Encyclopaedia of Classes

Volume 1

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 ..... **xxxi**

Who Should Read this Encyclopaedia xxxi
What's Included in this Encyclopaedia xxxi
Related Documentation xxxi
Conventions xxxii

Chapter 1 System Classes **33**

ActiveXAutomation Class 42
ActiveXAutomation Properties 43
remoteServerName 44
usePresentationClient 44
ActiveXAutomation Methods 44
attachAutomationObject 45
beginNotifyAutomationEvent 45
createAutomationObject 46
endNotifyAutomationEvent 46
getInterface 46
Example of Using an Imported ActiveX Automation Object 47
ActiveXInterface Class 49
ActiveXInvokeException Class 50
ActiveXInvokeException Properties 50
activeXErrorCode 50
description 50
helpContext 50
helpFile 51
source 51
Application Class 52
Application Class Constants 52
Application Properties 53
aboutForm 54
appVersion 54
applicationType 54
controlSpacing 56
currentLocale 57
currentLocaleInfo 57
defaultMdi 58
finalizeMethod 58
fontBold 58
fontName 58
fontSize 59
formMargin 59
heightSingleLineControl 59
helpFile 59
icon 60
initializeMethod 60
mdiCaption 60
mdiFrame 60
mousePointer 61
name 62
printer 62
showBubbleHelp 62
startupForm 63
userSecurityLevel 63
webMinimumResponseTime 64
Application Methods 64
activateApp 70

Contents iv

activeControl 71
activeForm 71
actualTime 71
actualTimeAppServer 71
actualTimeServer 71
actualTimeStampOffset 72
actualTimeStampOffsetAppServer 72
actualTimeStampOffsetServer 72
alert 72
asyncFinalize 73
asyncInitialize 73
beep 74
checkPictureFile 74
clock 75
clearWriteWindow 75
closeWriteWindow 76
computerName 76
computerNameAppServer 76
copyImageFromClipboard 76
copyImageToClipboard 77
copyStringFromClipboard 77
copyStringToClipboard 78
createSessionErrorMessage 78
currentUTCBias 78
dbPath 79
dbServerComputerName 79
debugApplication 79
debugApplicationWithParameter 79
defaultLocale 80
doWindowEvents 80
enableThinClientConnBalancing 81
endOdbcSession 82
executeMethodNotFoundMessage 82
finalize 82
finalizeOdbcSelect 82
flushThinClient 82
forms 83
formsCount 83
generateUuid 84
getApplicationSkin 84
getCurrentSession 85
getCurrentSessionId 85
getExternalDatabase 85
getForm 85
getEnhancedLocaleSupport 86
getIniFileName 86
getIniFileNameAppServer 86
getJadeInstallDir 87
getJadeInstallDirAppServer 87
getJadeTextEditGlobalSettings 87
getJadeTextEditOneSetting 87
getMessageText 88
getMouseMoveTime 88
getOdbcSessionObject 88
getParamListTypeEntry 89
getParamListTypeLength 89
getPersistentApp 90
getProcess 90
getProfileString 90
getProfileStringAppServer 91
getRootSchemaFormTranslation 92

Contents v

getSchema ... 93
getSessionTimeout ... 94
getSkin ... 94
getSkinCollection ... 94
getSystemVersion ... 95
getTempDir ... 95
getTempDirAppServer ... 95
getThinClientEncryptionType ... 95
getTransientDbPath ... 96
getUTCTime ... 96
getWebMachineName ... 96
getWebVirtualDirectory ... 96
globalLockException ... 97
htmlPageNotFoundMessage ... 97
inactiveTimeout ... 97
initialize ... 97
initializeOdbcSelect ... 98
invalidWebSessionMessage ... 98
isActiveXClassIdRegistered ... 98
isAppRunning ... 99
isBeingDebugged ... 99
isControlSupported ... 99
isFormOpen ... 99
isMultiUser ... 99
isUnicode ... 100
isValidObject ... 100
isWebService ... 100
jadeReportWriterAppName ... 100
jadeReportWriterParamLiteral ... 100
jadeReportWriterParameterIsSet ... 101
jadeReportWriterParamObjects ... 101
jadeReportWriterTimeDetails ... 101
jadeWebServiceInputError ... 102
licencesExceededMessage ... 102
loadPicture ... 102
minimumResponseTimeExceededMsg ... 103
msgBox ... 103
- Using the flags Parameter ... 104
- msgBox Method Return Values ... 105
- Handling Translatable Strings on Message Box Button Captions ... 105
- Message Box Translatable String Handling for Button Captions ... 106

odbcWorkerFinalize ... 106
odbcWorkerInitialize ... 106
paintIfRequired ... 107
playSound ... 107
playSoundAsync ... 108
productionMode ... 108
random ... 108
random31 ... 109
relativeMachineMicros ... 109
relativeMachineTime ... 110
removeSessionMessage ... 110
repairCollection ... 110
rpsDataPumpFinalize ... 111
rpsDataPumpInitialize ... 111
seedRandom ... 112
serverName ... 112
setApplicationSkin ... 112
setEndpointForWebService ... 113
setInactiveTimeoutPeriod ... 114
setJadeLocale ... 114

Contents vi

setMouseMoveTime 115
setOdbcSessionObject 115
setProfileString 115
setProfileStringAppServer 116
setSessionTimeout 117
setSkin 118
setStatusLineDisplay 118
setWebMachineName 118
setWebVirtualDirectory 118
skinDelete 119
skinExtract 119
skinLoad 119
skinMakeDirectory 120
startApplication 120
startApplicationWithParameter 121
startApplicationWithString 123
startAppMethod 125
startOdbcSession 126
timedOutSessionMessage 127
updateJadeTextEditAppSettings 127
userName 127
webApplicationDirectory 127
ApplicationContext Class 129
ApplicationContext Properties 129
initialApp 129
initialPackage 129
initialProcess 130
initialSchema 130
Array Class 131
Using Subscripts in Arrays 131
Array Methods 132
add 132
at 133
atPut 133
countOf 134
countOf64 134
createIterator 134
first 134
getStatistics 135
includes 136
indexOf 136
indexOf64 136
initialise 136
insert 137
last 137
remove 137
removeAt 137
replace 137
BinaryArray Class 138
BooleanArray Class 139
Btree Class 140
Btree Method 140
setLoadFactor 140
ByteArray Class 141
ByteArray Methods 141
binarySearch 141
binarySearch64 141
CharacterArray Class 143
Class Class 144
Caveat When Handling Persistent Class Instances 144
Caveat When Handling Shared Transient Class Instances 145

