache activities, or both
- Individual local request records (record type **14**)
- Individual remote request records (record type **10**)
- Cache buffer activity records (record type **2**)

For more details, see Chapter 4 of the *JADE Object Manager Guide*.

beginLockContentionStats

Signature `beginLockContentionStats(tableSize: Integer);`

The **beginLockContentionStats** method of the **System** class starts recording lock contentions for persistent objects. A lock contention occurs when an attempt to lock a persistent object is queued because the object is already locked.

After recording has been initiated, you can retrieve lock contention information using the **getLockContentionStats** method of the **System** class. The **endLockContentionStats** method of the **System** class is used to stop the recording of lock contentions.

The value of the **tableSize** parameter determines the maximum number of individual contended objects that can be recorded. When the first contention for an object is noted, the object is added to the table of contentions, provided the maximum table size has not been reached. If the table has reached the maximum size, contentions for objects not found in the table are grouped together in a single entry identified by a null object identifier; that is, the class number and instance number are both set to zero (**0**).

As a guideline, the size of each entry is approximately 40 bytes, so 25,000 entries would consume approximately 1M byte of memory.

Only one process at a time can control the recording of lock contentions. If a process executes the **beginLockContentionStats** method when recording of lock contentions has already been initiated by another process, an 1131 exception (*Another process is currently in control of lock contention statistics*) is raised. However, processes other than the one that initiated lock contention recording can retrieve lock contention information, but you should be aware that the information may be cleared or become unavailable at any time.

If the process that started to record lock contentions terminates without having called the **endLockContentionStats** method to stop lock contention recording, the lock contention recording is automatically ended.

beginObjectTracking

Signature `beginObjectTracking(fileName: String);`

The **beginObjectTracking** method of the **System** class starts recording, in a file (on the database server node) specified by the **fileName** parameter, every persistent object read operation from the database or write operation to the database. The information recorded distinguishes between read operations to get an object and read operations to lock an object. For write operations, it also distinguishes between write operations to create an object, write operations to delete an object, and write operations to update an object.

Note This does not necessarily record every time an application uses an object, because if the object resides in the persistent object cache, it may not have to be fetched from the database.

Object tracking is terminated by using the **endObjectTracking** method defined in the **System** class.

The file is a standard text file. The first line is a header record and the last line is a trailer record. The lines in between represent individual object accesses. In each line, fields are separated by spaces.

The format of the three types of records is similar to that used for node sampling files. The format of the header record is shown in the following table.

For details, see "System::*beginObjectTracking* Method", in Chapter 4 of the *JADE Object Manager Guide*.

Only one object tracking session can be active at a time. If the **beginObjectTracking** method is called when object tracking is already active, an 1138 exception (*Object tracking is already active*) is raised.

Caution Use this method with caution. When object tracking is active, the tracking file can fill very rapidly. Object tracking should therefore be used in relatively short bursts.

beginSample

Signature
```
beginSample(samplingHandle:          Integer;
            libraryName:             String;
            initializationParameter: String);
```

The **beginSample** method of the **System** class opens a new sampling context for each of the nodes in the sample definition group, begins the accumulation of sampling statistics for those nodes, and invokes the following entry points.

- **NodeSampleInfoCallBack**, passing it the **initializationParameter** string and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleProcessInfoCallBack**, invoked every time a process begins and once for every existing process at the time sampling begins.

The **samplingHandle** parameter identifies the context that is sampled. (This is the identifier of the sampling context, returned by the **beginSampleGroupDefinition** method when sampling started.)

When this method is called in your application, request statistics are stored in transient memory for every process in the nodes in the group until they are passed to the corresponding entry point in the user library specified in the **libraryName** parameter.

If you are using the **filesmpl** or **tcpsmpl** JADE sampling library, you can set the **initializationParameter** parameter to **"<null>"** or to **""** so that sample values will not be output. For **filesmpl**, the values will not be written to a file. For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted. For more details, see "Direct Node Sampling", in Chapter 4 of the *JADE Object Manager Guide*.

The JADE-supplied library writes a begin process record (type **6**) to the statistics file.

beginSampleGroupDefinition

Signature
```
beginSampleGroupDefinition(): Integer;
```

The **beginSampleGroupDefinition** method of the **System** class opens a new remote sampling context for a group of nodes. The nodes are included in the sampling context by using the **System** class **enableRemoteSampling** method.

The **beginSampleGroupDefinition** method returns the sampling handle number used to identify the sampling context that is opened. All subsequent methods use this sampling context handle as the first parameter, and they are initially executed by the server node and sent to each of the nodes in the definition group by means of internal notifications.

Any error condition at the individual node level is written to the JADE Object Manager message log file for that node.

The following example shows the use of the **beginSampleGroupDefinition** method.

```
testManualSamplingFullInterval();
vars
    cust           : Customer;
    samplingHandle : Integer;
begin
    samplingHandle := system.beginSampleGroupDefinition;
    system.beginSample("filesmpl", "c:\temp\fullInterval%p, txt");
    system.logObjectCaches(samplingHandle, true, true, false, false,
                           false, false, 111, "cachesSampling");
    system.beginIndividualRequestsLogging(samplingHandle, false, true, true,
                                          true, false, 557, "fullInterval");
    foreach cust in Customer.instances do
        write cust.name;
    endforeach;
    system.endIndividualRequestsLogging(samplingHandle, 557,
                                        "fullInterval");
    system.endSample(samplingHandle);
    system.endSampleGroupDefinition(samplingHandle);
end;
```

For more details, see Chapter 4 of the *JADE Object Manager Guide*.

clearLockContentionStats

Signature `clearLockContentionStats();`

The **clearLockContentionStats** method of the **System** class removes all existing lock contention data and restarts recording of lock contentions. A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked.

The lock contention table is cleared, but retains the same maximum size. Use this method for recording lock contention activity over set periods without having to end and begin lock contention recording multiple times.

Only the process that started lock contention recording can use this method. If any other process attempts to use this method, an exception is raised.

If lock contention recording is not active when this method is called, it has no effect.

createSystemSequenceNumber

Signature
```
createSystemSequenceNumber(name:         String;
                           initialValue: Integer64);
```

The **createSystemSequenceNumber** method of the **System** class creates a system sequence number with the name specified by the value of the **name** parameter and a current value specified by the **initialValue** parameter. If a system sequence number with the specified name already exists, the current value is not changed and no error is reported. All access to the system sequence number table is single-threaded and is independent of process transaction state.

If the value of the **name** parameter is **null**, it contains embedded null characters, or is longer than 60 characters, a 1454 (*The SystemSequenceNumber name is invalid*) exception is raised.

If the value of the **initialValue** parameter is less than zero (**0**), a 1455 (*The SystemSequenceNumber initial value cannot be negative*) exception is raised. There is no restriction on the number of system sequence numbers you can create.

Ensure that the initial value passed to the **createSystemSequenceNumber** method does not cause the **getSystemSequenceNumberNext** method to return an already used number. If the **getSystemSequenceNumberNext** method returns zero (**0**), determine the highest number that has been assigned to an object stored in the database and call the **createSystemSequenceNumber** method passing that value.

The following method returns the next available customer number, where the customer number is used as a key in an exclusive **MemberKeyDictionary** collection owned by a **Company** object.

```
getNextCustomerNumber(): Integer64;
constants
    SSN_CustomerNumber: String = "MySchema::CustomerNumber";
vars
    nextNumber: Integer64;
    coy: Company;
    cust: Customer;
begin
    nextNumber := system.getSystemSequenceNumberNext(SSN_CustomerNumber);
    if nextNumber = 0 then
       coy := Company.firstInstance();
       if coy <> null then
            cust := coy.allCustomersByNumber.last();
           if cust <> null then
               nextNumber := cust.number
           endif;
       endif;
       system.createSystemSequenceNumber(SSN_CustomerNumber, nextNumber);
       nextNumber := system.getSystemSequenceNumberNext(SSN_CustomerNumber);
    endif;
    return nextNumber;
end;
```

disableRemoteSampling

Signature `disableRemoteSampling(samplingHandle: Integer; n: Node);`

The **disableRemoteSampling** method of the **System** class disables the sampling of statistics on the node specified in the **n** parameter.

This method takes the specified node out of the sample definition group of the context identified by the **samplingHandle** parameter. (The sampling handle is the identifier of the sampling context, returned by the **beginSampleGroupDefinition** method when sampling started.)

The sampling for the context identified in the **samplingHandle** parameter is started by using the **beginSample** method. For more details, see Chapter 4 of the *JADE Object Manager Guide*.

dumpCharacterEntityTable

Signature `dumpCharacterEntityTable(): String;`

The **dumpCharacterEntityTable** method of the **System** class returns a string that displays a table showing the name, code point, and description of each supported character entity. Part of the output is shown in the following lines.

```
nbsp    160        no-break space = non-breaking space
iexcl    161        inverted exclamation mark
cent    162        cent sign
...
```

For details about the use of character entities in UTF8 strings, see "StringUtf8 Type", in Chapter 1 of the *JADE Encyclopaedia of Primitive Types*.

enableRemoteSampling

Signature
```
enableRemoteSampling(samplingHandle: Integer;
                     n:              Node);
```

The **enableRemoteSampling** method of the **System** class enables the sampling of statistics on the node specified in the **n** parameter.

This method includes the node in the sampling group of the context identified in the **samplingHandle** parameter. (The sampling handle is the identifier of the sampling context, returned by the **beginSampleGroupDefinition** method when sampling started.)

Use the **System** class **disableRemoteSampling** method to disable the sampling of statistics on the specified node (that is, take the specified node out of the sample definition group of the context identified in the **samplingHandle** parameter).

For more details, see Chapter 4 of the *JADE Object Manager Guide*.

endIndividualRequestsLogging

Signature
```
endIndividualRequestsLogging(samplingHandle: Integer;
                             userNumber:     Integer;
                             userText:       String);
```

The **endIndividualRequestsLogging** method of the **System** class terminates the sampling of individual requests or cache activities started by the **beginIndividualRequestsLogging** method of the **System** class for each of the nodes in the sample definition group and invokes the **NodeSamplIntervalCallBack** entry point with the **eventType** parameter of the entry point set to **2**.

The **endIndividualRequestsLogging** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSampleGroupDefinition** method when sampling started
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The following code fragment shows an example of the **endIndividualRequestsLogging** method and its parameters.

```
system.endIndividualRequestsLogging(samplingHandle, 4, "End logging of
                        remote requests");
```

The JADE-supplied library writes an **endInterval** record (type **12**) containing your specified user number and text to the output file. For more details, see Chapter 4 of the *JADE Object Manager Guide*.

endLockContentionStats

Signature `endLockContentionStats();`

The **endLockContentionStats** method of the **System** class stops recording lock contentions for persistent objects and removes all lock contention data.

Only the process that started lock contention recording can use this method. If any other process attempts to use this method, an 1131 exception (*Another process is currently in control of lock contention statistics*) is raised.

If the process that started the recording of lock contentions terminates without having called **endLockContentionStats** to stop lock contention recording, the lock contention recording is automatically ended.

endObjectTracking

Signature `endObjectTracking();`

The **endObjectTracking** method of the **System** class ends recording persistent object read operations from the database or write operations to the database.

An object tracking session is started using the **beginObjectTracking** method defined in the **System** class. If the **endObjectTracking** method is called when object tracking is not active, an 1139 exception (*Object tracking is not active*) is raised.

endSample

Signature `endSample(samplingHandle: Integer);`

The **endSample** method of the **System** class terminates the sampling of statistics on each of the nodes in the sample definition group for the context identified by the **samplingHandle** parameter and invokes the following entry points.

- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **2**.
- **NodeSampleInfoCallBack**, which your user library should consider the last call for the node sampling context.

The JADE-supplied library closes and releases the current sampling file, which you can then analyze.

You can produce multiple files during a node lifetime, by using the **System** class **beginSample** and **endSample** methods, but you cannot sample statistics simultaneously on the same node.

For more details, see Chapter 4 of the *JADE Object Manager Guide*. (See also the **System** class **beginSampleGroupDefinition** method, for details about the sampling handle.)

endSampleGroupDefinition

Signature `endSampleGroupDefinition(samplingHandle: Integer);`

The **endSampleGroupDefinition** method of the **System** class terminates the sampling context identified in the **samplingHandle** parameter for the group of nodes.

For more details, see Chapter 4 of the *JADE Object Manager Guide*. See also the **System** class **beginSampleGroupDefinition** method, for details about the sampling handle.

findCharacterEntityByName

Signature
```
findCharacterEntityByName(name:        String;
                          number:      Integer output;
                          description: String output): Boolean;
```

The **findCharacterEntityByName** method of the **System** class returns **true** if the **name** parameter corresponds to a supported character entity name, and **false** otherwise.

If **true** is returned, the Unicode code point and a description for the character entity are returned in the output parameters **number** and **description**, respectively.

The following code fragment shows an example of the **findCharacterEntityByName** method and its parameters.

```
vars
    cp   : Integer;
    desc : String;
begin
    write system.findCharacterEntityByName("euro",cp,desc);  // "true"
    write cp;                                                //  8364
    write desc;                                              // "euro sign"
```

findCharacterEntityByNumber

Signature
```
findCharacterEntityByNumber(number:      Integer;
                            name:        String output;
                            description: String output): Boolean;
```

The **findCharacterEntityByNumber** method of the **System** class returns **true** if the **number** parameter corresponds to a supported character entity name, and **false** otherwise.

If this method returns **true**, the character entity name and description are returned in the output parameters **name** and **description**, respectively.

The following code fragment shows an example of the **findCharacterEntityByNumber** method and its parameters.

```
vars
    ent  : String;
    desc : String;
begin
    write system.findCharacterEntityByNumber(174,ent,desc);  // "true"
    write ent;                                               // "reg"
    write desc;           // "registered sign = registered trade mark sign"
```

forceOffUser

Signature `forceOffUser(node: Node; process: Process) serverExecution;`

The **forceOffUser** method of the **System** class requests the system object to force a sign-off operation for a specified process. The parameters of the **forceOffUser** method are listed in the following table.

Parameter	Description
node	Specifies the node to which the process belongs
process	Specifies the process that is to be forced off

The following example shows the use of the **forceOffUser** method:

```
vars
    allNodes: NodeDict;
    allProcesses: ProcessDict;
    nod: Node;
    proc: Process;
begin
    allNodes := system.nodes.cloneSelf(true);
    foreach nod in allNodes do
        allProcesses := nod.processes.cloneSelf(true);
        foreach proc in allProcesses do
            if process.signOnUserCode = "John" then
                system.forceOffUser(nod, proc);
                return;
            endif;
        endforeach;
        delete allProcesses;
    endforeach;
    return;
epilog
    delete allNodes;
    delete allProcesses;
end;
```

getAllUsers

Signature `getAllUsers(): StringArray;`

The **getAllUsers** method of the **System** class returns a reference to an array of all users on all nodes in the system.

getClassAccessFrequencies

Signature `getClassAccessFrequencies(clsNumArray: IntegerArray; freqArray: Integer64Array input);`

The **getClassAccessFrequencies** method of the **System** class returns access counts for specified classes. The class numbers of these classes are added to the array specified by the **clsNumArray** parameter before the method is called. The access counts for classes are held on the database server node, and are incremented every time an instance of that class or one of its subobjects is written to the database or fetched from the database. Only persistent object accesses are counted.