Contents vii

Class Properties 145
implementedInterfaces 145
instances 146
properties 146
subclasses 146
superclass 146
transient 147
Class Methods 147
addDynamicProperty 149
addDynamicPropertyCluster 150
allInstances 150
allInstancesInPartition 151
allLocalSubclasses 151
allProcessTransientInstances 151
allProperties 152
allPropertiesUpTo 152
allSharedTransientInstances 152
allSubclasses 153
allSubclassesInSubschemas 153
allSubclassesUpToSchema 153
allSuperclassesUpTo 153
anyInstance 154
causeClassEvent 154
compactDynamicPropertyClusters 155
countPersistentInstances 156
countPersistentInstances64 156
countPersistentInstancesLim64 156
countPersistentInstancesLimit 157
createPartition 157
deleteDynamicProperty 158
deleteDynamicPropertyCluster 158
findConstant 158
findDynamicPropertyCluster 158
findLocalSubclass 159
findMethodInSubclasses 159
findProperty 159
findPropertyInSubClasses 159
firstInstance 159
firstProcessTransientInstance 160
firstSharedTransientInstance 160
getConstantInHTree 160
getDbFile 160
getImplementors 160
getMethodInHTree 160
getNextSubClasses 161
getProperties 161
getProperty 161
getPropertyInHTree 161
getRpsMappingRefs 161
getSubclass 161
getSubclasses 162
getSubclassesUpToSchema 162
getSuperclass 162
hasInstance 162
hasRpsReferences 162
hasSubclasses 162
implementsInterface 162
instancesExist 163
lastInstance 163
lastProcessTransientInstance 163
lastSharedTransientInstance 163

needsReorg 163
resynchInstances 164
withAllSubclasses 164
withAllSuperclasses 164
CMDialog Class 165
CMDialog Properties 166
dialogTitle 166
helpContextId 166
helpFile 166
helpKeyword 167
CMDColor Class 169
CMDColor Properties 169
color 169
customColors 169
fullOpen 170
preventFullOpen 170
CMDColor Method 170
open 170
CMDFileOpen Class 172
CMDFileOpen Properties 172
allowMultiSelect 173
createPrompt 173
defaultExt 173
extensionDifferent 174
fileMustExist 174
fileName 174
fileTitle 175
filter 175
filterIndex 175
hideReadOnly 176
initDir 176
noReadOnlyReturn 176
pathMustExist 176
readOnly 176
resetCurrentPath 176
shareAware 177
validate 177
CMDFileOpen Methods 177
getMultiSelectCount 177
getMultiSelectDirectory 178
getMultiSelectFileTitle 178
open 178
CMDFileSave Class 179
CMDFileSave Properties 179
allowMultiSelect 180
createPrompt 180
defaultExt 180
extensionDifferent 180
fileMustExist 180
fileName 181
fileTitle 181
filter 181
filterIndex 182
hideReadOnly 182
initDir 182
noReadOnlyReturn 182
overwritePrompt 183
pathMustExist 183
readOnly 183
resetCurrentPath 183
shareAware 183

Contents ix

validate 184
CMDFileSave Methods 184
getMultiSelectCount 184
getMultiSelectDirectory 184
getMultiSelectFileTitle 184
open 185
CMDFont Class 186
CMDFont Properties 186
ansiOnly 187
fixedPitchOnly 187
fontBold 187
fontItalic 187
fontName 188
fontSize 188
fontStrikethru 188
fontUnderline 188
forceFontExist 188
maxSize 189
minSize 189
noNameSelection 189
noSizeSelection 189
noStyleSelection 189
printerDC 190
printerDC64 190
printerFonts 190
scalableOnly 190
screenFonts 191
showEffects 191
simulations 191
textColor 191
trueTypeOnly 191
vectorFonts 191
wysiwyg 192
CMDFont Method 192
open 192
CMDPrint Class 193
CMDPrint Class Constants 193
CMDPrint Properties 193
allPagesStatus 194
collateStatus 194
copies 195
disablePageNumbers 195
disablePrintToFile 195
disableSelection 195
documentType 195
duplex 198
fromPage 198
hidePrintToFile 199
initializeWith 199
maxPage 199
minPage 199
orientation 199
pageNumbersStatus 200
paperSource 200
printSetup 201
printToFileStatus 201
printerDC 201
printerDC64 201
printerName 201
returnDC 202
selectionStatus 202

Contents x

toPage 202
warnIfNoDefault 202
CMDPrint Method 203
open 203
Collection Class 204
Collection Methods 205
add 206
clear 206
copy 206
countOf 207
countOf64 207
createIterator 207
deleteIfEmpty 207
first 208
getOwner 208
getStatistics 208
includes 209
indexNear 209
indexNear64 210
indexOf 210
indexOf64 210
inspect 210
inspectModal 210
instantiate 211
isEmpty 211
last 211
maxSize 211
maxSize64 211
purge 211
rebuild 212
remove 212
setBlockSize 212
size 213
size64 213
Connection Class 214
Connection Class Constants 214
Connection Properties 214
fillReadBuffer 215
name 215
state 216
timeout 216
Connection Methods 217
close 217
closeAsynch 218
getMaxMessageSize 219
getNextSessionId 219
listen 219
listenAsynch 219
listenContinuous 221
listenContinuousAsynch 221
open 222
openAsynch 223
openPipeCallback 224
readBinary 224
readBinaryAsynch 225
readPipeCallback 226
readUntil 226
readUntilAsynch 227
sendReply 227
writeBinary 227
writeBinaryAsynch 228

Contents xi

ConnectionException Class 230
ConnectionException Properties 230
connection 230
dataBuffer 230
ConstantNDict Class 231
CurrencyFormat Class 232
CurrencyFormat Class Constants 232
CurrencyFormat Properties 233
intlCurrencySymbol 233
intlDecimalPlaces 233
negSymbolPrecedesAmount 233
negSymbolSeparated 234
negativeSignPosition 234
posSymbolPrecedesAmount 234
posSymbolSeparated 235
positiveFormat 235
symbol 235
CurrencyFormat Method 235
defineCurrencyFormat 235
Database Class 238
Database Properties 238
dbFiles 238
path 238
schema 238
serverName 238
Database Methods 239
getFile 239
getName 239
DateArray Class 240
DateFormat Class 241
DateFormat Class Constants 241
DateFormat Properties 242
activeCalendarType 242
dayHasLeadingZeros 243
firstDayOfWeek 243
firstWeekOfYear 243
longDayNames 243
longFormat 243
longFormatOrder 244
longMonthNames 244
monthHasLeadingZeros 244
optionalCalendarType 244
separator 245
shortDayNames 245
shortFormat 245
shortFormatOrder 245
shortMonthNames 245
showFullCentury 245
DateFormat Methods 246
defineLongDateFormat 246
defineShortDateFormat 247
DbFile Class 249
DbFile Class Constants 249
DbFile Properties 250
database 250
excludeFromBackup 251
kind 251
partitionable 251
path 252
DbFile Methods 252
backupFile 253