Notes If the **enableClassAccessFrequencies** method of the **Process** class has not been called to enable the counting of accesses to classes an exception is raised.

The access counts do not directly indicate how many times applications have used objects. If an object resides in the persistent object cache, it may not have to be fetched from the database when used. The access counts reflect database activity, rather than application activity.

The access counts are returned as **Integer64** values in the **freqArray** parameter, which is an instance of the **Integer64Array** class, passed to this method. Each entry in the **freqArray** array contains information relating to the class number specified by the entry with the same index in the **clsNumArray** array. If a class number is invalid, the corresponding access count is set to zero (**0**). The **freqArray** array is cleared every time the method is called.

The calling process is responsible for creating and deleting the two arrays used with this method.

The access counts are cumulative values, which do not get reset during the lifetime of the database server node, are held as 64-bit unsigned integer values, and are added to the **freqArray** array object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

When dealing with classes, retrieve the class number by using the **number** property of the **Class** class and find the class with a specific number by using the **getClassByNumber** method of the **Schema** class.

The following example shows the use of the **getClassAccessFrequencies** method.

```
showClassAccessFrequencies();
vars
   clsNumArray : IntegerArray;
   freqArray : Integer64Array;
   i : Integer;
   cls : Class;
begin
   create clsNumArray transient;
   create freqArray transient;
   foreach i in 2048 to 10000 do //check user classes
      clsNumArray.add(i);
   endforeach;
   system.getClassAccessFrequencies(clsNumArray,freqArray);
   foreach i in 1 to clsNumArray.size do
      if freqArray[i] > 0 then
         cls := currentSchema.getClassByNumber(clsNumArray[i]);
         write "Schema " & cls.schema.name &  " Class " & cls.name &
               " accesses= " & freqArray[i].String;
      endif;
   endforeach;
epilog
   delete clsNumArray;
   delete freqArray;
end;
```

The output from the **getClassAccessFrequencies** method shown in the previous example is as follows.

```
Schema CompilerSchemaImport Class GCompilerSchemaImport accesses= 1
Schema CompilerSchema Class C1 accesses= 6
Schema CompilerSchema Class GCompilerSchema accesses= 1
Schema CompilerSchemaSub Class GCompilerSchemaSub accesses= 1
Schema CompilerSchema Class C2 accesses= 4
```

```
Schema CompilerSchema Class C1Dict accesses= 2
Schema CompilerSchema Class C2Dict accesses= 4
Schema Martini Class GMartini accesses= 1
Schema Martini Class Root accesses= 1
Schema Martini Class SportsTeam accesses= 4
Schema CompilerVersioningTests Class GCompilerVersioningTests accesses= 1
Schema CompilerVersioningTests Class TestInfo accesses= 25
```

getDatabaseRole

Signature `getDatabaseRole(): Integer;`

The **getDatabaseRole** method of the **System** class returns an integer value that represents the database role of the server node on which the JADE system is running.

Note The **System** class **getDatabaseRole** method is an alias for the **JadeDatabaseAdmin** class **sdsGetDatabaseRole** method and it enables you to obtain the current database role for the JADE system in which it is executing without having to create and then delete an instance of the **JadeDatabaseAdmin** class.

The returned value is one of the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value
SDS_RolePrimary	1
SDS_RoleSecondary	2
SDS_RoleUndefined (returned when the method is invoked on a non-SDS-capable or non-RPS-capable system)	0

getDatabaseStats

Signature `getDatabaseStats(jdo: JadeDynamicObject input);`

The **getDatabaseStats** method of the **System** class returns statistics relating to persistent database activity. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The returned values are cumulative **Integer64** values representing counts of actions (fetching objects, updating objects, opening files, and so on) that do not get reset during the lifetime of the database server node. JADE applications that use the **getDatabaseStats** method defined in the **System** class therefore need to compare values from one call to the next, to work out the value differences.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

For details about the properties returned in the dynamic object, see "System::*getDatabaseStats* Method", in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. This method is most efficient when the properties match those to be returned.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getDatabaseStats** method.

```
tryDbStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   system.getDatabaseStats(jdo);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getDatabaseStats** method shown in the previous example is as follows.

```
---DatabaseStatistics(207)---
fileOpens = 30
fileCloses = 10
committedTrans = 13
abortedTrans = 0
checkPoints = 1
lastCheckPointDate = 14 May 2012
lastCheckPointTime = 15:40:39
lastCheckPointDuration = 1
maxCheckPointDuration = 1
avgCheckPointDuration = 1
editionGets = 190
objectGets = 17216
objectCreates = 23
objectUpdates = 34
objectDeletes = 0
dirtyReads = 0
osmReads = 0
priorEditionReads = 0
absentCollGets = 0
overflowDeleteGets = 0
```

getDatabaseSubrole

Signature `getDatabaseSubrole(): Integer;`

The **getDatabaseSubrole** method of the **System** class returns an integer value that represents the database subrole of the server node on which the JADE system is running.

Note The **System** class **getDatabaseSubrole** method is an alias for the **JadeDatabaseAdmin** class **sdsGetDatabaseSubrole** method and it enables you to obtain the current database subrole for the JADE system in which it is executing without having to create and then delete an instance of the **JadeDatabaseAdmin** class.

The returned value is one of the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value
SDS_RoleUndefined (returned when the method is invoked on a non-SDS-capable or non-RPS-capable system)	0

Global Constant	Integer Value
SDS_SubroleNative	1
SDS_SubroleRelational	2

getDbDiskCacheStats

Signature `getDbDiskCacheStats(jdo: JadeDynamicObject input);`

The **getDbDiskCacheStats** method of the **System** class returns statistics relating to the persistent database cache. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter. For details about the properties returned in the dynamic object, see "System::*getDbDiskCacheStats* Method", in Chapter 4 of the *JADE Object Manager Guide*. For further explanation of these values, refer to the JADE Monitor knowledge base.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. This method is most efficient when the properties match those to be returned.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getDbDiskCacheStats** method.

```
showDbDiskCacheStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   system.getDbDiskCacheStats(jdo);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **showDbDiskCacheStats** method shown in the previous example is as follows.

```
---DatabaseDiskCacheStatistics(210)---
cacheMisses = 21
gets = 8162
puts = 331
blockReads = 8
getsWithFetch = 0
putsWithFetch = 0
blocksFetched = 1258
blockReadsMultiple = 333
bufferReassigns = 0
bufferSteals = 0
maxHashCollisions = 0
maxConcFlushIos = 15
blockWrites = 4
blockWritesMultiple = 111
```

getDeltaDatabaseStatus

Signature `getDeltaDatabaseStatus(): Integer;`

The **getDeltaDatabaseStatus** method of the **System** class returns the delta database status represented by an **Integer** value listed in the following table.

Integer Value	Description
0	The server node is not specified to be delta database-capable
1	Inactive
2	Active
3	Being activated (this value is not currently used)
4	Being deactivated

getEnvironmentServerIdentity

Signature `getEnvironmentServerIdentity(): String;`

The **getEnvironmentServerIdentity** method of the **System** class returns the environment and server identities; for example:

```
24389438-15d7-e011-82f0-5ae520524153/24389438-15d7-e011-82f0-5ae520524153
```

getLockContentionInfo

Signature
```
getLockContentionInfo(obj:       Object;
                      lci:       LockContentionInfo input;
                      startTime: TimeStamp output);
```

The **getLockContentionInfo** method of the **System** class returns lock contention information for a single object specified by the **obj** parameter.

A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked. The information is copied into attributes of the **LockContentionInfo** instance specified by the **lci** parameter.

The calling process is responsible for creating and deleting the **LockContentionInfo** instance.

The **startTime** parameter is an output parameter that receives the date and time at which lock contention recording was started or restarted.

For details about the information available in **LockContentionInfo** instances, see "LockContentionInfo Class", in Chapter 4 of the *JADE Object Manager Guide*.

If there have been no lock contentions for the specified object, the values of the **totalContentions**, **maxWaitTime**, and **totalWaitTime** attributes are set to zero (**0**).

If this method is called when lock contentions are not being recorded, the **startTime** parameter and information in the **LockContentionInfo** instance are set to zero (**0**) values. The **beginLockContentionStats** and **endLockContentionStats** methods are used to control recording of lock contentions.

getLockContentionStats

Signature
```
getLockContentionStats(oa:             ObjectArray input;
                       maxEntries:     Integer;
                       minContentions: Integer;
                       startTime:      TimeStamp output);
```

The **getLockContentionStats** method of the **System** class retrieves lock contention information.

A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked. The information includes the number of lock contentions for individual objects, and the average and maximum times spent waiting to acquire a lock on each individual object.

Information is returned in transient instances of the **LockContentionInfo** class, added to the transient **ObjectArray** instance specified by the **oa** parameter.

The value of the **maxEntries** parameter specifies the maximum number of entries to be returned. Returned entries are added to the array in no particular order. When the **maxEntries** limit is reached, no more entries are added.

The value of the **minContentions** parameter specifies the minimum number of contentions for entries to be returned. Only entries with contention counts greater than or equal to the specified minimum are returned.

The **startTime** parameter is an output parameter that is set to the time when the lock contention recording was started or restarted. This enables you to calculate the number of contentions per second.

The calling method is responsible for creating and deleting the transient **ObjectArray** instance and for deleting **LockContentionInfo** instances in the array (for example, by using the **purge** method on the **ObjectArray** instance before deleting it).

When the method is called, any existing **LockContentionInfo** instances in the array are not removed. New **LockContentionInfo** instances are added to the end of the array.

If lock contentions are not being recorded when this method is called, no entries are added to the **ObjectArray** instance. You can use the **queryLockContentionStats** method of the **System** class to determine if lock contentions are currently being recorded.

For details about the information available in **LockContentionInfo** instances, see "LockContentionInfo Class", in Chapter 4 of the *JADE Object Manager Guide*.

Note A **LockContentionInfo** instance with a null object reference for the **target** value indicates that it holds combined information for all contentions that occurred on objects that could not be included in the table because the maximum table size had been reached.

getLocks

Signature
```
getLocks(locks:      LockArray input;
         maxEntries: Integer);
```

The **getLocks** method of the **System** class populates the array specified in the **locks** parameter with transient instances of the current persistent object locks held by all the processes in the system.

The parameters of the **getLocks** method are listed in the following table.

Parameter	Specifies the …
locks	Locks array that is to be populated with the lock instances
maxEntries	Maximum number of lock instances that are to be included in the array

The following example shows the use of the **getLocks** method:

```
vars
    lock      : Lock;
    lockArray : LockArray;
begin
    create lockArray transient;
    system.getLocks(lockArray, 40);
        foreach lock in lockArray do     //access the lock entry properties
            write lock.requestedBy.String;
            write lock.target.String;
    endforeach;
epilog
    lockArray.purge;
    delete lockArray;
end;
```

getMostAccessedClasses

Signature `getMostAccessedClasses(clsNumArray: IntegerArray input;`
`freqArray: Integer64Array input;`
`maxWanted: Integer);`

The **getMostAccessedClasses** method of the **System** class returns access counts for the classes that have been most frequently accessed since the database server node was initialized.

The access counts for classes are held on the database server node and are incremented every time an instance of that class or one of its subobjects is written to the database or fetched from the database. Only persistent object accesses are counted.

Note The access counts do not indicate how many times applications have used objects. If an object resides in the persistent object cache, it may not have to be fetched from the database when used. The access counts reflect database activity, rather than application activity.

The information is returned in a pair of arrays. The **clsNumArray** array contains a set of class numbers, and the **freqArray** array contains a matching set of access counts. Each entry in the **freqArray** array corresponds to the entry with the same index in the **clsNumArray** array.

The entries in the array are sorted in descending order of access count; that is, the class with the highest access count is the first array member, the class with the second highest access count is second, and so on.

The **maxWanted** parameter specifies the maximum number of entries to be placed in the arrays.

The calling process is responsible for creating and deleting the **clsNumArray** array and the **freqArray** array.

When the **getMostAccessedClasses** method is called, the arrays passed as parameters are cleared of all entries.

The access counts are cumulative values, which do not get reset during the lifetime of the database server node, are held as 64-bit unsigned integer values and added to the **freqArray** array object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

When dealing with classes, the class that has a particular number can be found using the **getClassByNumber** method of the **Schema** class.

The following example shows the use of the **getMostAccessedClasses** method.

```
showMostAccessedClasses();
vars
   clsNumArray: IntegerArray;
   freqArray: Integer64Array;
   ix : Integer;
   cls : Class;
begin
   create clsNumArray transient;
   create freqArray transient;
   system.getMostAccessedClasses(clsNumArray, freqArray, 1000);
   foreach ix in 1 to clsNumArray.size do
      if clsNumArray[ix] > 2047 then //only show user classes
         cls := currentSchema.getClassByNumber(clsNumArray[ix]);
         write "Schema " & cls.schema.name & " Class " & cls.name
             & " Accesses=" & freqArray[ix].String;
      endif;
   endforeach;
epilog
   delete clsNumArray;
   delete freqArray;
end;
```

The output from the **getMostAccessedClasses** method shown in the previous example is as follows.

```
Schema CompilerVersioningTests Class TestInfo Accesses=25
Schema CompilerSchema Class C1 Accesses=6
Schema CompilerSchema Class C2 Accesses=4
Schema CompilerSchema Class C2Dict Accesses=4
Schema Martini Class SportsTeam Accesses=4
Schema CompilerSchema Class C1Dict Accesses=2
Schema CompilerSchemaImport Class GCompilerSchemaImport Accesses=1
Schema CompilerSchema Class GCompilerSchema Accesses=1
Schema CompilerSchemaSub Class GCompilerSchemaSub Accesses=1
Schema Martini Class GMartini Accesses=1
Schema Martini Class Root Accesses=1
Schema CompilerVersioningTests Class GCompilerVersioningTests Accesses=1
```

getNotes

Signature

```
getNotes(notes:      NotificationArray input;
         transients: Boolean;
         maxEntries: Integer);
```

The **getNotes** method of the **System** class populates the array specified in the **notes** parameter with transient instances of the current notification requests by all the processes in the system.

The parameters of the **getNotes** method are listed in the following table.

Parameter	Specifies…
notes	The notifications array that is to be populated with the notification instances.

Parameter	Specifies…
transients	Whether the notifications to be reported correspond to target transient objects corresponding to this node (**true**) or to target persistent objects corresponding to all the nodes in the system (**false**).
maxEntries	The maximum number of notification instances to include in the array.

Note As this method creates transient instances of the **Notification** class, it is the responsibility of the method caller to purge the collection used by the method to delete these transient instances. The collection should be purged *before* the deletion of the notification array passed to the method in the **notes** parameter.

The following examples show the use of the **getNotes** method.

```
vars
    note              : Notification;
    notificationArray : NotificationArray;
begin
    create notificationArray transient;
    system.getNotes(notificationArray, true, 100);
        foreach note in notificationArray do
            //access the notification entry properties
            write note.target.String;
            // now check subscriber class is valid for the user
            if app.isValidObject(note.subscriber) then
                write note.subscriber.String;
            endif;
        endforeach;
epilog
    notificationArray.purge;
    delete notificationArray;
end;

vars
    notificationArray : NotificationArray;
begin
    create notificationArray transient;
    system.getNotes(notificationArray, true, 32000);
    write notificationArray.size.String & ' transient notifications';
    notificationArray.clear;
    system.getNotes(notificationArray, false, 32000);
    write notificationArray.size.String & ' persistent notifications';
epilog
    notificationArray.purge;
    delete notificationArray;
end;
```

getObjectLockProcesses

Signature

```
getObjectLockProcesses(locktarget: Object;
                       processes:  ProcessDict input;
                       maxEntries: Integer);
```

The **getObjectLockProcesses** method of the **System** class populates the dictionary specified in the **processes** parameter with all processes that have locks on the object referenced by the **locktarget** parameter.

The parameters of the **getObjectLockProcesses** method are listed in the following table.

Parameter	Description
locktarget	Specifies the object whose locks are to be obtained.
processes	Specifies the process dictionary that is to be populated with processes that have locks on the object referenced by the **locktarget** parameter.
maxEntries	Specifies the maximum number of object lock process instances to include in the dictionary.

The following example shows the use of the **getObjectLockProcesses** method.

```
writeWhoHasTheLocks(target: Object);
vars
    process     : Process;
    processDict : ProcessDict;
begin
    create processDict transient;
    system.getObjectLockProcesses(target, processDict, 10);
    foreach process in processDict do
        write process.userCode;
    endforeach;
epilog
    delete processDict;
end;
```

getObjectPartitionID

Signature `getObjectPartitionID(object: Object): Integer64;`

The **getObjectPartitionID** method of the **System** class returns the identifier of the database file partition in which the object specified in the **object** parameter is located. (See also the **moveToPartition** method of the **Object** class.)

getQueuedLocks

Signature
```
getQueuedLocks(locks:      LockArray input;
               maxEntries: Integer);
```

The **getQueuedLocks** method of the **System** class is similar to the **getObjectLockProcesses** and **getLocks** methods, but it includes only the lock requests that are waiting for objects to be unlocked by the processes that currently have them locked.

The value of the **maxEntries** parameter specifies the maximum number of entries to be inserted into the array specified by the **locks** parameter. Entries are inserted in no particular order.

The following example shows the use of the **getQueuedLocks** method.

```
vars
    lock      : Lock;
    lockArray : LockArray;
begin
    create lockArray transient;
    system.getQueuedLocks(lockArray, 40);
    foreach lock in lockArray do      //access the lock entry properties
        write lock.requestedBy.String;
```

```
            write lock.elapsedTime.String;
            write lock.waitTime.String;
        endforeach;
    epilog
        lockArray.purge;
        delete lockArray;
    end;
```

Lock objects returned in the **locks** parameter can have lock entries in the array that have the **Lock** class **lockedBy** property set to **null** if the lock request is still waiting to be processed in the lock queue.

When this occurs, the process that caused the lock request to be queued has already released it but because of high activity on the executing node, the lock request has not been retried.

getRequestStats

Signature `getRequestStats(jdo: JadeDynamicObject input);`

The **getRequestStats** method of the **System** class returns system statistics relating to requests carried out by the database server node. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The system statistics are held on the database server node. For details about the properties returned in the dynamic object, see "LockContentionInfo Class" in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. This method is most efficient when the properties match those to be returned.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getRequestStats** method.

```
showSystemRequestStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   system.getRequestStats(jdo);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getRequestStats** method shown in the previous example is as follows.

```
---SystemStatistics(105)---
committedTransactions = 114
abortedTransactions = 0
getObjects = 41561
queuedLocks = 0
createObjects = 433
```

```
deleteObjects = 160
updateObjects = 703
lockObjects = 22444
unlockObjects = 12310
beginNotifications = 686
endNotifications = 33
deliveredNotifications = 169
serverMethodExecutions = 0
totalLockQueueWaitTime = 0
causeEvents = 70
```

getRpcServerStatistics

Signature `getRpcServerStatistics(jdo: JadeDynamicObject input detailed: Boolean);`

The **getRpcServerStatistics** method of the **System** class RPC statistics relating to activity between the database server node and all client nodes. The values returned represent information about the connection between client nodes and the database server, and totals for requests received and replies sent. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

The **detailed** parameter specifies whether the values returned should be combined for all requests, or individual totals for each request type.

For details about the attributes returned in the dynamic object properties, see "System::*getRpcServerStatistics* Method", in Chapter 4 of the *JADE Object Manager Guide*.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. Properties are added to the object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. This method is most efficient when the properties match those to be returned.

The cumulative values are held as 64-bit unsigned integer values, and are copied to the dynamic object as **Integer64** values. The maximum value before they wrap around to negative values is therefore **2^63 - 1** (approximately 8 Exabytes).

The following example shows the use of the **getRpcServerStatistics** method.

```
showRpcServerStats();
vars
   jdo : JadeDynamicObject;
begin
   create jdo transient;
   system.getRpcServerStatistics(jdo, false);
   write jdo.display;
epilog
   delete jdo;
end;
```

The output from the **getRpcServerStatistics** method shown in the previous example is as follows.

```
---RPCServerStatistics(106)---
timeStarted = 27 April 2007, 12:31:14
```

```
connectionType = 0
lastInboundRequest = 27 April 2007, 14:31:32
requestsFromClients = 22551
repliesToClients = 22550
requestPacketsFromClients = 22551
replyPacketsToClients = 22550
requestBytesFromClients = 3475340
replyBytesToClients = 9598785
requestsToClients = 31
repliesFromClients = 31
requestPacketsToClients = 31
replyPacketsFromClients = 31
requestBytesToClients = 35313
replyBytesFromClients = 16665
notificationPacketsToClients = 0
notificationBytesToClients = 0
```

getStatistics

Signature

```
getStatistics(committedTransactions:  Integer output;
              abortedTransactions:    Integer output;
              getObjects:             Integer output;
              queuedLocks:            Integer output;
              createObjects:          Integer output;
              deleteObjects:          Integer output;
              updateObjects:          Integer output;
              lockObjects:            Integer output;
              unlockObjects:          Integer output;
              beginNotifications:     Integer output;
              endNotifications:       Integer output;
              deliveredNotifications: Integer output;
              serverMethodExecutions: Integer output);
```

The **getStatistics** method of the **System** class loads the values of all the specified parameters with the corresponding system statistics.

Note This method raises exception 1406 if any of the statistic values exceed **Max_Integer** (this can happen if your JADE system has been up for a long time; that is, the actual number of operations exceeds **Max_Integer**).

The parameters for the **getStatistics** method are listed in the following table.

Parameter	Obtains the number of...
committedTransactions	Committed transactions
abortedTransactions	Aborted transactions
getObjects	**getObject** operations performed
queuedLocks	**queuedLock** operations performed
createObjects	**createObject** operations performed
deleteObjects	**deleteObject** operations performed
updateObjects	**updateObject** operations performed

Parameter	Obtains the number of...
lockObjects	**lockObject** operations performed
unlockObjects	**unlockObject** operations performed
beginNotifications	**beginNotification** operations performed
endNotifications	**endNotification** operations performed
deliveredNotifications	Notifications that were sent
serverMethodExecutions	Methods executed in the server node operations

The following example shows the use of the **getStatistics** method.

```
vars
    committedTransactions  : Integer;
    abortedTransactions    : Integer;
    getObjects             : Integer;
    queuedLocks            : Integer;
    createObjects          : Integer;
    deleteObjects          : Integer;
    updateObjects          : Integer;
    lockObjects            : Integer;
    unlockObjects          : Integer;
    beginNotifications     : Integer;
    endNotifications       : Integer;
    deliveredNotifications : Integer;
    serverMethodExecutions : Integer;
begin
    system.getStatistics(committedTransactions,
        abortedTransactions,
        getObjects,
        queuedLocks,
        createObjects,
        deleteObjects,
        updateObjects,
        lockObjects,
        unlockObjects,
        beginNotifications,
        endNotifications,
        deliveredNotifications,
        serverMethodExecutions);
end;
```

getStatistics64

Signature

```
getStatistics64(committedTransactions:  Integer64 output;
                abortedTransactions:    Integer64 output;
                getObjects:             Integer64 output;
                queuedLocks:            Integer64 output;
                createObjects:          Integer64 output;
                deleteObjects:          Integer64 output;
                updateObjects:          Integer64 output;
                lockObjects:            Integer64 output;
                unlockObjects:          Integer64 output;
                beginNotifications:     Integer64 output;
                endNotifications:       Integer64 output;
                deliveredNotifications: Integer64 output;
                serverMethodExecutions: Integer64 output);
```

The **getStatistics64** method of the **System** class loads the values of all the specified parameters with the corresponding system statistics.

The parameters for the **getStatistics64** method are listed in the following table.

Parameter	Obtains the number of...
committedTransactions	Committed transactions
abortedTransactions	Aborted transactions
getObjects	**getObject** operations performed
queuedLocks	**queuedLock** operations performed
createObjects	**createObject** operations performed
deleteObjects	**deleteObject** operations performed
updateObjects	**updateObject** operations performed
lockObjects	**lockObject** operations performed
unlockObjects	**unlockObject** operations performed
beginNotifications	**beginNotification** operations performed
endNotifications	**endNotification** operations performed
deliveredNotifications	Notifications that were sent
serverMethodExecutions	Methods executed in the server node operations

The following example shows the use of the **getStatistics** method.

```
vars
    committedTransactions  : Integer64;
    abortedTransactions    : Integer64;
    getObjects             : Integer64;
    queuedLocks            : Integer64;
    createObjects          : Integer64;
    deleteObjects          : Integer64;
    updateObjects          : Integer64;
    lockObjects            : Integer64;
    unlockObjects          : Integer64;
```

```
    beginNotifications     : Integer64;
    endNotifications       : Integer64;
    deliveredNotifications : Integer64;
    serverMethodExecutions : Integer64;
begin
    system.getStatistics64(committedTransactions,
        abortedTransactions,
        getObjects,
        queuedLocks,
        createObjects,
        deleteObjects,
        updateObjects,
        lockObjects,
        unlockObjects,
        beginNotifications,
        endNotifications,
        deliveredNotifications,
        serverMethodExecutions);
end;
```

getSystemSequenceNumberNext

Signature `getSystemSequenceNumberNext(name: String): Integer64;`

The **getSystemSequenceNumberNext** method of the **System** class increments the current value of the *system-sequence-number* specified by the **name** parameter and returns the new value. The range of numbers returned is **1** through **Max_Integer64**, unless the system sequence number has not been created when the method returns zero (**0**).

Note The sequence number should be initialized on system startup, by using the **createSystemSequenceNumber** method.

If the method returns **Max_Integer64** for the specified system sequence number, all subsequent calls for that system sequence number raise a 1456 (*The SystemSequenceNumber has reached the maximum value (Max_Integer64)*) exception. All access to the system sequence number table is single-threaded and is independent of process transaction state.

Ensure that the initial value passed to the **createSystemSequenceNumber** method does not cause the **getSystemSequenceNumberNext** method to return an already used number.

If the **getSystemSequenceNumberNext** method returns zero (**0**), determine the highest number that has been assigned to an object stored in the database and call the **createSystemSequenceNumber** method passing that value.

If an object obtains a sequence number but the object is not persisted because the transaction is aborted, there will be a gap in the stored number sequence.

The following method returns the next available customer number, where the customer number is used as a key in an exclusive **MemberKeyDictionary** collection owned by a **Company** object.

```
getNextCustomerNumber(): Integer64;
constants
    SSN_CustomerNumber :String = "MySchema::CustomerNumber";
vars
    nextNumber : Integer64;
    coy : Company;
```

```
    cust : Customer;
begin
    nextNumber := system.getSystemSequenceNumberNext(SSN_CustomerNumber);
    if nextNumber = 0 then
       coy := Company.firstInstance();
       if coy <> null then
            cust := coy.allCustomersByNumber.last();
           if cust <> null then
               nextNumber := cust.number
           endif;
       endif;
       system.createSystemSequenceNumber(SSN_CustomerNumber, nextNumber);
       nextNumber := system.getSystemSequenceNumberNext(SSN_CustomerNumber);
    endif;
    return nextNumber;
end;
```

getTimeInTransactionState

Signature `getTimeInTransactionState(p: Process): Integer;`

The **getTimeInTransactionState** method of the **System** class returns the number of milliseconds that a process is in transaction state.

Note This method applies only to persistent transactions.

interruptUser

Signature
```
interruptUser(node:    Node;
              process: Process) serverExecution;
```

The **interruptUser** method of the **System** class causes a conditional interruption of a specified process. The parameters for the **interruptUser** method are listed in the following table.

Parameter	Specifies the…
node	Node to which the process belongs
process	Process that is to be interrupted

The target process, when interrupted, receives a continuable user interrupted execution-type exception.

The following example shows the use of the **interruptUser** method.

```
vars
    allNodes: NodeDict;
    allProcesses: ProcessDict;
    nod: Node;
    proc: Process;
begin
    allNodes := system.nodes.cloneSelf(true);
    foreach nod in allNodes do
        allProcesses := nod.processes.cloneSelf(true);
        foreach proc in allProcesses do
            if process.signOnUserCode = "Wilbur" then
```

```
                    system.interruptUser(nod, process);
                    return;
                endif;
            endforeach;
            delete allProcesses;
        endforeach;
        return;
    epilog
        delete allNodes;
        delete allProcesses;
    end;
```

The following example shows the handling of a conditional interrupt in an exception handler.