beginPartitionedFileBackup 254
certifyFile 255
changeAccessMode 255
compactFile 255
createPartition 256
disableAuditing 256
drop 257
enableAuditing 257
endPartitionedFileBackup 257
freeze 257
getBackupTimestamp 257
getCreationTimestamp 258
getCryptStatus 258
getFileLength 258
getFileStatus 258
getFreeSpace 259
getFullBackupTimestamp 260
getModifiedTimestamp 260
getName 260
getOpenPartitions 260
getPartition 260
getPartitionCount 261
getPartitionModulus 261
getPartitions 261
getPatchVersion 261
getStatistics 262
getTotalFileLength64 264
getUserPatchVersion 265
isAuditing 265
isFrozen 265
isOpen 265
isPartitioned 265
setPartitionModulus 266
setPartitioned 266
thaw 266
DbFile Class Event Notifications 266
Subscribing to Backup Progress Events 267
Notification Event Methods 267
DbFileArray Class 269
DeadlockException Class 270
DeadlockException Properties 270
lockDuration 270
lockTimeout 270
lockType 271
targetLockedBy 271
DeadlockException Methods 271
lockTarget 272
obtainedLock 272
DecimalArray Class 273
Dictionary Class 274
Using String Keys in Dictionary Methods 274
Using Subscripts in Dictionaries 274
Dictionary Methods 274
createIterator 275
getAtKey 276
getAtKeyGeq 277
getAtKeyGtr 277
getAtKeyLeq 278
getAtKeyLss 278
getIteratorKeys 278
includesKey 279

Contents xiii

removeKey 279
removeKeyEntry 279
startKeyGeq 280
startKeyGtr 280
startKeyLeq 281
startKeyLss 281
stringKeyCompareGeq 281
stringKeyCompareGtr 282
stringKeyCompareLeq 282
stringKeyCompareLss 283
DynaDictionary Class 285
DynaDictionary Methods 286
addExternalKey 286
addExternalKeyWithSortOrder 287
addMemberKey 287
addMemberKeyWithSortOrder 288
clearKeys 288
endKeys 289
isValid 289
setMembership 289
Using Dynamic Dictionaries 289
Exception Class 293
Exception Class Return Values 293
Exception Properties 295
category 295
continuable 295
currentMethodDesc 296
errorCode 296
errorItem 297
extendedErrorText 297
helpBook 298
kind 298
level 298
remoteErrorCode 298
reportingMethodDesc 298
resumable 299
Exception Methods 299
createSOAPMessage 300
debug 300
defaultHandler 300
errorObject 301
initializationHandler 301
logExceptionHistory 301
logProcessHistory 301
logSelf 301
setErrorObject 301
showDialog 302
text 302
ExceptionHandlerDesc Class 303
ExceptionHandlerDesc Properties 303
armingApplication 303
armingMethod 303
exceptionClass 304
exceptionHandlerMethod 304
exceptionHandlerReceiver 304
invocationCount 304
isGlobal 304
ExternalArray Class 305
Using Subscripts in External Arrays 305
ExternalCollection Class 306
ExternalCollection Properties 306

database 306
filterExpression 307
sortExpression 307
ExternalCollection Methods 308
at 308
canCreate 308
createIterator 308
createObject 309
first 309
getSQL 309
includes 309
last 309
maxSize 309
maxSize64 310
size 310
size64 310
ExternalDatabase Class 311
ExternalDatabase Properties 311
connectionString 311
name 311
password 311
serverName 312
userName 312
ExternalDatabase Methods 312
abortExternalTransaction 313
beginExternalTransaction 313
canTransact 313
close 313
commitExternalTransaction 313
executeSQL 314
getFileDSN 314
getLastError 314
getMachineDSN 314
importStoredProcedures 314
isOpen 314
isSQLValid 315
isUpdatable 315
loadProcedure 315
open 315
setFileDSN 315
setMachineDSN 315
ExternalDictionary Class 317
Associating External Dictionary Key Access Using Subscript Notation 317
ExternalDictionary Methods 317
getAtKey 318
getAtKeyGeq 318
getAtKeyGtr 318
getAtKeyLeq 319
getAtKeyLss 319
includesKey 319
startKeyGeq 319
startKeyGtr 319
startKeyLeq 319
startKeyLss 320
ExternalIterator Class 321
ExternalIterator Methods 321
back 321
getCollection 321
isValid 321
next 322
reset 322

Contents

startAtIndex 322
ExternalObject Class 323
ExternalObject Methods 323
deleteSelf 323
isUpdatable 323
update 323
ExternalSet Class 324
ExternalSet Method 324
includes 324
ExtKeyDictionary Class 325
Using Subscripts in Dictionaries 325
ExtKeyDictionary Methods 325
add 325
putAtKey 326
remove 326
FatalError Class 327
File Class 328
File Class Constants 329
File Properties 330
allowReplace 330
endOfField 330
endOfLine 331
kind 331
maxIOSize 332
maxRecordSize 333
mode 333
recordSize 333
shareMode 334
unicodeBOM 334
File Methods 334
close 335
commit 336
currentOffset 336
currentOffset64 337
currentOffsetDec 337
endOfFile 337
errorCode 338
extractSort 338
Example of Sorted Variable Fields or Records 339
Example of Sorted Fixed Fields or Records 339
fileLength 340
fileLength64 341
fileLengthDec 341
isAvailable 342
isOpen 342
lastAccessed 342
lastModified 342
open 342
Creating or Replacing Unicode Text Files 343
openInput 343
openOutput 343
peek 343
purge 344
readBinary 344
readLine 344
readString 346
rename 346
seek 346
seek64 347
seekDec 347
timeCreated 347

tryOpen 347
writeBinary 347
writeLine 348
writeString 348
FileException Class 349
FileException Methods 349
file 349
fileNode 349
FileFolder Class 350
FileFolder Property 350
mask 350
FileFolder Methods 350
browseForAppServerFolder 351
browseForFolder 351
browseForServerFolder 352
files 352
isAvailable 353
isValidPathName 353
make 353
purge 353
rename 354
FileNode Class 355
FileNode Properties 355
allowCreate 355
archive 356
createdTime 356
fileName 356
hidden 356
lastAccessedTime 356
lastModifiedTime 357
name 357
readOnly 357
systemFile 357
usePresentationFileSystem 357
FileNode Methods 358
directorySeparator 358
isAvailable 358
purge 358
FileNodeArray Class 359
Global Class 360
Global Methods 360
alert 360
beep 361
getAndValidateUser 361
isUserValid 362
isValidObject 363
jadeReportWriterSystemName 363
lockExceptionHandler 363
Global Event 364
initialize 364
GUIClass Class 365
GUIClass Method 365
createPrintForm 365
HugeStringArray Class 366
IDispatch Class 367
IDispatch Methods 368
beginNotifyAutomationEvent 368
endNotifyAutomationEvent 369
getInterface 369
loadPicture 369
makePicture 369

savePicture 370
IDispatchArray Class 371
Integer64Array Class 372
Integer64Array Methods 372
binarySearch 372
binarySearch64 372
IntegerArray Class 374
IntegerArray Methods 374
binarySearch 374
binarySearch64 374
IntegrityViolation Class 376
InternetPipe Class 377
InternetPipe Methods 377
openPipeCallback 377
readPipeCallback 377
sendReply 378
Iterator Class 379
Iterator Methods 379
back 379
current 380
excludeOfflineObjects 380
getCollection 380
getCurrentKey 381
getCurrentKeys 381
isValid 382
next 382
reset 383
startAtIndex 383
startAtObject 383
startNearIndex 384
IUnknown Class 385
JadeAuditAccess Class 386
JadeAuditAccess Class Constants 387
JadeAuditAccess Properties 387
autoDescription 387
currentClassNumber 388
currentObjectType 388
currentOid 388
currentRecordType 388
descriptionFilename 389
descriptionPath 389
descriptionTS 389
JadeAuditAccess Methods 390
clearRegisteredFilters 391
descriptionClassIsSubclass 391
generateDescription 392
getAfterImage 392
getAfterPropertyValue 392
getBeforeImage 393
getBeforePropertyValue 393
getBlobProperty 393
getBlobValue 394
getChangeUserData 394
getChangedPropertyNames 394
getClassName 394
getClassNumber 395
getClassProperty 395
getClassPropertyNames 396
getCollectionBlockKeys 396
getCollectionBlockOid 396
getCollectionBlockOids 397

getJournal 397
getJournalName 397
getJournalNumber 397
getJournalPath 397
getNextJournal 398
getNextRecord 398
getNextRecordUTC 399
getProperty 401
getUTCBias 401
getUserData 401
loadDescription 401
loadDescriptionByName 402
registerFilterClass 402
registerFilterClassName 402
registerFilterCollection 402
registerFilterCollectionName 403
registerFilterTimeRange 403
registerFilterTimeRangeUTC 403
setAccessMode 404
setFilterExcludes 404
JadeAuditAccess Class Method Example 404
JadeBytes Class 407
Shared File JadeBytes 407
Dedicated File JadeBytes 408
JadeBytes Properties 408
singleFile 408
readOnly 409
unaudited 410
JadeBytes Methods 410
add 411
allocate 412
appendData 412
at 413
atPut 413
clear 414
copy 414
createIterator 414
display 415
extractToFile 415
extractToFileDirect 416
extractUsingFile 417
first 417
getContent 418
getData 418
getFileTitle 419
getLength 419
getSegmentCount 420
getSegmentSize 420
getStatistics 420
grow 421
isEmpty 421
last 422
loadFromFile 422
loadFromFileDirect 423
loadUsingFile 423
matchChecksum 424
purge 424
putData 425
setCaching 425
setContent 426
setExpectedLength 426

truncate 427
updateChecksum 427
JadeDatabaseAdmin Class 428
JadeDatabaseAdmin Class Constants 428
JadeDatabaseAdmin Methods 429
abortBackup 432
backupAllDbFiles 433
backupDbFiles 435
backupJournal 436
beginBackup 437
changeDbAccessMode 437
closeCurrentJournal 438
commitBackup 438
commitCoherentBackup 439
compactDbFiles 439
createRpsDatabase 439
disableByteProgressEvents 440
disableProgressEvents 441
doCheckpoint 441
doQuietpoint 441
enableByteProgressEvents 442
enableProgressEvents 442
getAbortJournalNumber 443
getAllDbFiles 443
getArchiveJournalDirectory 443
getCreationTimestamp 443
getCurrentJournalDirectory 444
getCurrentJournalName 444
getCurrentJournalNumber 444
getCurrentJournalOffset 444
getDbFiles 444
getLastCheckpoint 445
getLatestBackupTimestamp 445
getLatestFullBackupTimestamp 446
getOpenTimestamp 446
getReasonTrackingStoppedString 446
getRpsMappedFiles 447
isArchival 447
rpsAuditSqlScriptForReplay 447
rpsExtractData 448
rpsExtractDataAll 449
rpsExtractDataUsingIniOptions 450
rpsGetDatabaseParameters 450
rpsStartDataPump 451
rpsStopDataPump 451
sdsAuditStopTracking 451
sdsDisablePrimaryConnection 452
sdsDisablePrimaryConnectionAt 453
sdsDisableReadAccess 453
sdsDisableReadAccessAt 453
sdsEnableReadAccess 454
sdsEnableReadAccessAt 454
sdsGetDatabaseRole 454
sdsGetDatabaseSubrole 455
sdsGetMyServerInfo 455
sdsGetSecondaryInfo 461
sdsGetSecondaryProxies 461
sdsGetSecondaryProxy 462
sdsGetTransactions 464
sdsGetTransactionsAt 465
sdsInitiateHostileTakeover 465

Contents xx

sdsInitiateTakeover 465
sdsIsInitialized 466
sdsIsRunning 466
sdsReconnectNow 466
sdsReplayNextJournal 467
sdsReplayNextJournalAt 467
sdsResume 467
sdsResumeAt 467
sdsStartService 468
sdsStartTracking 468
sdsStartTrackingAt 468
sdsStopService 468
sdsStopTracking 468
sdsStopTrackingAt 469
verifyJournal 469
JadeDatabaseAdmin Class Event Notifications 469
Subscribing to JadeDatabaseAdmin Events 470
JadeDbFilePartition Class 472
JadeDbFilePartition Properties 472
dbFile 472
partitionID 472
JadeDbFilePartition Methods 472
allInstances 473
backupFilePartition 474
certifyFile 475
disableAuditing 475
display 475
drop 476
enableAuditing 476
freeze 476
getBackupTimestamp 476
getCreationTimestamp 477
getFileLength 477
getFileStatus 477
getFreeSpace 478
getFullBackupTimestamp 478
getLabel 478
getLocation 478
getModifiedTimestamp 479
getName 479
getStatistics 479
getTotalFileLength64 481
isAuditing 481
isFrozen 481
isOffline 481
markOffline 482
markOnline 482
move 482
setLabel 483
setLocation 483
thaw 483
JadeDotNetInvokeException Class 484
JadeDotNetInvokeException Properties 484
dotNetExceptionObject 484
helpLink 484
innerException 484
message 485
source 485
target 485
JadeDotNetType Class 486
JadeDotNetType Property 487

usePresentationClient 487
JadeDotNetType Methods 487
beginEventNotification 488
createBoxedPrimitive 488
createColor 489
createDotNetObject 489
createEventNameMap 490
createFont 490
createPicture 491
enableEvent 491
endEventNotification 492
getBoxedPrimitive 492
getColor 492
getFont 493
getPicture 493
JadeDynamicObject Class 495
JadeDynamicObject Properties 496
children 496
name 496
parent 496
type 497
JadeDynamicObject Methods 497
addProperty 497
clear 498
clearValues 498
display 498
getName 498
getPropertyName 498
getPropertyType 498
getPropertyTypeByIndex 498
getPropertyValue 499
getPropertyValueByIndex 499
propertyCount 499
setPropertyValue 499
setPropertyValueByIndex 499
JadeDynamicObjectArray Class 500
JadeDynamicPropertyCluster Class 501
JadeDynamicPropertyCluster Properties 501
name 501
properties 501
schemaType 501
JadeDynamicPropertyCluster Methods 502
addDynamicProperty 502
addExclusiveDynamicProperty 502
deleteDynamicProperty 503
findDynamicProperty 503
JadeGenericMessage Class 504
JadeGenericMessage Class Constants 504
JadeGenericMessage Properties 505
body 505
correlationID 505
createdWhen 506
expiresWhen 507
feedback 507
format 507
messageID 507
replyQueueFullName 508
report 508
retrievedWhen 508
senderID 508
tag 509