```
    vars
    begin
        if exObj.continuable then
            if allowInterrupt then
                return Ex_Abort_Action;
            endif;
            return Ex_Continue;
        endif;
    end;
```

isDatabaseEncryptionEnabled

Signature `isDatabaseEncryptionEnabled(): Boolean;`

The **isDatabaseEncryptionEnabled** method of the **System** class returns **true** if the database encryption is enabled; otherwise it returns **false**.

isDbArchival

Signature `isDbArchival(): Boolean;`

The **isDbArchival** method of the **System** class returns **true** if database archival recovery is enabled for the server node on which the JADE system is running.

isRemoteSamplingEnabled

Signature
```
isRemoteSamplingEnabled(samplingHandle: Integer;
                        n:              Node): Boolean;
```

The **isRemoteSamplingEnabled** method of the **System** class returns **true** if the node specified in the **n** parameter is included in the sample definition group identified in the **samplingHandle** parameter.

The sampling for the context identified in the **samplingHandle** parameter is started by using the **beginSample** method. For details, see Chapter 4 of the *JADE Object Manager Guide*. See also the **System** class **beginSampleGroupDefinition** method, for details about the sampling handle.

isValidProcess

Signature `isValidProcess(process: Process): Boolean;`

The **isValidProcess** method of the **System** class returns **true** if the process specified by the **process** parameter represents a signed-on application.

A **Process** instance without a corresponding signed-on application can exist under certain circumstances; for example, if an error occurs during process sign off that prevents the **Process** instance from being deleted. This method can be used to identify these *zombie* **Process** instances.

When a zombie process is encountered in a monitor operation, the instance is deleted; for example, an interrupt or force off user, call stack request, and so on.

logObjectCaches

Signature

```
logObjectCaches(samplingHandle:              Integer;
                persistentCacheStats:        Boolean;
                persistentCacheBuffers:      Boolean;
                transientCacheStats:         Boolean;
                transientCacheBuffers:       Boolean;
                remoteTransientCacheStats:   Boolean;
                remoteTransientCacheBuffers: Boolean;
                userNumber:                  Integer;
                userText:                    String);
```

The **logObjectCaches** method of the **System** class specifies the object cache statistics that are logged by invoking the **NodeSampleCacheInfoCallBack** or the **NodeSampleObjectBuffer** entry point, or both of these entry points, in the user library for each of the nodes in the sample definition group.

The JADE-supplied library logs the statistics to the file specified in the **initializationParameter** parameter of the **System** class **beginSample** method and writes the following statistics to your output file on each node in the group.

- Cache header record (type **1**) for cache statistics
- Cache buffer records (type **2**) for individual object buffers

The **logObjectCaches** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSampleGroupDefinition** method when sampling started
persistentCacheStats	Logs statistics of the persistent objects cache
persistentCacheBuffers	Logs statistics of the persistent object cache buffers
transientCacheStats	Logs statistics of the transient objects cache
transientCacheBuffers	Logs statistics of the transient object cache buffers
remoteTransientCacheStats	Logs statistics of the remote transient objects cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache buffers
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the cache statistics that you require, set the appropriate Boolean cache parameters to **true**. The following code fragment shows an example of the **logObjectCaches** method and its parameters.

```
system.logObjectCaches(samplingHandle, true, true, false, false, false,
          false, 50, "After the load data operation");
```

All buffers containing non-shared transient objects are listed when node sampling snapshots are requested. For details, see "Statistics File Format", in Chapter 4 of the *JADE Object Manager Guide*.

logRequestStatistics

Signature
```
logRequestStatistics(samplingHandle: Integer;
                     local:          Boolean;
                     remote:         Boolean;
                     userNumber:     Integer;
                     userText:       String);
```

The **logRequestStatistics** method of the **System** class specifies the request statistics that are logged for all processes in each of the nodes in the sample definition group, by invoking the **NodeSampleRequestStatisticsCallBack** entry point in the user library.

The JADE-supplied library automatically writes the following statistics.

- Local request statistics record (type **8**)
- Remote request statistics record (type **9**)

The **logRequestStatistics** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSampleGroupDefinition** method when sampling started
local	Logs statistics of all requests invoked on the local node
remote	Logs statistics of all requests from the local node to remote nodes
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the request statistics that you require, set the appropriate Boolean cache parameters to **true**.

The user number and text values specified in the **userNumber** and **userText** parameters are written in the corresponding records.

The following code fragment shows an example of the **logRequestStatistics** method and its parameters.

```
system.logRequestStatistics(samplingHandle, true, true, 23, "Before
            method m1");
```

For details, see "Statistics File Format", in Chapter 4 of the *JADE Object Manager Guide*.

logUserCommand

Signature
```
logUserCommand(samplingHandle: Integer;
               command:        String;
               userNumber:     Integer;
               userText:       String);
```

The **logUserCommand** method of the **System** class causes the invocation of the **NodeSampleUserCommandCallBack** entry point in the user library for each of the nodes in the sample definition group, passing the **command** parameter to it.

The **logUserCommand** method parameters are listed in the following table.

Parameter	Description
samplingHandle	Identifies the sampling context returned by the **beginSampleGroupDefinition** method when sampling started
command	Action specific to your user library (for example, the JADE-supplied library uses this command for filtering)
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The JADE supplied library automatically writes the user command (type **13**).

For details, see "JADE Sampling Libraries" and "Statistics File Format", in Chapter 4 of the *JADE Object Manager Guide*.

processDumpAllNodes

Signature `processDumpAllNodes();`

The **processDumpAllNodes** method of the **System** class invokes a near-simultaneous process dump of all nodes attached to a database server and the database server node itself.

queryLockContentionStats

Signature
```
queryLockContentionStats(active:       Boolean output;
                         startingProc: Process output;
                         maxEntries:   Integer output;
                         startTime:    TimeStamp output);
```

The **queryLockContentionStats** method of the **System** class retrieves information about the current recording of lock contentions.

A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked. For details about the information returned in the output parameters, see "System::*queryLockContentionStats* Method", in Chapter 4 of the *JADE Object Manager Guide*.

Note The recording of lock contentions is under the control of the process that initiated the recording; that is, only that process can stop or restart recording of lock contentions. Although any process can use the **queryLockContentionStats** method, it should be assume that recording can stop or restart at any time, if it is not the process that started the recording.

removeNode

Signature `removeNode(node: Node) serverExecution;`

The **removeNode** method of the **System** class requests the system object to force a sign-off operation of all users on the node specified in the **node** parameter.

sdsAuditEnableSecondaryApps

Signature `sdsAuditEnableSecondaryApps();`

The **sdsAuditEnableSecondaryApps** method of the **System** class is intended for execution on a JADE Synchronized Database Service (SDS) primary system. It writes an audit record into the journal that, when replayed on the secondary, starts server applications and enables user sign-on.

It is typically used after a number of schema loads that have changed class definitions but have not required the schema to be versioned. In such cases, applications are stopped on the secondary system to prevent outdated class definitions being used.

Where there is no transition phase to identify the end of the sequence of schema loads, that point can be identified by executing the **sdsAuditEnableSecondaryApps** method on the primary system.

verifyDbEncryptionMasterKey

Signature `verifyDbEncryptionMasterKey(): Boolean;`

The **verifyDbEncryptionMasterKey** method of the **System** class specifies whether the database encryption master key is present and correct; otherwise it returns **false**.

SystemException Class

The **SystemException** class is the superclass of all exceptions relating to errors detected by the JADE kernel.

Inherits From: NormalException

Inherited By: DeadlockException, IntegrityViolation, LockException, NotificationException

TcpIpConnection Class

The **TcpIpConnection** class implements the interface defined by the **Connection** class specifically for the *Transmission Control Protocol / Internet Protocol* (TCP/IP) API. The **TcpIpConnection** class supports both synchronous and asynchronous operations. Asynchronous methods have a receiver object and a message (method name) specified as parameters. When the method completes, the specified (callback) method of the object is called. The callback method must match the signature required by the calling asynchronous method.

Only one synchronous operation can be performed at one time. Only one synchronous or asynchronous read operation can be performed at one time on a connection. Many asynchronous write operations can be performed at the same time on one connection.

Notes As you can create a **TcpIpConnection** object as a shared transient object, you can pass it to another JADE process on the same JADE node, if required. Shared transient TCP/IP connection objects enable you to create a communicator application that passes on messages to worker threads and to share connections between processes so that a new connection can be passed on to a worker application. Ensure that you are in shared transient transaction state before you create or delete a **TcpIpConnection** object, by setting the **port** property or the **Connection** class **name** property.

The **Connection** class **name** property for a **TcpIpConnection** object may be set to a valid IP address.

For details about the constants, properties, and methods defined in the **TcpIpConnection** class, see "TcpIpConnection Constants", "TcpIpConnection Properties", and "TcpIpConnection Methods", in the following subsections.

Inherits From: Connection

Inherited By: JadeInternetTCPIPConnection

TcpIpConnection Class Constants

The constants provided by the **TcpIpConnection** class are listed in the following table.

Constant	Integer Value	Constant	Integer Value
ProtocolFamilyTcpIPAny	-1	ProtocolFamilyTcpIPv4	0
		ProtocolFamilyTcpIPv6	1

TcpIpConnection Properties

The properties defined in the **TcpIpConnection** class are summarized in the following table.

Property	Description
authenticationLibrary	Contains the name of the library that contains the authentication method
cryptLibrary	Contains the name of the library that contains the encryption and decryption methods
decryptMethod	Contains the name of the decryption method in the encryption and decryption library
encryptMethod	Contains the name of the encryption method in the encryption and decryption library

Property	Description
genAuthChallengeMethod	Contains the name of the method used to generate the authentication challenge
genAuthResponseMethod	Contains the name of the method used to generate the response from the authentication challenge
localInterface	Contains the interface name or IP address of a local network interface
localIpAddress	Contains the local IP address
localPort	Contains the local service port
networkProxy	Contains a reference to the object identifier (oid) of the proxy in the **TcpIpConnection** class object
port	Contains the target service port or listen port
protocolFamily	Contains the protocol used by the connection
remoteIpAddress	Contains the remote host IP address
remoteName	Contains the remote host name
remotePort	Contains the port number used on the remote node
resolveRemoteName	Specifies whether the remote host name is to be located
usePresentationClient	Specifies whether the connection is opened on the thin client or application server
sslContext	Causes the core network facilities to use SSL instead of TCP/IP when a connection is active
userObject	Contains an object to associate with any TCP/IP connection
verifyAuthResponseMethod	Contains the name of the method used to verify the response to the authentication challenge

authenticationLibrary

Type: String

The **authenticationLibrary** property of the **TcpIpConnection** class contains the name of the library that contains the **genAuthChallengeMethod**, **genAuthResponseMethod**, and **verifyAuthResponseMethod** authentication methods.

The code fragment in the following example shows the use of the **authenticationLibrary** property.

```
authConf := AuthConf.firstInstance;
if authConf <> null then
    self.tcp.authenticationLibrary    := authConf.authLib;
    self.tcp.genAuthChallengeMethod   := authConf.challengeMethod;
    self.tcp.genAuthResponseMethod    := authConf.responseMethod;
    self.tcp.verifyAuthResponseMethod := authConf.verifyMethod;
endif;
```

cryptLibrary

Type: String[128]

The **cryptLibrary** property of the **TcpIpConnection** class contains the name of the library that contains the **encryptMethod** and **decryptMethod** methods.

The code fragment in the following example shows the use of the **cryptLibrary** property.

```
cryptConf := CryptConf.firstInstance;
if cryptConf <> null then
    self.tcp.cryptLibrary  := cryptConf.cryptLib;
    self.tcp.encryptMethod := cryptConf.encryptMethod;
    self.tcp.decryptMethod := cryptConf.decryptMethod;
endif;
```

decryptMethod

Type: String

The **decryptMethod** property of the **TcpIpConnection** class contains the name of the decryption method in the encryption and decryption library that is executed after a successful **readBinary** or **readBinaryAsynch** operation.

The following example sets the decryption method for the connection and checks to make sure TCP is in connected state (**2**). If it is, and binary data is received through the connection, it displays the data in the text box. The parameter of **50** specifies that the data must be no more than 50 bytes long.

```
buttonReceive_click(btn: Button input) updating;
begin
    self.tcp.decryptMethod := "okDecrypt";
    if self.tcp.state = Connection.Connected then
        textBox1.text := self.tcp.readBinary(50).String;
    endif;
end;
```

encryptMethod

Type: String

The **encryptMethod** property of the **TcpIpConnection** class contains the name of the encryption method in the encryption and decryption library that is executed after a successful **writeBinary** or **writeBinaryAsynch** operation.

The code fragment in the following example shows the use of the **encryptMethod** property.

```
if cryptConf <> null then
    self.tcp.cryptLibrary  := cryptConf.cryptLib;
    self.tcp.encryptMethod := cryptConf.encryptMethod;
    self.tcp.decryptMethod := cryptConf.decryptMethod;
endif;
```

genAuthChallengeMethod

Type: String

The **genAuthChallengeMethod** property of the **TcpIpConnection** class contains the name of the method in the authentication library that is used to generate the authentication challenge.

If this property contains the name of a generate authentication challenge method, the specified method is executed after a successful **listen** or **listenAsynch** operation.

The following example shows the setting of the authentication challenge and verification methods for a connection.

```
buttonListen_click(btn: Button input) updating;
begin
    self.tcp.genAuthChallengeMethod   := "okGenAuthChallenge";
    self.tcp.verifyAuthResponseMethod := "okVerifyAuthResponse";
    // Sets the TCP to listen on the current port.  If a connection
    // is made, sets the status bar to read 'connected' and fills the
    // text boxes with the IP address and name information.
    self.tcp.listen;
    if self.tcp.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox3.text       := self.tcp.localIpAddress;
        textBox2.text       := self.tcp.remoteIpAddress;
        textBox4.text       := self.tcp.name;
    endif;
end;
```

genAuthResponseMethod

Type: String

The **genAuthResponseMethod** property of the **TcpIpConnection** class contains the name of the method in the authentication library used to generate the response from the authentication challenge.

If this property contains the name of a generate authentication response method, the specified method is executed after a successful **open** or **openAsynch** operation.

The following example of the **genAuthResponseMethod** property sets the authentication response method for the connection.

```
buttonOpen_click(btn: Button input) updating;
begin
    self.tcp.genAuthResponseMethod := "okGenAuthResponse";
    // Attempts to connect to the current port.  If an application is
    // listening on the port, a connection is made and the status bar
    // is set to read 'connected'.
    self.tcp.open;
    if self.tcp.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox2.text := self.tcp.localIpAddress;
        textBox3.text := self.tcp.remoteIpAddress;
        textBox4.text := self.tcp.remoteName;
    endif;
end;
```

localInterface

Type: String[128]

The **localInterface** property of the **TcpIpConnection** class contains the local interface name or IP address of a local network interface.

Note Use this property only if you want to receive new connections from a specific local interface. By default, JADE receives connections on all local interfaces. For example, to allow an administrator to ensure connections from clients connect on the fastest interface or to allow easier security when used in conjunction with a firewall or router access list, specify the local interface name or IP address if you want to select a specific network adapter in a server node that has more than one network adapter installed.

localIpAddress

Type: String

The read-only **localIpAddress** property of the **TcpIpConnection** class contains the local IP address after the successful establishment of a TCP/IP connection.

The code fragment in the following example shows the use of the **localIpAddress** property.

```
if self.tcp.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox3.text       := self.tcp.localIpAddress;
    textBox2.text       := self.tcp.remoteIpAddress;
    textBox4.text       := self.tcp.name;
endif;
```

localPort

Type: Integer

The **localPort** property of the **TcpIpConnection** class contains the local service port when using the **open** or **openAsynch** method.

Note Use the property only if you want to connect through a specific local port. The default value of zero (**0**) indicates that JADE connects through any available local port.

networkProxy

Type: JadeTcpIpProxy

The **networkProxy** property of the **TcpIpConnection** class contains a reference to a **JadeTcpIpProxy** object identifier of the proxy in the **TcpIpConnection** class object.

If this reference contains a non-null value, the **JadeTcpIpProxy** class **connect** method is executed, which connects to a proxy server, asking it to in turn connect to the destination address and port. You can reimplement the **JadeTcpIpProxy** class **connect** method. If the **networkProxy** property value is **null**, the **TcpIpConnection** class **open** or **openAsynch** method is executed.

Network proxies are supported only for the **TcpIpConnection** class **open** or **openAsynch** method.

port

Type: Integer

The **port** property of the **TcpIpConnection** class contains the target service port when using the **open** or **openAsynch** method or it defines the listen port when using the **listen** or **listenAsynch** method.

The code fragment in the following example shows the use of the **port** property.

```
// Creates a normal TCP/IP connection, sets the name to the current
// computer name, and sets the listen port to 7895.
create tcp;
self.tcp.name := app.computerName;
self.tcp.port := 7895;
```

protocolFamily

Type: Integer

The **protocolFamily** property of the **TcpIpConnection** class contains the protocol used by the connection.

The **protocolFamily** property values are listed in the following table.

Class Constant	Integer Value	Description
ProtocolFamilyTcpIPv4	0	TCP/IP version 4 protocol
ProtocolFamilyTcpIPv6	1	TCP/IP version 6 protocol
ProtocolFamilyTcpIPAny	-1	TCP/IP version 4 or version 6 protocol

If you do not change your code, your existing code runs using TCP/IP version 4 only.

Note The **JadeTcpIpProxy** class currently works only with the TCP/IP version 4 protocol.

remoteIpAddress

Type: String

The read-only **remoteIpAddress** property of the **TcpIpConnection** class contains the remote host IP address after a successful **open**, **openAsynch**, **listen**, or **listenAsynch** method.

The code fragment in the following example shows the use of the **remoteIpAddress** property.

```
self.tcp.listenContinuousAsynch(conlog, "updateListenContinuousCalls");
if self.tcp.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox2.text       := self.tcp2.remoteIpAddress;
    textBox3.text       := self.tcp2.remotePort.String;
    textBox4.text       := self.tcp2.localIpAddress;
    textBox5.text       := self.tcp2.name;
endif;
```

remoteName

Type: String

The read-only **remoteName** property of the **TcpIpConnection** class contains the remote host name of the remote node in the local HOSTS file or the Domain Name Service (DNS) node after a successful **open**, **openAsynch**, **listen**, or **listenAsynch** method.

If the name of the remote host cannot be determined, the **remoteName** property is zero-length.

The code fragment in the following example shows the use of the **remoteName** property.

```
if self.tcp.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox2.text       := self.tcp.localIpAddress;
    textBox3.text       := self.tcp.remoteIpAddress;
    textBox4.text       := self.tcp.remoteName;
    textBox5.text       := self.tcp.remotePort.String;
endif;
```

remotePort

Type: Integer

The read-only **remotePort** property of the **TcpIpConnection** class contains the port number used on the remote node after a successful **open**, **openAsynch**, **listen**, or **listenAsynch** method call.

resolveRemoteName

Type: Boolean

The **resolveRemoteName** property of the **TcpIpConnection** class specifies whether the remote host name must be resolved from the IP address after a successful **listen** or **listenAsynch** method.

If this property is set to **false** (the default value), the **remoteName** property contains a zero-length string.

usePresentationClient

Type: Integer

The **usePresentationClient** property of the **TcpIpConnection** class specifies whether the connection is opened on the presentation client or application server.

By default, the connection is opened on the application server; that is, this value is set to **false**. To open the connection on the presentation client, set this property to **true**.

Note This property is ignored when the application is running from a standard client.

sslContext

Type: JadeSSLContext

The **sslContext** property of the **TcpIpConnection** class causes the core network facilities to use the SSL instead of TCP/IP protocol when a connection is active.

Note Asynchronous connection operations are executed on another thread. If this asynchronous worker thread needs to access JADE objects (for example, the **TcpIpConnection**, **JadeSSLContext**, and **JadeX509Certificate** objects), these objects need to be shared transient or persistent objects.

The following example shows the use of the **sslContext** property to open an outgoing SSL connection.

```
vars
    tcpip      : TcpIpConnection;
    sslContext : JadeSSLContext;
    x509       : JadeX509Certificate;
begin
    create x509 transient;
    x509.readCertificateDataFromFile("c:\Certificates\client.pem");
    x509.readPrivateKeyDataFromFile("c:\Certificates\client.key",
                                    "myPassword");
    create sslContext transient;
    sslContext.methodType := JadeSSLContext.MethodTLSv1_2;
    sslContext.caFile := "c:\Certificates\serverCAcerts.pem";
    sslContext.x509 := x509;
    create tcpip transient;
    tcpip.name := "mySSLNode";
    tcpip.port := 8097;
    tcpip.sslContext := sslContext;
    tcpip.open;
    // ... send and receive some data
    tcpip.close;
epilog
    delete x509;
    delete sslContext;
    delete tcpip;
end;
```

See also the **JadeSSLContext** and **JadeX509Certificate** classes, earlier in this chapter.

userObject

Type: Object

The **userObject** property of the **TcpIpConnection** class contains a reference to an object that you can associate with any TCP/IP connection.

The default value is **null**.

verifyAuthResponseMethod

Type: String

The **verifyAuthResponseMethod** property of the **TcpIpConnection** class contains the name of the method in the authentication library used to verify the authentication response received after sending the authentication challenge.

If this property contains the name of a verify authentication response method, the specified method is executed after a successful generation of the authentication challenge and the successful receipt of the remote authentication response.

The code fragment in the following example shows the use of the **verifyAuthResponseMethod** property to set the authentication challenge and verification methods.

```
self.tcp.genAuthChallengeMethod   := "okGenAuthChallenge";
self.tcp.verifyAuthResponseMethod := "okVerifyAuthResponse";
```

TcpIpConnection Methods

The methods defined in the **TcpIpConnection** class are summarized in the following table.

Method	Description
close	Closes a connection to a remote application and returns when the connection is closed
closeAsynch	Closes a connection to a remote application and returns immediately
getMaxMessageSize	Returns the maximum message size that can be sent or received at one time
listen	Waits for a remote application to connect to its port and returns when the connection is established
listenAsynch	Waits for a remote application to connect to its port and returns immediately
listenContinuous	Waits for a remote application to connect to its port and returns the new connection on a new instance of the **TcpIpConnection** class while the original instance is still available for listening on subsequent calls
listenContinuousAsynch	Waits for remote applications to connect to its port and returns immediately
open	Establishes a connection to a remote application and returns when the connection is established
openAsynch	Establishes a connection to a remote application and returns immediately
readBinary	Reads binary data from the connection and returns when the specified number of bytes has been read or when a block of data is received
readBinaryAsynch	Reads binary data from the connection and returns immediately
readUntil	Reads data from the connection and returns when the specified delimiter is found in the data stream
readUntilAsynch	Reads data from the connection until the specified delimiter is found in the data stream and returns immediately
writeBinary	Writes binary data to the connection and returns when the operation is complete
writeBinaryAsynch	Writes binary data to the connection and returns immediately

close

Signature `close();`

The **close** method of the **TcpIpConnection** class closes a connection to a remote application and returns when the connection is closed. This method can be called when the connection is in any state. The following example shows the use of the **close** method to unload the form and close the connection if TCP/IP has been left in connection state.

```
buttonUnload_click(btn: Button input) updating;
begin
```

```
    // If a connection is present, closes the connection
    if self.tcp.state = Connection.Connected then
        self.tcp.close;
    endif;
    self.unloadForm;
end;
```

closeAsynch

Signature `closeAsynch(receiver: Object; msg: String);`

The **closeAsynch** method of the **TcpIpConnection** class closes a connection to a remote application and returns immediately. When the connection is closed, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter.

The **closeAsynch** method can be called when the connection is in any state.

Note On asynchronous calls, the state may not change immediately, and it may remain **Connected** (**2**) for a short period until JADE has rescheduled the request.

When the **closeAsynch** method completes, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **closeAsynch** method, as follows.

Signature `closeCallback(tcp: TcpIpConnection);`

The following example shows the use of the **closeAsynch** method to set the variable **conlog** to reference a **ConnectionLog** object, create the object, and initialize its properties if no such object exists.

```
closeAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Closes the current connection and returns immediately.  When
    // the connection is closed, the ConnectionLog object referenced
    // by conlog is called and told to run the method updateCloseCalls.
    self.tcp.closeAsynch(conlog, "updateCloseCalls");
    statusLine1.caption := "Disconnected";
    textBox2.text := "";
    textBox3.text := "";
    textBox4.text := "";
end;
```

getMaxMessageSize

Signature `getMaxMessageSize(): Integer;`

The **getMaxMessageSize** method of the **TcpIpConnection** class returns the maximum message size that can be sent or received at one time.

The result of this method is not defined until the connection has been opened. A value of zero (**0**) indicates that there is no upper limit to the allowable message size.

Note As this feature is not supported for the TCP/IP protocol, a value of zero (**0**) is always returned for a TCP/IP connection.

listen

Signature `listen();`

The **listen** method of the **TcpIpConnection** class waits for a remote application to connect to its port and returns when a connection attempt has been made.

The value of the **Connection** class **state** property changes to **Connecting** (**1**) when listening is in progress and to **Connected** (**2**) when the connection is open.

The code fragment in the following example shows the use of the **listen** method. This code sets the TCP/IP connection to listen to the current port. If a connection is made, it sets the status bar to read *Connected* and fills the text boxes with the IP address and name information.

```
self.tcp.port := 7895;
self.tcp.listen;
if self.tcp.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox3.text       := self.tcp.localIpAddress;
    textBox2.text       := self.tcp.remoteIpAddress;
    textBox4.text       := self.tcp.name;
endif;
...
```

See also the **Connection** class **timeout** property.

listenAsynch

Signature `listenAsynch(receiver: Object;`
`msg: String);`

The **listenAsynch** method of the **TcpIpConnection** class waits for a remote application to connect to its port and returns immediately.

When a connection attempt has been made by a remote application, the object specified in the **receiver** parameter is sent the message specified in the **msg** parameter.

The **listenAsynch** method can be called only when the value of the **Connection** class **state** property is **Disconnected** (**0**).

When this method is called, the value of the **state** property changes to **Connecting** (**1**). See also the **Connection** class **timeout** property.

Note On asynchronous calls, the state may not change immediately and it may remain **Disconnected** (**0**) for a short period until JADE has rescheduled the request.

The following example of the **listenAsynch** method sets the authentication challenge and verification methods for the connection.

```
listenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    self.tcp.genAuthChallengeMethod   := "okGenAuthChallenge";
    self.tcp.verifyAuthResponseMethod := "okVerifyAuthResponse";
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, the object is created and its properties
```

```
    // are initialized.
    conlog := ConnectionLog.firstInstance;
    if conlog = null then
        beginTransaction;
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        commitTransaction;
    endif;
    // Sets the tcp to listen on the current port and returns
    // immediately.  If a connection is made, the ConnectionLog object
    // referenced by conlog is called and told to run the updateListenCalls
    // method.
    self.tcp.port := 7895;
    self.tcp.listenAsynch(conlog, "updateListenCalls");
end;
```

The user-written callback method specified in the **msg** parameter is called when the **listenAsynch** method receives a connection request. The callback method must match the signature required by the **listenAsynch** method, as follows.

Signature `listenCallback(tcp: TcpIpConnection);`

The following method is an example of a **ConnectionLog** class callback method for the **listenAsynch** method, which updates the number of method invocations recorded for this method.

```
updateListenCalls(tcp: TcpIpConnection) updating;
begin
    beginTransaction;
    self.numberOfListenCalls := self.numberOfListenCalls + 1;
    commitTransaction;
end;
```

listenContinuous

Signature `listenContinuous(): TcpIpConnection;`

The **listenContinuous** method of the **TcpIpConnection** class waits for a remote application to connect to its port and returns a reference to the new connection on a new instance of the **TcpIpConnection** class while the original instance is still available for listening on subsequent calls.

The value of the **Connection** class **state** property changes to **Connecting** (**1**) when listening is in progress. See also the **Connection** class **timeout** property. The newly created instance of the **TcpIpConnection** class has its **state** property set to **Connected** (**2**) after the successful connection. The following example of the **listenContinuous** method sets the authentication challenge and verification methods for the connection.

```
listenContinuous_click(btn: Button input) updating;
begin
    self.tcp.genAuthChallengeMethod   := "okGenAuthChallenge";
    self.tcp.verifyAuthResponseMethod := "okVerifyAuthResponse";
    /* Sets the TCP to listen on the current port.  When a connection is
      made, a new instance of TcpIpConnection is returned and referenced
      by TCP.  The original instance remains available for listening on
```

```
      subsequent calls while the new instance maintains the newly made
      connection.  When this connection is made, the status bar is set to
      read 'connected', and the text boxes filled with the IP address
      and name information. */
    self.tcp.port := 7895;
    self.tcp      := tcp.listenContinuous;
    if self.tcp.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox3.text       := self.tcp2.localIpAddress;
        textBox2.text       := self.tcp2.remoteIpAddress;
        textBox4.text       := self.tcp2.name;
    endif;
end;
```

listenContinuousAsynch

Signature `listenContinuousAsynch(receiver: Object; msg: String);`

The **listenContinuousAsynch** method of the **TcpIpConnection** class waits for remote applications to connect to its port and returns immediately.

When a connection attempt has been made by a remote application, the object specified in the **receiver** parameter is sent the message specified in the **msg** parameter.

The **listenContinuousAsynch** method can be called only when the value of the **Connection** class **state** is **Disconnected** (**0**).

When this method is called, the value of the **state** property changes to **Connecting** (**1**). See also the **Connection** class **timeout** property.

Note On asynchronous calls, the state may not change immediately and it may remain **Disconnected** (**0**) for a short period until JADE has rescheduled the request.

The following example of the **listenContinuousAsynch** method sets the authentication challenge and verification methods for the connection.

```
listenContAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    self.tcp.genAuthChallengeMethod   := "okGenAuthChallenge";
    self.tcp.verifyAuthResponseMethod := "okVerifyAuthResponse";
    /*  Sets the TCP to listen on the current port.  When a connection is
    made, a new instance of TcpIpConnection is created.  The original
    instance remains available for listening on subsequent calls while the
    new instance maintains the newly made connection.  When this connection
    is made, the ConnectionLog object referenced by conlog is called and
    told to run the updateListenContinuousCalls method.  The new
    TcpIpConnection instance is passed to this method as a parameter. */
    self.tcp.port := 7895;
    self.tcp.listenContinuousAsynch(conlog, "updateListenContinuousCalls");
    if self.tcp.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox3.text       := self.tcp2.localIpAddress;
        textBox2.text       := self.tcp2.remoteIpAddress;
```

```
            textBox4.text       := self.tcp2.name;
        endif;
    end;
```

The user-written callback method specified in the **msg** parameter is called when the **listenContinuousAsynch** method receives a connection request.

The callback method must match the signature required by the **listenContinuousAsynch** method, as follows.

Signature `listenContinuousCallback(tcp: TcpIpConnection; newTcp: TcpIpConnection);`

The following method is an example of **ConnectionLog** class callback method for the **listenContinuousAsynch** method, which updates the number of method invocations recorded for this method.