type 509
JadeGenericMessage Methods 509
appendBinary 510
appendBodyTuple 510
appendString 511
appendStringAsUtf8 511
appendStringUtf8 511
getBodyLength 511
getBodyTuple 511
getMessageProperty 512
getTransportName 512
getUtf8bodyAsString 512
initializeForGet 512
initializeForPut 513
setExpiryRelativeToNow 513
setMessageProperty 513
JadeGenericMessagingIF Interface 514
JadeGenericMessagingIF Interface Constants 515
JadeGenericMessagingIF Interface Callback Method Signatures 515
managementEvent 515
messageArrivedEvent 515
JadeGenericQueue Class 516
JadeGenericQueue Class Constants 516
JadeGenericQueue Properties 516
defaultGetMessageOptions 517
defaultPutMessageOptions 517
fullName 517
maxMemoryInuse 518
maxMessageCount 518
maxMessageLength 518
name 518
tag 519
JadeGenericQueue Methods 519
beginMessage 520
cancelNotify 521
close 521
countQueuedMessages 521
countReceivers 521
createMessage 521
getMaxSegmentLength 522
getMessage 522
getMessageByCorrelationID 523
getQueueManager 524
getQueueProperty 525
getTransportName 525
isOpen 525
isRemote 525
notifyEventsAsync 525
purge 526
putMessage 526
setQueueProperty 527
JadeGenericQueueManager Class 528
JadeGenericQueueManager Property 528
name 528
JadeGenericQueueManager Methods 528
getFullName 529
getQueueManagerProperty 529
getTransportName 529
setQueueManagerProperty 529
JadeHTMLClass Class 530
JadeHTMLClass Properties 530

generateHTMLForJadeTagsOnly 530
goToPage 531
httpString 531
securePage 531
JadeHTMLClass Methods 531
addCookie 532
addHiddenField 533
addOptionTag 533
addPageBounce 534
buildFormActionOnly 534
buildLink 534
clearCookies 535
generateHTMLString 535
getAttributes 536
getCookies 536
getHttpValue 536
processRequest 536
queryIncludePage 536
reply 537
replyAsBinary 537
setAttributes 538
tableAddData 538
tableAddInput 539
tableAddItem 539
tableBegin 540
tableBeginRow 540
tableEnd 540
tableEndRow 540
updateValues 541
wasPageBounced 541
JadeHTTPConnection Class 542
JadeHTTPConnection Class Constants 542
JadeHTTPConnection Properties 544
abs_path 544
connectTimeout 545
fragment 545
hostname 545
httpVersion 545
password 545
port 545
protocolFamily 546
proxyConfig 546
proxyHostname 546
proxyPassword 546
proxyUser 546
query 547
receiveTimeout 547
responseBody 547
responseHeaders 547
scheme 547
sendTimeout 548
state 548
url 548
usePresentationClient 548
user 549
JadeHTTPConnection Methods 549
configureProxy 550
getHeader 550
getHttpPage 551
getHttpPageBinary 551
open 552

queryConnectionIsClose 552
queryContentLength 552
queryContentType 553
queryDate 553
queryInfo 553
queryStatusCode 553
readBody 553
sendRequest 554
setAccept 554
setContentType 554
setKeepAlive 555
setReferer 555
setReload 555
setSoapAction 555
setUserAgent 555
JadeIdentifierArray Class 556
JadeInternetTCPIPConnection Class 557
JadeInternetTCPIPConnection Methods 557
openPipeCallback 557
readBinary 558
readDataWithLength 558
readPipeCallback 558
sendReply 559
writeBinary 559
JadeJson Class 560
JadeJson Constants 561
JadeJson Methods 561
generateJson 561
generateJsonFile 562
parse 563
parseFile 563
parsePrimitive 564
parsePrimitiveFile 564
JadeLicenceInfo Class 566
JadeLicenceInfo Class Constants 566
JadeLicenceInfo Properties 566
developmentLicences 567
expiryDate 567
jadeStandardMin 567
jadeThinMin 567
licenceName 567
licenceRestriction 567
maxDBSize 568
nHtmlThinSessions 568
nJadeDevProcesses 568
nJadeThinNonJadeDevProcesses 568
nNonJadeDevProcesses 568
nProcessesLeft 568
processLicences 568
uuid 569
JadeLicenceInfo Methods 569
display 569
getLicenceInfo 570
JadeLog Class 571
JadeLog Class Properties 572
bufferOutput 573
fileExtension 573
fileName 573
filePath 573
formatOutput 574
maxFileSize 574

Contents XXV

selector 574
versionFile 575
JadeLog Class Methods 575
commitLog 576
getActualFileName 577
getActualFileNameClient 577
getActualFileNameServer 577
info 577
infoClient 577
infoDump 577
infoDumpClient 578
infoDumpServer 578
infoServer 578
log 578
logClient 578
logDump 579
logDumpClient 579
logDumpServer 579
logServer 579
rollOverLog 579
rollOverLogClient 580
rollOverLogServer 580
trace 580
traceClient 580
traceDump 580
traceDumpClient 580
traceDumpServer 581
traceServer 581
JadeMessagingException Class 582
JadeMessagingException Properties 582
theMessage 582
theQueue 582
theQueueManager 582
JadeMessagingFactory Class 583
JadeMessagingFactory Class Methods 583
generateAccessPassword 583
openQueue 584
JadeMetadataAnalyzer Class 586
JadeMetadataAnalyzer Class Constants 586
JadeMetadataAnalyzer Methods 587
canAccessSchemaEntity 587
canAccessStatement 588
validateMethod 588
JadeMethodContext Class 590
JadeMethodContext Class Constants 590
JadeMethodContext Properties 591
state 591
tag 591
timeout 591
workerAppName 592
JadeMethodContext Methods 592
getErrorNumber 592
getErrorText 592
getReturnValue 592
getTimestamps 593
initialize 593
invoke 593
isComplete 594
isProcessing 594
isTimedOut 594
JadeMultiWorkerTcpConnection Class 595