```
updateListenContinuousCalls(tcp:    TcpIpConnection;
                            newTcp: TcpIpConnection) updating;
begin
    beginTransaction;
    self.numberOfListenContinuousCalls := self.numberOfListenContinuousCalls
                    + 1;
    commitTransaction;
    self.newTcp.readBinaryAsynch(1024, newTcp, "readCallback");
end;
```

The **listenContinuousAsynch** method continues accepting new connection requests until the listener **TcpIpConnection** class instance is closed.

The **listenContinuousCallback** method is called for every successful connection request.

open

Signature `open();`

The **open** method of the **TcpIpConnection** class establishes a connection to a remote application and returns when the connection is established.

Use the **name** property of the **Connection** class to define the name or the IP address of the TCP/IP target host used when opening a TCP/IP connection using the **open** or **openAsynch** method. The **name** property, if not an IP address, must be specified in the local HOSTS file or be defined on the Domain Name Service (DNS) node before executing the **open** or **openAsynch** method.

Use the **port** property to define the target service port when using the **open** or **openAsynch** method or to define the listen port when using the **listen** or **listenAsynch** method.

The **open** method can be called only when the value of the **Connection** class **state** property is **Disconnected** (**0**).

The value of the **Connection** class **state** property changes to **Connected** (**2**) when the connection is open.

The code fragment in the following example shows the use of the **open** method.

```
if bOpen.value = true then
    self.tcp.open;
elseif bListen.value = true then
    statusLine1.caption := "Listening";
    self.tcp.listen;
else
```

```
    self.tcp.close;
endif;
```

openAsynch

Signature `openAsynch(receiver: Object; msg: String);`

The **openAsynch** method of the **TcpIpConnection** class establishes a connection to a remote application and returns immediately. When the connection is established, the object specified in the **receiver** parameter is sent the message specified in the **msg** parameter.

Use the **name** property of the **Connection** class to define the name or the IP address of the TCP/IP target host used when opening a TCP/IP connection using the **open** or **openAsynch** method. The **name** property, if not an IP address, must be specified in the local HOSTS file or be defined on the Domain Name Service (DNS) node before executing the **open** or **openAsynch** method.

Use the **port** property to define the target service port when using the **open** or **openAsynch** method or to define the listen port when using the **listen** or **listenAsynch** method.

The **openAsynch** method can be called only when the value of the **Connection** class **state** property is **Disconnected** (**0**). When this method is called, the value of the **state** property changes to **Connecting** (**1**).

Note On asynchronous calls, the state may not change immediately and it may remain **Disconnected** (**0**) for a short period until JADE has rescheduled the request.

The following example of the **openAsynch** method sets the authentication response for the connection.

```
buttonOpenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    self.tcp.genAuthResponseMethod := "okGenAuthResponse";
    // Attempts to connect to the current port and returns immediately.
    // If a connection is made, the ConnectionLog object referenced by
    // conlog is called and told to run the updateOpenCalls method.
    self.tcp.openAsynch(conlog, "updateOpenCalls");
end;
```

When the **openAsynch** method establishes a connection, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **openAsynch** method, as follows.

Signature `openCallback(tcp: TcpIpConnection);`

The following method is an example of **ConnectionLog** class callback method for the **openAsynch** method, which updates the number of method invocations recorded for this method.

```
updateOpenCalls(tcp: TcpIpConnection) updating;
begin
    beginTransaction;
    self.numberOfOpenCalls := self.numberOfOpenCalls + 1;
    commitTransaction;
    self.tcp.readBinaryAsynch(1024, tcp, "readCallback");
end;
```

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **TcpIpConnection** class reads binary data from the connection and returns when the number of bytes of data specified in the **length** parameter have been read or when a block of data is received, depending on the setting of the **Connection** class **fillReadBuffer** property.

This method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property.

Only one synchronous or asynchronous read operation can be performed at one time on a connection.

Note When executing the **readBinary** notification method, ensure that all received data has been handled, copied, or stored before issuing another **readBinaryAsynch** method.

If the **readBinary** notification method executes another **readBinaryAsynch** method, it overwrites the data that was previously received if data is readily available on the connection.

The following example of the **readBinary** method sets the decryption method for the connection.

```
buttonReceive_click(btn: Button input) updating;
begin
    self.tcp.decryptMethod := "okDecrypt";
    // Checks to make sure TCP is in connected state (2).  If it is
    // and binary data is received through the connection, displays
    // the data in the text box.  The parameter of 50 specifies that
    // the data must be no more than 50 bytes long.
    if self.tcp.state = Connection.Connected then
        textBox1.text := self.tcp.readBinary(50).String;
    endif;
end;
```

See also the **readBinaryAsynch** method.

readBinaryAsynch

Signature
```
readBinaryAsynch(length:   Integer;
                 receiver: Object;
                 msg:      String);
```

The **readBinaryAsynch** method of the **TcpIpConnection** class reads binary data from the connection and returns immediately. When the bytes of data specified in the **length** parameter have been read or when a block of data is received, depending on the setting of the **Connection** class **fillReadBuffer** property, the object specified in the **receiver** parameter is sent the message specified in the **msg** parameter.

Only one synchronous or asynchronous read operation can be performed at one time on a connection.

The **readBinaryAsynch** method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property.

When the bytes of data specified in the **length** parameter have been read or when a block of data is received, the user-written callback method specified in the **msg** parameter is called. The value of the **length** parameter must be greater than zero (**0**).

The following example of the **readBinaryAsynch** method sets the decryption method for the connection.

```
receiveAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    self.tcp.decryptMethod := "okDecrypt";
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, it is created and its properties initialized.
    if self.tcp.state = Connection.Connected then
    // Reads binary data from the connection and returns immediately.
    // When the data is read, the ConnectionLog object referenced by
    // conlog is called and told to run the updateBinaryReads method.
    // It is passed a parameter containing the binary data that was
    // read from the connection.
       self.tcp.readBinaryAsynch(50, conlog, "updateBinaryReads");
    endif;
end;
```

The callback method must match the signature required by the calling **readBinaryAsynch** method, as follows.

Signature
```
readBinaryCallback(tcp:    TcpIpConnection;
                   buffer: Binary);
```

The following method is an example of **ConnectionLog** class callback method for the **readBinaryAsynch** method, which updates the number of method invocations recorded for this method.

```
updateBinaryReads(tcp:    TcpIpConnection;
                  buffer: Binary) updating;
begin
    beginTransaction;
    self.obj.data := buffer;
    commitTransaction;
    self.tcp.readBinaryAsynch(1024, self.tcp, "readCallback");
end;
```

readUntil

Signature
```
readUntil(delimiter: Binary;
          maxLength: Integer): Binary;
```

The **readUntil** method of the **TcpIpConnection** class reads binary data from the connection and returns when the delimiter specified in the **delimiter** parameter is found in the data stream. Use this method if you use delimiters as an end-of-message mechanism as part of your communications protocol so that you do not have to read a character at a time and scan or handle your own data buffering.

You can use the **maxLength** parameter to specify a maximum read size if the specified delimiter cannot be found. (A value of zero (**0**) indicates that there is no maximum read size.)

This method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property.

Only one synchronous or asynchronous read operation can be performed at one time on a connection.

Notes The delimiter is not included in the returned data.

A **String** value typecast to a **Binary** value and specified as a delimiter in a Unicode JADE system contains Unicode characters in the **Binary** value.

readUntilAsynch

Signature
```
readUntilAsynch(delimiter: Binary;
                maxLength: Integer;
                receiver:  Object;
                msg:       String);
```

The **readUntilAsynch** method of the **TcpIpConnection** class reads binary data from the connection, and returns immediately. Use this method if you use delimiters as an end-of-message mechanism as part of your communications protocol so that you do not have to read a character at a time and scan or handle your own data buffering.

When the delimiter specified in the **delimiter** parameter has been read, the object specified in the **receiver** parameter is sent the message specified in the **msg** parameter.

You can use the **maxLength** parameter to specify a maximum read size if the specified delimiter cannot be found. A value of zero (**0**) indicates that there is no maximum read size.

A **String** value typecast to a **Binary** value and specified as a delimiter in a Unicode JADE system contains Unicode characters in the **Binary** value.

When executing the **readUntilAsynch** notification method, ensure that all received data has been handled, copied, or stored before issuing another **readUntilAsynch** method. If the **readUntilAsynch** notification method executes another **readUntilAsynch** method, it overwrites the data that was previously received if data is readily available on the connection.

Only one synchronous or asynchronous read operation can be performed at one time on a connection.

The **readUntilAsynch** method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property. When the delimiter specified in the **delimiter** parameter has been read, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **readUntilAsynch** method, as follows.

Signature
```
readUntilNotify(tcp: TcpIpConnection;
                bin: Binary);
```

writeBinary

Signature
```
writeBinary(buffer: Binary);
```

The **writeBinary** method of the **TcpIpConnection** class writes binary data to the connection and returns when the operation is complete.

The **writeBinary** method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property.

Messages are sent in the order that the connection object receives them.

The following example of the **writeBinary** method sets the **encryptMethod** property for the connection.

```
buttonSend_click(btn: Button input) updating;
begin
```

```
    self.tcp.encryptMethod := "okEncrypt";
    // Checks to make sure TCP is in connection state 2 (connected).
    // If it is, binary data from the text box is written to the connection.
    if self.tcp.state = Connection.Connected then
        self.tcp.writeBinary(textBox1.text.Binary);
    endif;
end;
```

writeBinaryAsynch

Signature `writeBinaryAsynch(buffer: Binary; receiver: Object; msg: String);`

The **writeBinaryAsynch** method of the **TcpIpConnection** class writes binary data to the connection and returns immediately.

When the operation is complete, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter.

User-written methods specified in the **msg** parameter are sent in the order that they are received by the connection object.

Multiple asynchronous write operations can be performed against one connection simultaneously.

The **writeBinaryAsynch** method can be called only when the value of the **Connection** class **state** property is **Connected** (**2**). See also the **Connection** class **timeout** property.

When the write operation has been completed, the user-written callback method specified in the **msg** parameter is called.

The following example shows the use of the **writeBinaryAsynch** method to set the **encryptMethod** property for the connection.

```
buttonSendAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    tcp.encryptMethod := "okEncrypt";
        // Outputs the binary data from the text box to the connection
        // and returns immediately.  When the data is written, the
        // ConnectionLog object referenced by conlog is called and
        // told to run the updateBinaryWrites method.
        tcp.writeBinaryAsynch(textBox1.text.Binary,c,"updateBinaryWrites");
    endif;
end;
```

The callback method must match the signature required by the calling **writeBinaryAsynch** method, as follows.

Signature `writeBinaryCallback(tcp: TcpIpConnection);`

The following method is an example of a **ConnectionLog** class callback method for the **writeBinaryAsynch** method, which updates the number of method invocations recorded for this method.

```
updateBinaryWrites(tcp: TcpIpConnection) updating;
begin
    tcp.readBinaryAsynch(1024, tcp, "readCallback");
end;
```

TimeArray Class

The **TimeArray** class is an ordered collection of **Time** values in which the values are referenced by their position in the collection.

Time arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a Time array.

Inherits From: Array

Inherited By: (None)

TimeFormat Class

The **TimeFormat** class is used to store Windows locale time information.

You cannot modify system-created instances of the **TimeFormat** class (that is, instances created and maintained by JADE to store locale information and user-defined formats) from your JADE code.

JADE automatically creates a transient instance of **TimeFormat** for each application, which you can read by using **app.currentLocaleInfo.timeInfo**. This instance contains time information for the current locale.

TimeFormat instances are also used to store user-defined time formats that can be passed to the various primitive type user format methods. You can maintain these formats only by using the appropriate Formats menu command, accessed from the Format Browser.

For details about returning a string containing the receiver in the supplied time format, see the **Time** primitive type **userFormat** method and for details about the properties and method defined in the **TimeFormat** class, see "TimeFormat Properties" and "TimeFormat Method", in the following subsections.

Inherits From: LocaleFormat

Inherited By: (None)

TimeFormat Properties

The properties defined in the **TimeFormat** class are summarized in the following table.

Property	Description
amText	Contains the text for the morning time marker
ampmIsSuffix	Specifies whether the time marker is displayed before or after the time string
format	Contains the time formatting string
is12HourFormat	Specifies whether the 12-hour or 24-hour time format is used
pmText	Contains the text for the afternoon time marker
separator	Contains the string value used to separate hours, minutes, and seconds
showLeadingZeros	Specifies whether a leading zero is displayed in time fields less than ten
showSeconds	Specifies whether seconds are displayed in the time format

amText

Type: String[100]

The **amText** property of the **TimeFormat** class contains the text for the pre-noon time marker in the time format; for example, **"AM"**.

ampmIsSuffix

Type: Boolean

The **ampmIsSuffix** property of the **TimeFormat** class is set to **true** if the time marker string (AM or PM designator) follows the time string; for example, **"9:15 AM"**.

This property is set to **false** if the time marker precedes the time string; for example, **"AM 9:15"**.

format

Type: String[127]

The **format** property of the **TimeFormat** class contains the time formatting string. (For details, see the **Time** primitive type **format** method, in Chapter 1 of the *JADE Encyclopaedia of Primitive Types*.)

is12HourFormat

Type: Boolean

The **is12HourFormat** property of the **TimeFormat** class is set to **true** if a 12-hour time format is used; for example, **1:15:43 PM**.

This property is set to **false** if a 24-hour time format is used; for example, **13:15:43**.

pmText

Type: String[100]

The **pmText** property of the **TimeFormat** class contains the text for the post-noon time marker in the time format; for example, **"PM"**.

separator

Type: String[10]

The **separator** property of the **TimeFormat** class contains the character used to separate hours, minutes, and seconds; for example, **"/"**.

showLeadingZeros

Type: Boolean

The **showLeadingZeros** property of the **TimeFormat** class is set to **true** if a leading zero is displayed in time fields less than 10; for example, **08:45**.

showSeconds

Type: Boolean

The **showSeconds** property of the **TimeFormat** class is set to **true** if seconds are displayed in the time format; for example, **8:45:39 AM**.

TimeFormat Method

The method defined in the **TimeFormat** class is summarized in the following table.

Property	Description
defineTimeFormat	Defines the characteristics of a time format

defineTimeFormat

Signature