JadeMultiWorkerTcpConnection Class Constants 595
JadeMultiWorkerTcpConnection Properties 595
connectionId 596
currentEventBusyElapsed 596
currentEventBusyWhen 596
currentEventQueuedElapsed 596
currentEventQueuedWhen 596
fillReadBuffer 597
idleTimeout 597
keepAssigned 597
localAddress 597
localPortnumber 597
remoteAddress 598
remotePortnumber 598
state 598
timeout 598
userObject 598
userState 599
JadeMultiWorkerTcpConnection Methods 599
bindToAssignedWorker 600
bindToWorkerId 600
causeUserEvent 600
close 600
getAssignedWorkerId 600
getBoundWorkerId 600
getFullName 601
getGroupId 601
getLocalHostname 601
getRemoteHostname 601
readBinary 601
readUntil 601
unbind 602
writeBinary 602
writeBinaryAndFile 602
JadeMultiWorkerTcpTransport Class 603
Connection Assignment 604
Connection Binding 604
JADE MultiWorker Events 604
Connection Events 605
Management Events 605
Reading and Writing Data 606
JadeMultiWorkerTcpTransport Class Constants 606
JadeMultiWorkerTcpTransport Properties 606
listenHostname 607
listenPortnumber 607
listenState 607
queueDepthLimit 607
queueDepthLimitTimeout 607
statisticsLogInterval 608
userGroupObject 608
workerId 608
workerIdleTimeout 608
JadeMultiWorkerTcpTransport Methods 608
beginListening 609
cancelNotify 609
causeUserEventOnConnId 610
countAssignedConnections 610
countBoundConnections 610
countIdleWorkers 610
countQueuedConnections 610
countWorkers 610

Contents xxvii

getFullName 611
getGroupId 611
getGroupStatistics 611
getMyStatistics 611
notifyEventsAsync 611
stopListening 612
validateServerProcess 612
JadeMultiWorkerTcpTransportIF Interface 613
JadeMultiWorkerTcpTransportIF Interface Constants 614
JadeMultiWorkerTcpTransportIF Interface Callback Method Signatures 614
closedEvent 614
connectionEvent 614
managementEvent 615
openedEvent 615
readReadyEvent 615
userEvent 615
JadePatchControlInterface Class 616
JadePatchControlInterface Methods 616
closePatchNumber 616
getAllEntitiesForPatchNumber 616
getCheckedOutEntitiesForPatch 617
getDeletedEntitiesForPatchNo 617
getEntitiesForPatchNumber 617
getOpenPatchNumbers 617
getPatchDetailsForPatchNumber 617
getResynchClasses 618
reassignPatchNumbers 618
setPatchNumber 619
unsetPatchNumber 619
JadePrintData Class 620
JadePrintDirect Class 621
JadePrintDirect Properties 622
fontBold 622
fontItalic 623
fontName 623
fontSize 623
fontStrikethru 624
fontUnderline 624
left 624
text 625
top 625
JadePrintPage Class 626
JadePrintPage Properties 626
customPaperHeight 627
customPaperWidth 627
documentType 627
orientation 629
pageImage 630
pageNumber 630
paperSource 630
JadeProfiler Class 631
JadeProfiler Properties 633
codeCoverageFileName 633
fileName 633
methodCount 634
profileRemoteExecutions 634
reportActualTime 634
reportCacheStatistics 635
reportInCSVFormat 635
reportLoadTime 635
reportMethodSize 635

reportStatistics 636
reportTotalTime 636
JadeProfiler Methods 636
clearMethodCache 637
isProfiling 637
report 637
reportCodeCoverage 639
reset 640
resetCodeCoverage 640
start 640
startCodeCoverage 640
stop 640
stopCodeCoverage 641
viewCodeCoverage 641
JadeRelationalAttributeIF Interface 642
JadeRelationalAttributeIF Interface Method Signatures 643
getJadeType 643
getLength 643
getSQLName 643
getScaleFactor 643
JadeRelationalEntityIF Interface 644
JadeRelationalEntityIF Interface Method Signatures 645
allInstances 645
callIFAllInstances 645
getJadeClass 646
getPropertyValue 646
getQueryProvider 646
getSQLName 646
isAttributeValid 646
JadeRelationalQueryProviderIF Interface 647
JadeRelationalQueryProviderIF Interface Constants 648
JadeRelationalQueryProviderIF Interface Callback Method Signatures 648
binaryExpression 648
executeQuery 649
finalizeQuery 649
getResultSet 649
unaryExpression 650
JadeReport Class 651
JadeReport Properties 652
collate 652
copies 652
description 652
duplex 652
printArray 653
timeStamp 653
JadeReport Method 653
pageCount 653
JadeReportWriterManager Class 654
Running an Existing Report 655
JadeReportWriterManager Methods 656
createReport 657
getAllReportNames 657
getAllViewNames 658
getFolderPaths 658
getFoldersInFolderPath 658
getReport 659
getReportsInFolderPath 659
getViewDetails 659
isReportWriterInstalled 660
setSecurity 660
setSecurityObject 660

setUserName 660
startReportDesignerForReport 661
startReportWriterConfiguration 661
startReportWriterDesigner 661
JadeReportWriterReport Class 663
JadeReportWriterReport Class Constants 663
JadeReportWriterReport Methods 664
getDefaultOutputDestination 666
getDefaultProfile 666
getDelimitedFileOptions 667
getExtraParameterDetails 668
getFolder 668
getHtmlOptions 668
getOutputFileTitle 669
getPageOptions 669
getParameterDetails 670
getParameters 670
getParametersForProfile 671
getProfileDescription 671
getProfiles 671
getQueryOptions 672
getReportDescription 673
getReportingViewName 673
getRootCollections 673
getTextOptions 673
getUseClientFileSystem 674
getXmlOptions 674
run 674
runWithStatus 675
setDelimitedFileOptions 677
setEndingNotification 677
setHtmlOptions 678
setLocaleDateOptions 678
setLocaleNumericOptions 679
setLocaleTimeOptions 679
setOutputDestination 680
setOutputFileTitle 680
setPageOptions 681
setParameter 682
setParameterIgnoreInSelection 682
setPreviewOptions 683
setProfile 684
setQueryOptions 684
setStartEndMethods 685
setTextOptions 685
setUseClientFileSystem 686
setXmlOptions 686
JadeReportWriterSecurity Class 687
JadeReportWriterSecurity Class Constants 687
JadeReportWriterSecurity Methods 687
canAccessConfiguration 688
canAccessDesigner 689
canAccessFolder 689
canAccessReport 690
canAccessView 690
canAccessViewClass 691
canAccessViewFeature 691
canDeleteReport 692
canMaintainFolders 692
canMaintainSystemOptions 693
canMaintainViews 693

Contents XXX

folderDeleted 693
folderPathChanged 693
isViewFeatureAccessSet 694
newFolderAdded 694
newReportAdded 694
newViewAdded 694
reportDeleted 695
reportNameChanged 695
viewDeleted 695
viewNameChanged 695
JadeRestService Class 696
JadeRestService Class Constants 696
JadeRestService Properties 696
httpStatusCode 696
objectsToBeDeleted 697
JadeRestService Methods 697
createVirtualDirectoryFile 697
deleteVirtualDirectoryFile 698
getOutputFormat 698
getServerVariable 699
isVDFilePresent 699
processRequest 700
reply 700
JadeRpsDataPumpIF Interface 701
JadeRpsDataPumpIF Interface Constants 702
JadeRpsDataPumpIF Interface Callback Method Signature 702
updateCallback 702
JadeSerialPort Class 704
JadeSerialPort Class Constants 704
JadeSerialPort Properties 705
dataBits 705
flowControl 705
parity 706
speedBps 706
stopBits 707
usePresentationClient 708
JadeSerialPort Methods 708
close 709
closeAsynch 709
listen 710
listenAsynch 710
open 711
openAsynch 712
readBinary 713
readBinaryAsynch 713
readUntil 715
readUntilAsynch 715
writeBinary 716
writeBinaryAsynch 716
JadeSkin Class (superseded from JADE release 6.0) 718

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 first volume contains system (non-GUI) classes in the range **ActiveXAutomation** class through **JadeSkin** class, inclusive. Volume 2 (that is, **EncycloSys2.pdf**) contains system (non-GUI) classes in the range **JadeSkinApplication** class through **WebSession** 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 "DbFile Class Event Notifications" 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 1) contains system (non-GUI) classes in the range **ActiveXAutomation** class through **JadeSkin** class, inclusive. Volume 2 (that is, **EncycloSys2.pdf**) contains system (non-GUI) classes in the range **JadeSkinApplication** class through **WebSession** 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	Holds references to instances of **Constant** (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
FileFolder	Contains a collection of files or subdirectories

Class	Description
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 the 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
JadeGenericQueue	Encapsulates a destination for the transmission and retrieval of messages

Class	Description
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 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 licence 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 connections 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 in a relational view
JadeRelationalEntityIF	Provides an interface to expose soft entities, which are mapped to a table in the relational view
JadeRelationalQueryProviderIF	Provides a search implementation that optimally finds and filters instances of a soft entity

Class	Description
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
JadeTcpIpProxy	Implements TCP/IP network proxy support that enables you to open a TCP/IP network connection through a proxy host

Class	Description
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
List	Encapsulates behavior required to reference objects by their position in the collection
Locale	Defines the locales (languages) supported by a schema

Class	Description
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
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

Class	Description
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
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

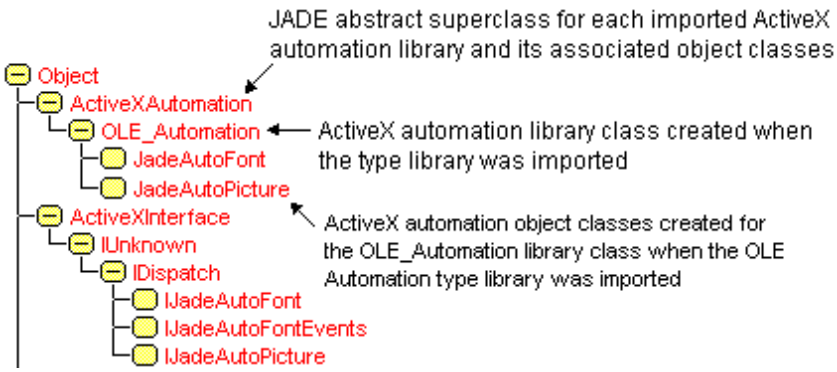
Class	Description
Type	Superclass of all class and primitive type meta classes
UserInterfaceException	Defines behavior for exceptions relating to the handling of windows
WebSession	Maintains Internet session information
WebSocketException	Defines behavior for WebSocket protocol exceptions

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

ActiveXAutomation Class

The **ActiveXAutomation** class is the abstract class that provides a superclass for all ActiveX automation library and object classes imported into JADE. You can create only transient instances of **ActiveXAutomation** subclasses.

When you import an ActiveX automation type library into JADE, an abstract class of the specified type library name is created as a subclass of the **ActiveXAutomation** class. This abstract class becomes the superclass for all classes that are subsequently generated corresponding to objects in the imported automation type library. The following example shows the hierarchy in the **RootSchema** of the OLE Automation type library that was preloaded into JADE.



Automation is the ability of a client to drive or direct a Component Object Model (COM) object by calling methods or setting properties using one or more interfaces of that object. For example, Microsoft Excel and Word are automation controllers; that is, they are objects that can be controlled by automation. Automation is simply the execution of a set of commands that set and get properties and call methods using the properties and methods of the generated ActiveX interface classes.

To handle events in ActiveX automation, you must register your interest in a specific event, which involves specifying the event method of the interface and the method that you want executed when the event is triggered. (For details, see the **beginNotifyAutomationEvent** method.)

To use an automation object from within JADE, you simply create an instance of the JADE class (that is, a subclass of the **ActiveXAutomation** class) that corresponds to the automation object and call the **createAutomationObject** method before the first property access or method call required to support ActiveX. This creates the automation object in the server and a transient instance of the default interface (which is a subclass of the **IDispatch** class).

When an automation object (that is, an instance of an **ActiveXAutomation** subclass) is deleted, JADE notifies COM that it has finished with it so that COM can do whatever it wants with the DLL of the object, and so forth. As this occurs in the destructor of the **ActiveXAutomation** subclass object, if you do not explicitly delete the instance, the destructor is not called and memory for the transient instance is not removed until the application closes down. You should therefore always delete your transient objects when they are no longer needed.

A reference is established between the automation class and its default interface. You can then call JADE automation class methods as you can for any other JADE class. These method calls are passed by the default interface to the actual automation object.

If the automation object returns a reference to another interface in response to a method call or the getting of a property, JADE creates an instance of the corresponding JADE interface class and returns a reference to that instance instead. (A mapping is maintained between JADE interface instances and automation server interface instances.)

Note In JADE thin client mode, ActiveX automation objects run on the presentation client by default.

Although you cannot specify the **serverExecution** method option for ActiveX automation methods, you can use them in a non-GUI application started by using the **app.startApplication** method that is part of a server execution method.

As most type libraries include the OLE Automation library, this **STDOLE2.TLB** library has been preloaded into the **RootSchema** so that it is provided with your installed JADE development environment. The **OLE_Automation** class provided as a subclass of the **ActiveXAutomation** class contains the **JadeAutoFont** and **JadeAutoPicture** standard object subclasses. You can create and then manipulate instances of the OLE automation object in the same way that you can any other imported ActiveX object.

The following example shows the creation of a font object and the setting of font properties using the supplied **JadeAutoFont** class.

```
createAFont();
vars
    autoFont : JadeAutoFont;
begin
    create autoFont;
    write autoFont.bold;          // Outputs false
    autoFont.bold := true;
    write autoFont.bold;          // Outputs true
    delete autoFont;
end;
```

For details about the methods and properties defined in the preloaded **OLE_Automation** object class and the **JadeAutoFont** and **JadeAutoPicture** subclasses, refer to your COM documentation. For details about importing ActiveX control and automation type libraries into JADE, see Chapter 4 of the *JADE External Interface Developer's Reference*.

For details about the properties and methods defined in the **ActiveXAutomation** class and using an imported ActiveX automation type library within JADE, see "ActiveXAutomation Properties", "ActiveXAutomation Methods", and "Example of Using an Imported ActiveX Automation Object", in the following subsections.