```
defineTimeFormat(showAs12HourClock:   Boolean;
                 showLeadingZero:     Boolean;
                 showAmPmSuffix:      Boolean;
                 textAm:              String;
                 textPm:              String;
                 hourMinSecSeparator: String;
                 showSecs:            Boolean) updating;
```

The **defineTimeFormat** method of the **TimeFormat** class enables you to dynamically define the characteristics of a time format. (For details about returning a string containing the receiver in the supplied time format, see the **Time** primitive type **userFormat** method.)

Set the **showAs12HourClock** parameter to **true** if you want to display a 12-hour time. Alternatively, set this parameter to **false** if you want to display a 24-hour time.

Set the **showLeadingZero** parameter to **true** if you want to display a leading zero (**0**) for hours less than **10**. Alternatively, set this parameter to **false** if you do not want to display a leading zero.

Set the **showAmPmSuffix** parameter to **true** if you want to display a time marker (for example, **am** or **pm**) after the time in 12-hour format. Alternatively, set this parameter to **false** if you do not want to display a time marker after the time (for example, when the time format is specified as 24-hour format).

The **textAm** and **textPm** parameters enable you to specify a string of up to 30 characters that is to be displayed for times before midday and after midday, respectively. If the strings are longer than 30 characters, they are truncated to 30 characters.

Use the **hourMinSecSeparator** parameter to specify a string of up to 10 characters that contains the text to be displayed between the hours and seconds. If the string contains any of the **d**, **M**, **y**, **g**, **h**, **H**, **m**, **s**, or **t** characters, these characters are removed. If the string is longer than 10 characters, it is truncated to 10 characters.

Set the **showSecs** parameter to **true** if you want to display a seconds. Alternatively, set this parameter to **false** if you do not want to display seconds.

TimeStampArray Class

The **TimeStampArray** class is an ordered collection of **TimeStamp** values in which the values are referenced by their position in the collection.

TimeStamp arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a TimeStamp array.

Inherits From: Array

Inherited By: (None)

TimeStampIntervalArray Class

The **TimeStampIntervalArray** class is an ordered collection of **TimeStampInterval** values in which the values are referenced by their position in the collection.

TimeStampInterval arrays inherit the methods defined in the **Array** class.

The bracket (**[ ]**) subscript operators enable you to assign values to and receive values from a **TimeStampInterval** array.

Inherits From: Array

Inherited By: (None)

TranslatableString Class

The **TranslatableString** class is a named text entity that enables different text string values to be displayed in an application depending on the locale of the client machine. Instead of hard coding a string literal value in your method source or for the caption of a label, you can define and use a translatable string.

When you define a translatable string with a specified name, separate translatable strings are created for each declared locale.

Each locale-specific translatable string can contain a different text string. For example, you can add a translatable with the name **Hello** and provide the following text string values for the France and New Zealand versions of the **Hello** translatable string.

```
Hello = "Bonjour" // for locale 1036 (French - France)
Hello = "Gidday"  // for locale 5129 (English - New Zealand)
```

In the JADE Editor and the JADE Painter, a translatable string is always prepended with a dollar sign ($), as shown in the following **write** instruction.

```
write $Hello;
```

At runtime, the output from the **write** instruction would depend on the locale of the client machine. In France, "Bonjour" would be output and in New Zealand "Gidday" would be output.

Translatable strings can be used in method source code and in the JADE Painter; for example, you could use a translatable string for the caption on a **Label** control. For details, see "Translating Control Properties", in Chapter 5 of the *JADE Development Environment User's Guide*.

For details about working with translatable strings, see "Adding a New Translatable String", "Updating an Existing Translatable String", "Extract Translatable Strings Method Example", and "Load Translatable Strings Method Example", in Chapter 11 of the *JADE Development Environment User's Guide*.

The **Locale** class provides the **getTranslatableStringLocal**, **getTranslatableStrings**, and **getTranslatableStringsByNum** methods for retrieving translatable strings defined for a locale.

For details about the properties and method defined in the **TranslatableString** class, see "TranslatableString Properties" and "TranslatableString Method", in the following subsections.

Inherits From: Constant

Inherited By: (None)

TranslatableString Properties

The properties defined in the **TranslatableString** class are summarized in the following table.

Property	Contains a reference to the …
formBuildDataRefs	Set of forms that contain the translatable string
locale	Locale of the translatable string

The **TranslatableString** class inherits the properties summarized in the following table from the **Constant** superclass, which is an undocumented metaschema class.

Property	Contains…
constantRefs	A reference to a set of translatable strings that use (reference) the translatable string
constantUsages	A reference to a collection of embedded usages of the translatable string

formBuildDataRefs

Type: FormSet

The read-only **formBuildDataRefs** property of the **TranslatableString** class contains a reference to the set of forms that contain the translatable string.

The control properties that can utilize translatable strings are listed in the following table.

Property	Control Type in which the Property Value Can be Translated
bubbleHelp	Window
caption	Button, CheckBox, JadeDockBase, Form, Frame, GroupBox, JadeMask, Label, MenuItem, OptionButton, Sheet, StatusLine
helpKeyword	MenuItem, Window
mask	JadeEditMask
text	JadeEditMask, TextBox

For more details, see "Translating Control Properties", in Chapter 5 of the *JADE Development Environment User's Guide*.

locale

Type: Locale

The read-only **locale** property of the **TranslatableString** class contains a reference to the locale to which the translatable string belongs.

TranslatableString Method

The method defined in the **TranslatableString** class is summarized in the following table.

Method	Description
updateCompile	Updates the existing translatable string

updateCompile

Signature

```
updateCompile(source:      String;
              errorCode:   Integer output;
              errorOffset: Integer output;
              errorLength: Integer output): Boolean updating;
```

The **updateCompile** method of the **TranslatableString** class compiles and updates the existing translatable string. If the compilation fails, the method returns **true**, the translatable string is not updated and the current transaction is aborted.

The **updateCompile** method parameters are listed in the following table.

Parameter	Description
source	The new source for the translatable string.
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the translatable string compiled successfully.
errorOffset	The position of the error in the translatable string. Note that the first character of the translatable string has a position of zero (**0**).
errorLength	The length in characters of the error in the translatable string.

Note To add and compile a new translatable string to all base locales of the receiving schema, use the **addCompileTranslatableString** method of the **Schema** class.

Type Class

The **Type** class is the abstract superclass of all class, primitive type, and JADE interface meta classes.

For details about the properties and methods defined in the **Type** class, see "Type Properties" and "Type Methods", in the following subsections.

Inherits From: SchemaEntity

Inherited By: Class, JadeInterface, PrimType

Type Properties

The properties defined in the **Type** class are summarized in the following table.

Property	Contains the …
consts	Dictionary of constants in the type
methods	Dictionary of methods in the type
schema	Schema in which the class or primitive type is defined
superschemaType	Type of the superschema class

consts

Type: ConstantNDict

Availability: Protected

The **consts** property of the **Type** class contains a reference to the dictionary of constants in the type.

methods

Type: MethodNDict

Availability: Protected

The **methods** property of the **Type** class contains a reference to the dictionary of methods in the type.

schema

Type: Schema

Availability: Read-only

The **schema** property of the **Type** class contains a reference to the schema in which the class, primitive type, or interface is defined.

superschemaType

Type: Type

Availability: Read-only

The **superschemaType** property of the **Type** class contains a reference to the type of the superschema.

Type Methods

The methods defined in the **Type** class are summarized in the following table.

Method	Description
allMethods	Populates a method set with all methods of the receiver
findConstant	Returns the specified constant from the current schema
findConstantInSuperschema	Returns the specified constant from the superschema
findProperty	Returns the specified property
getConstant	Returns the constant with the specified name
getConstants	Adds the constants in the type to the specified dictionary
getConstantsInSchema	Adds the constants in the specified schema to the specified dictionary
getMethod	Returns the method with the specified name
getMethods	Adds the methods to the specified dictionary
getName	Returns a string containing the name of the class or primitive type
getProperty	Returns the property with the specified name
inheritsFrom	Returns **true** if the class inherits methods and properties from the specified class
instancesExist	Returns **true** if instances exist of the class or its subclasses

allMethods

Signature `allMethods(methSet: MethodSet io);`

The **allMethods** method of the **Type** class populates the method set specified in the **methSet** parameter with a reference to all methods in the receiver.

findConstant

Signature `findConstant(str: String): Constant;`

The **findConstant** method of the **Type** class returns a reference to the constant specified in the **str** parameter.

findConstantInSuperschema

Signature `findConstantInSuperschema(constantName: String): Constant;`

The **findConstantInSuperschema** method of the **Type** class returns a reference to the constant specified in the **constantName** parameter from the superschema.

findProperty

Signature `findProperty(str: String): Property;`

The **findProperty** method of the **Type** class returns a reference to the property specified in the **str** parameter in the type of the receiver or a super-type.

getConstant

Signature `getConstant(name: String): Constant;`

The **getConstant** method of the **Type** class returns a reference to the constant specified in the **name** parameter.

getConstants

Signature `getConstants(constDict: ConstantNDict input);`

The **getConstants** method of the **Type** class adds references to the constants in the type to the constants dictionary specified in the **constDict** parameter.

The dictionary is not cleared before instances are added.

The following example shows the use of the **getConstants** method.

```
vars
    dict : ConstantNDict;
    con  : Constant;
begin
    create dict transient;
    getConstants(dict);
    foreach con in dict do
        ...                    // do some processing here
    endforeach;
epilog
    delete dict;
end;
```

getConstantsInSchema

Signature
```
getConstantsInSchema(topSchema: Schema;
                     constDict: ConstantNDict input);
```

The **getConstantsInSchema** method of the **Type** class adds references to the constants in the schema specified in the **topSchema** parameter to the constants dictionary specified in the **constDict** parameter.

The dictionary is not cleared before instances are added.

getMethod

Signature `getMethod(name: String): Method;`

The **getMethod** method of the **Type** class returns a reference to the method specified in the **name** parameter; for example:

```
meth := Fault.getMethod("getDaysOpen");
```

getMethods

Signature `getMethods(meths: MethodNDict input);`

The **getMethods** method of the **Type** class adds references to the methods in the type to the methods dictionary specified in the **meths** parameter.

The dictionary is not cleared before instances are added.

getName

Signature `getName(): String;`

The **getName** method of the **Type** class returns a string containing the name of the class or primitive type.

getProperty

Signature `getProperty(propName: String): Property;`

The **getProperty** method of the **Type** class returns a reference to the property specified in the **propName** parameter.

Use the **findProperty** method if you want to find the property in the type of the receiver or any of its supertypes.

inheritsFrom

Signature `inheritsFrom(type: Type): Boolean;`

The **inheritsFrom** method of the **Type** class returns **true** if the receiver inherits methods and properties from the type specified in the **type** parameter; for example:

```
if cls.inheritsFrom(MyNewDialog) then
    if form.borderStyle = BorderStyle_Sizable then
        write form.name;
    endif;
endif;
```

A type always inherits from itself.

instancesExist

Signature `instancesExist(): Boolean;`

The **instancesExist** method of the **Type** class returns **true** if instances exist of the class or its subclasses.

UserInterfaceException Class

The **UserInterfaceException** class is the transient class that defines behavior for exceptions relating to the handling of windows.

Inherits From: NormalException

Inherited By: ActiveXInvokeException, JadeDotNetInvokeException

WebSession Class

The **WebSession** class maintains all Internet session information. Each JADE application maintains its own copy of the **WebSession** object.

Note The term Web server refers to Microsoft Internet Information Server (IIS) or Apache HyperText Transfer Protocol (HTTP) Server.

An exception is raised if you use transient transactions in the **create** event of a **WebSession** subclass.

For details about the constant, properties, and methods defined in the **WebSession** class, see "WebSession Class Constant", "WebSession Properties", and "WebSession Methods", in the following subsections.

Inherits From: Object

Inherited By: RootSchemaSession

WebSession Class Constant

The constant provided by the **WebSession** class is listed in the following table.

Constant	Integer Value
WebSession_System_Timer_Event	18041999

For details, see the **timerEvent** method.

WebSession Properties

The properties defined in the **WebSession** class are summarized in the following table.

Property	Descriptions
lastAccessTime	Contains the timestamp of the last access of the JADE schema in the session
sessionId	Contains the unique random number identifier of the session
startTime	Contains the timestamp of the time that the session was started
usePageSequencing	Specifies whether the forms generated for Web-enabled applications have a hidden sequence number field

lastAccessTime

Type: TimeStamp

The **lastAccessTime** property of the **WebSession** class contains the timestamp of the last access of the JADE schema in the Web session.

The **lastAccessTime** property is used to determine the disconnect status. The Web session is terminated if there is no activity for the session for a specified time.

sessionId

Type: Integer

The **sessionId** property of the **WebSession** class contains the unique random number identifier of the Web session. This identifier is encrypted and stored as a hexadecimal string in a hidden field.

The **sessionId** property and the creation timestamp of the Web page are embedded as hidden text fields in every Web page sent to the client.

startTime

Type: TimeStamp

The **startTime** property of the **WebSession** class contains the timestamp of the time that the Web session was started.

usePageSequencing

Type: Boolean

The **usePageSequencing** property of the **WebSession** class specifies whether the forms that are generated for JADE Forms or HTML Documents Web-enabled applications have a hidden field with a sequence number that is incremented with each request.

When a response is received from a browser, the sequence number is compared to the one stored in the Web session. If the incoming number is less than the number on the Web session, an exception is raised (11091 Submitted HTML form is out of sequence).

To enforce correct sequencing of HTML pages, set the **page_sequencing** or **html_page_sequencing** element in the XML application configuration file to **true**, or add the following line to the **create** method of the **WebSession** subclass.

```
usePageSequencing := true;
```

WebSession Methods

The methods defined in the **WebSession** class are summarized in the following table.

Method	Description
browserType	Returns a string containing the type of Web browser.
createVirtualDirectoryFile	Passes image files created by a JADE application to the **jadehttp** library or JADE **mod_jadehttp** module.
deleteVirtualDirectoryFile	Deletes specified files from the virtual directory used by the **jadehttp** library.
getCurrentLocale	Returns the locale based on information returned from the browser.
getHttpParam	Returns the value associated with the specified HTTP parameter.
getHttpString	Returns the HTTP string that is returned from the Web browser.
getServerVariable	Returns the HyperText Transfer Protocol (HTTP) header information for your Web request from the Web server (that is, Microsoft IIS Server or Apache HTTP Server).

Method	Description
getSessionForm	Keeps track of all open instances of the specified form for the current session.
getWebSessionCount	Returns the total number of active Web sessions for all nodes connected to the server.
isVDFilePresent	Returns whether the requested file is present on the Web server side of the firewall when using the JADE Web interface via the **jadehttp** library file or the JADE **mod_jadehttp** module.
processRequest	Executed when a request is received from the Web.
removeSession	Removes the current Web session.
removeSessionWithMessage	Removes the current Web session and sends the specified message.
reply	Executed when all processing is complete and the JADE system is ready to send a response back to the Web browser.
setCurrentLocale	Switches locales from the locale of the requesting Web browser or default locale.
timerEvent	Invoked by the Web session when the session times out.

browserType

Signature `browserType(): String;`

The **browserType** method of the **WebSession** class returns a string containing the type of Web browser; that is, **Netscape** or **Internet-Explorer**.

An exception is raised if this method is invoked from a server method.

createVirtualDirectoryFile

Signature
```
createVirtualDirectoryFile(fileName:     String;
                           fileContents: Binary;
                           retain:       Boolean): Integer;
```

The **createVirtualDirectoryFile** method of the **WebSession** class, which passes image files created by a JADE application to the **jadehttp** library or the **mod_jadehttp** module, can be reimplemented in your user session class.

The **jadehttp** library or **mod_jadehttp** module creates the specified file in the directory (the virtual directory visible to Web browsers) in which the library is running. (See also the **WebSession** class **isVDFilePresent** method.)

The **createVirtualDirectoryFile** method parameters are listed in the following table.

Parameter	Description
fileName	Name of the file to be created in the virtual directory
fileContents	Binary holding the file contents
retain	Creates read-only files when set to **true** or standard files when set to **false**

This method returns zero (**0**) if the method successfully formats a request to the **jadehttp** library or **mod_jadehttp** module, or it returns the non-zero Windows error code indicating the failure to create the file.

The image files must be passed before the final reply to the Web request is returned.

This process is transparent if your application is using the standard JADE-generated Internet facility. However, if your application logic does additional file generation of its own, you must call this method.

You can specify whether files created in the virtual directory are deleted automatically and how this happens by setting the **PurgeDirectoryRule** parameter in the [*application-name*] section of the **jadehttp.ini** file or the **PurgeDirectoryRule** configuration directive in the JADE **mod_jadehttp**. If this parameter or directive is not set, files of type .**jpg**, .**png**, or .**gif** that are more than 12 hours old are removed. For details, see "Internal Housekeeping of the Virtual Directory", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Notes If your applications are not using the standard JADE-generated Internet facility, you need to set the JADE initialization file **Firewall** parameter in the [Jadehttp Files] section to **true** and call the **createVirtualDirectoryFile** method *only* if you require firewall separation. (For details, see "Configuring JadeHttp for Remote Connections", in Chapter 2 of the *JADE Installation and Configuration Guide*.) If you do not require firewall separation, JADE creates image files directly into the virtual directory and bypasses the **jadehttp** library or **mod_jadehttp** module.

The file cleanup process that is started when the JADE initialization file **Firewall** parameter is set to **true** deletes only files that are *not* read-only and which are of type .**jpg** .**png**, or .**gif**. You should therefore make all other files in this directory that you want to retain read-only, by setting the **retain** parameter to **true**.

deleteVirtualDirectoryFile

Signature

```
deleteVirtualDirectoryFile(filename:         String;
                           deleteIfReadOnly: Boolean): Integer;
```

The **deleteVirtualDirectoryFile** method of the **WebSession** class enables you to delete files that are in the directory specified by the **VirtualDirectory** parameter in the **jadehttp.ini** file.

The **deleteVirtualDirectoryFile** method parameters are listed in the following table.

Parameter	Description
filename	Name of the file to be deleted from the virtual directory
deleteIfReadOnly	Deletes files marked as read-only when set to **true**

This method returns zero (**0**) if the file deletion is successful or it returns a non-zero error code if the deletion fails.

You can specify whether files created in the virtual directory are deleted automatically and how this happens by setting the **PurgeDirectoryRule** parameter in the [*application-name*] section of the **jadehttp.ini** file or the **PurgeDirectoryRule** configuration directive in the JADE **mod_jadehttp**. If this parameter or directive is not set, files of type .**jpg**, .**png**, or .**gif** that are more than 12 hours old are removed. For details, see "Internal Housekeeping of the Virtual Directory", in Chapter 2 of the *JADE Installation and Configuration Guide*.

getCurrentLocale

Signature

```
getCurrentLocale(): Locale;
```

The **getCurrentLocale** method of the **WebSession** class returns the locale of the session object.

The locale is set from information returned by the browser when the session object is created. If the locale of the browser is not defined in the JADE system, then the default locale of the current schema is used. You can override the locale programmatically using the **setCurrentLocale** method.

getHttpParam

Signature `getHttpParam(paramName: String): String;`

The **getHttpParam** method of the **WebSession** class, which returns the value associated with the HyperText Transfer Protocol (HTTP) parameter specified in the **paramName** parameter, can be reimplemented in your user session class.

The HTTP string that is returned from the Web browser generally constitutes *name-value* pairs. The *name-value* pairs are separated by an ampersand character (**&**) and within this, the name and value are separated by an equals symbol (**=**).

Use this method to get the value portion of a *name-value* pair, as shown in the following example.

```
getUserName(): String;
vars
    userName: String;
begin
    //look for a field name called userName
    userName := currentSession.getHttpParam('userName');
    return userName;
end;
```

Note The **paramName** parameter is case-sensitive.

Using the **http://www.jadeworld.com/jadehttp.dll?TestApp&myparam=KiaOra** HTTP string, the code fragment in the following example shows the use of the **getHttpParam** method to return the specified value.

```
str := currentSession.getHttpParam("myparam");
write str;           // Outputs "KiaOra"
```

If the specified parameter does not exist, a null string (**""**) is returned.

getHttpString

Signature `getHttpString(): String;`

The **getHttpString** method of the **WebSession** class, which returns a string containing the HyperText Transfer Protocol (HTTP) string returned from the Web browser, can be reimplemented in your user session class.

getServerVariable

Signature `getServerVariable(var: String): String;`

The **getServerVariable** method of the **WebSession** class returns the HTTP header information for your World Wide Web request from your Web server.

As the **var** parameter depends on the Web server version and can change, refer to your Web server documentation for details.

The code fragment in the following example returns the IP address of the current Web session.

```
currentSession.getServerVariable('REMOTE_ADDR');
```

Common server environment variables, documented in the IIS documentation under the **ServerVariables** function, include those listed in the following table.

Variable	Returns…
HTTP_ACCEPT_LANGUAGE	A string describing the language to use for displaying content
HTTP_USER_AGENT	A string describing the browser that sent the request
HTTPS	**ON** (or **on**) if the request came in through a secure channel (SSL) or it returns **OFF** (or **off**) if the request is for a non-secure channel
REMOTE_ADDR	IP address of the remote host making the request
SERVER_NAME	Host name, DNS alias, or IP address of the server as it would appear in self-referencing URLs
SERVER_PORT	Port number to which the request was sent
URL	Base portion of the URL

An exception is raised if this method is invoked from a server method.

You can implement your own **getServerVariable** method (equivalent to this method in the **WebSession** class) if you are using a **JadeInternetTCPIPConnection** instance to communicate with the **jadehttp** library (that is, **jadehttp.dll**) when your application does not use **WebSession** functionality.

A name that is longer than 100 characters retrieved by the **getServerVariable** method call is truncated to 100 characters. This name is used to determine the method to invoke when using non-wrapped document literal format messages. When the name does not meet the JADE method naming requirements, the method invocation is likely to fail and a SOAP fault to be returned to the Web service consumer.

The following method returns the value of the Internet Server Application Programming Interface (ISAPI) variable (specified by the **var** parameter) associated with an Internet message that is received.

```
getServerVariable(var: String): String;
// The request for the ISAPI variable var is built in the bin variable
// The JadeInternetTCPIPConnection instance must exist and be connected
constants
    NULL:       Character = #00.Character;
vars
    bin:        Binary;
    connection: JadeInternetTCPIPConnection;
begin
    if connection <> null and connection.state = Connection.Connected then
        if IsUnicodeSystem then
           bin := ("GSV" & NULL & var.trimBlanks &
                    NULL).Binary._unicodeToAnsi; // unpublished Binary method
        else
           bin := ("GSV" & NULL & var.trimBlanks & NULL).Binary;
        endif;
        connection.writeBinary(bin);
        bin := connection.readBinary(0);
    endif;
    if IsUnicodeSystem then
        return bin.ansiToUnicode.trimBlanks;
    else
        return bin.String.trimBlanks;
```

```
    endif;
end;
```

Caution You can call this method only during the processing of a received Internet message and before the reply is sent. Accessing the method at any other time causes the process to wait indefinitely for the connection read or causes the message exchange process with the **jadehttp** library to be out of step.

getSessionForm

Signature `getSessionForm(formName: String): Form;`

The **getSessionForm** method of the **WebSession** class returns a transient instance of the form specified in the **formName** parameter, to enable you to keep track of all open form instances for the current session in a Web-enabled application that uses JADE forms (that is, a Web-enabled application that does not use HTML documents).

If there are multiple instances of the specified form, the first instance is returned. If there are no open forms for the specified value, a null value is returned.

getWebSessionCount

Signature `getWebSessionCount(): Integer;`

The **getWebSessionCount** method of the **WebSession** class, which returns the total number of active Web sessions for all nodes connected to the JADE server, can be reimplemented in your user session class.

This method returns zero (**0**) if there are no current active Web sessions.

isVDFilePresent

Signature `isVDFilePresent(fileName: String): Boolean;`

The **isVDFilePresent** method of the **WebSession** class returns whether the file specified in the **fileName** parameter is present on the Web server side of the firewall when using the JADE HTML (Web) thin client interface via the **jadehttp** library or **mod_jadehttp** module. The method returns **true** if the specified file exists or it returns **false** if it does not exist.

This method sends a message to the **jadehttp** library or **mod_jadehttp** module to perform this action. If the specified file name does not have a directory part, the current virtual directory defined in the **VirtualDirectory** parameter in the **jadehttp.ini** file for the application is used for Microsoft Internet Information Server; the **PhysicalDirectory** defined indirectly through the **httpd.conf** file is used for Apache HTTP server.

The file specified in the **fileName** parameter of the **isVDFilePresent** method is used if the file name has a directory part.

See also the **WebSession** class **createVirtualDirectoryFile** method.

processRequest

Signature
```
processRequest(httpString:  String;
               queryString: String) updating;
```

The **processRequest** method of the **WebSession** class, which is executed when a request is received from the Web, can be reimplemented in your user session class. The appropriate form is then updated with the information received from the incoming string and a reply is sent back to the Web browser after all processing is complete.

The **httpString** parameter is the string returned from a Web browser request as a result of a POST action on a Web page. When there is no POST action, the string that is returned is the same as the value returned by the **queryString** parameter.

The **queryString** parameter is the string returned as a result of the selection of a hyperlink or the entry of a Uniform Resource Locator (URL) on the address line of the Web browser.

You can reimplement this method in your applications. The following example shows the use of this method to extract information from the string to perform some prior processing.

```
processRequest(httpString, queryString: String) updating;
vars
    userName: String;
begin
    //look for a field name called userName
    userName := currentSession.getHttpParam('userName');
    if userName <> null then
        // do some processing
    endif;
    //now call the default implementation
    inheritMethod(httpString, queryString);
end;
```

Notes If the JADE implementation of this method is not called (by using the **inheritMethod** instruction), it is your responsibility to do any processing that is necessary and to send a reply back to the browser. For more details, see the **WebSession** class **reply** method.

JADE expects the incoming **httpString** parameter value to be in a certain format. If the string that is passed to the **inheritMethod** call violates this assumption, the results may be unpredictable; that is, an exception may be raised or information may be lost or updated incorrectly.

removeSession

Signature `removeSession();`

The **removeSession** method of the **WebSession** class removes the current Web session. You can call this method in your code so that your application programmatically removes the current Web session, instead of using the Web Monitor utility to remove the current session.

If you want to send a specific message when the Web session is removed, use the **WebSession** class **removeSessionWithMessage** method.

removeSessionWithMessage

Signature `removeSessionWithMessage(message: String) updating;`

The **removeSessionWithMessage** method of the **WebSession** class, which removes the current Web session and sends the message specified in the **message** parameter, can be reimplemented in your user session class.

You can call this method in your code so that your application programmatically removes the current Web session and sends a specific message, instead of using the default message to send back to the browser.

Tip To remove a Web session without sending a response back to the browser when the Web session is ended after a response has been sent, call this method with a null (**""**) string in the **message** parameter.

An attempt by a user to reconnect to the session is then treated as a new session.

reply

Signature `reply(html: String) updating;`

The **reply** method of the **WebSession** class, which is executed when all processing is complete and the JADE system is ready to send a response back to the Web browser, can be reimplemented in your user session class. The **html** parameter specifies the response that is to be sent back to the Web browser.

You can reimplement this method in your applications; for example, if you want to manipulate the string before sending a reply back to the Web browser. The following example shows the use of this method to keep a count of the replies that are sent to the Web browser.

```
reply(html: String) updating;
begin
    //count is a property in the Global class of the user-defined schema
    beginTransaction;
        global.count := global.count + 1;
    commitTransaction;
    //now call the default implementation
    inheritMethod(html);
end;
```

Notes If the JADE implementation of this method is not called (by using the **inheritMethod** instruction), it is your responsibility to send a response back to the Web browser.

If you modify the **html** parameter value, it is your responsibility to ensure that the information is displayed as you expect.

You can use the **MinMessageSize** parameter in the [*application-name*] section of the **jadehttp.ini** file or the **MinMessageSize** directive in the Apache **mod_jadehttp.so** library to specify the minimum size allowed for a Web message received from JADE when using the **reply** method to send HTML string Web requests back to the client node. The minimum value is **1** byte, the maximum value **1024** bytes, and the default value **10** bytes. This value is read the first time the specified application is accessed after **jadehttp.dll** has been loaded by Internet Information Server (IIS) or it is read once, when the Apache Web server starts.

setCurrentLocale

Signature `setCurrentLocale(loc: Locale) updating;`

The **setCurrentLocale** method of the **WebSession** class dynamically sets the current locale to the locale that is specified in the **loc** parameter. Call this method to ensure that hyperlinks in Web applications respond as expected for Web forms displayed for a translated locale.

By default, JADE determines the locale from the Web browser that is making the request.

If there is a form defined for the locale that is making the request, the HTML for that form is generated. If no form is defined, the default locale is used.

Note If you want to select the translation of all forms in the application from the schema default locale, regardless of the current locale of the application, set the **FormsUseDefaultSchemaLocale** parameter in the [Jade] section of the JADE initialization file to **true**.

timerEvent

Signature `timerEvent(eventTag: Integer) updating;`

The **timerEvent** method of the **WebSession** class, which you can reimplement in your user Web session subclasses, is called when the session times out.

Use the **eventTag** parameter to specify the **WebSession** class **WebSession_System_Timer_Event** constant; for example, your methods could contain your application-specific processing code and then the following.

```
inheritMethod(WebSession_System_Timer_Event);
```

Note For Web session timeouts to work, you must call the **inheritMethod** instruction.

WebSocketException Class

The **WebSocketException** class is the transient class that defines behavior for exceptions that occur as a result of the WebSocket protocol.

For details about handling WebSocket exceptions, see error message 31600 in "Error Messages and System Messages", in the **JADEMsgs.pdf** file.

Inherits From: NormalException

Inherited By: (None)