Inherits From: Object

Inherited By: Preloaded **OLE_Automation** class, automation objects imported by developers

ActiveXAutomation Properties

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

Method	Description
remoteServerName	Contains the name of the machine on which the ActiveX automation object is executed
usePresentationClient	Specifies whether the ActiveX automation object runs on the presentation client or the application server

remoteServerName

Type: String[256]

The **remoteServerName** property of the **ActiveXAutomation** class contains the name of the machine on which the ActiveX automation object is executed.

The ActiveX automation object must be a Distributed Component Object Model (DCOM) server. DCOM is included as part of all supported Windows operating systems.

You can specify any Universal Naming Convention (UNC) or Domain Name Service (DNS) name (for example, **"\\server"**, **"jadeworld.com"**, or **"123.4.56.78"**).

You can use the **remoteServerName** property in conjunction with the **usePresentationClient** property. The **usePresentationClient** property tells JADE to interface with the Component Object Model (COM) on the presentation client and the **remoteServerName** property tells COM the machine on which the ActiveX object is to run (which can be a machine other than one on which JADE is running). You could therefore have the **jadrap**, **jadapp**, and **jade** executable programs and the automation server all running on different machines.

Note You cannot set the **remoteServerName** property and then call the **attachAutomationObject** method to attach to an instance of the automation server that is already running, as you can attach only to local servers.

If you specify an invalid remote server name, an exception is raised when COM attempts to create the object when the **createAutomationObject** method is called.

usePresentationClient

Type: Boolean

The **usePresentationClient** property of the **ActiveXAutomation** class specifies whether the ActiveX automation object is run on the presentation client.

By default, automation objects are run on the presentation client; that is, this value is set to **true**. To run the ActiveX automation object on the application server, set this property to **false**.

Notes ActiveX controls must always be run on the presentation client.

This property is ignored when the application is running in standard, or fat, client mode.

For details about using this property in conjunction with the **remoteServerName** property, see the **ActiveXAutomation** class **remoteServerName** property.

ActiveXAutomation Methods

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

Method	Description
attachAutomationObject	Attempts to attach to an instance of the automation server that is already running
beginNotifyAutomationEvent	Registers the receiver to be notified when an event occurs on an ActiveX automation object
createAutomationObject	Creates an instance of the ActiveX automation object

Method	Description
endNotifyAutomationEvent	Terminates a previous **beginNotifyAutomationEvent** event
getInterface	Returns the specified ActiveX interface if it exists

attachAutomationObject

Signature `attachAutomationObject(createIfNone: Boolean): IDispatch;`

The **attachAutomationObject** method of the **ActiveXAutomation** class attempts to attach to an instance of the automation server that is already running. Use the **createIfNone** parameter to indicate that a new server should be started if an existing server cannot be found to which to attach.

For details about creating a new instance of the ActiveX automation object defined by the receiver, see "createAutomationObject", later in this section.

Notes For an automation server to be attached to, it must have registered itself in the Windows Running Object Table (ROT). Some servers (for example, Microsoft Office applications) register only the first instance of each application. Any **attachAutomationObject** calls therefore always connect to the first instance of the application. For example, if you start four Excel applications, only the first application is recorded in the ROT. If multiple instances of an application are recorded in the ROT, the **attachAutomationObject** method always attaches to the first instance that is found.

As you can attach to a local server only, you cannot set the **remoteServerName** property and then call the **attachAutomationObject** method.

beginNotifyAutomationEvent

Signature
```
beginNotifyAutomationEvent(receiver:          Object;
                           eventClassRefName: String);
```

The **beginNotifyAutomationEvent** method of the **ActiveXAutomation** class registers the receiver to be notified when a specified event occurs on an ActiveX automation object. The object that invokes the **beginNotifyAutomationEvent** is referred to as the *subscriber*.

An object that subscribes to an automation notification is notified when the nominated event occurs for that object.

The parameters for this method are listed in the following table.

Parameter	Description
receiver	The object that is to receive the event notification
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

A method implemented by the **eventClassRefName** parameter is executed each time its corresponding automation event occurs.

This event notification continues until the JADE automation object is deleted or until the **endNotifyAutomationEvent** method is called. The **endNotifyAutomationEvent** method has the same signature as the **beginNotifyAutomationEvent** method.

Caution There may be an impact on performance, particularly in JADE thin client mode or on a slow communications link, if you register for large numbers of automation events or events that are triggered often; for example, a cell **change** event in the Excel automation type library. (For details about JADE thin client mode performance, see "JADE Thin Client Performance Considerations", in Appendix A of the *JADE Thin Client Guide*.)

For more details about automation events, see "Using Automation Events", in Chapter 4 of the *JADE External Interface Developer's Reference*.

createAutomationObject

Signature `createAutomationObject(): IDispatch;`

The **createAutomationObject** method of the **ActiveXAutomation** class creates an instance of the ActiveX automation object defined by the receiver. For details about attempting to attach to an instance of the automation server that is already running, see "attachAutomationObject".

Note Your code must explicitly call this method before the first property access or method call. (If you want to specify a remote server or that the ActiveX automation object is executed on the application server, set the appropriate **remoteServerName** or **usePresentationClient** property before you call this method.)

endNotifyAutomationEvent

Signature
```
endNotifyAutomationEvent(receiver:          Object;
                         eventClassRefName: String);
```

The **endNotifyAutomationEvent** method of the **ActiveXAutomation** class terminates a previous **beginNotifyAutomationEvent** method. The parameters for this method, listed in the following table, must be the same as the parameters specified in the **beginNotifyAutomationEvent** method.

Parameter	Description
receiver	The object that is to receive the event notification
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

For more details about automation events, see "Using Automation Events", in Chapter 4 of the *JADE External Interface Developer's Reference*.

getInterface

Signature `getInterface(interface: Class): IDispatch;`

The **getInterface** method of the **ActiveXAutomation** class returns the ActiveX interface specified in the **interface** parameter, if it exists. (As ActiveX interfaces are created as subclasses of the **IDispatch** class when an ActiveX type library is imported, use the Class List of the Class Browser to obtain the names of ActiveX interfaces, if required.) If the specified interface does not exist, a **null** value is returned.

The following example shows the use of the **getInterface** method.

```
displayInterfaceName();
vars
    font      : JadeAutoFont;
    interface : IDispatch;
begin
```

```
    create font;
    font.bold := true;
    interface := font.getInterface(IJadeAutoFont);
    write interface.getName;                    // Outputs IJadeAutoFont
epilog
    delete font;
end;
```

Example of Using an Imported ActiveX Automation Object

The following example shows a Workspace or **JadeScript** method using a Microsoft Excel automation library imported into JADE to load three Excel cells with data, draw a chart, and print the result.

```
chartExample();
vars
    xl     : ExcelApp;      // ActiveX automation subclass
    wrkSht : Worksheet;     // interface subclass of the IDispatch class
    sht    : I_Worksheet;   // interface subclass of the IDispatch class
    rng    : Range;         // interface subclass of the IDispatch class
    chrts  : Sheets;        // interface subclass of the IDispatch class
    chrt   : I_Chart;       // interface subclass of the IDispatch class
begin
    // Start Excel
    create xl;
    xl.createAutomationObject;
    xl.visible := true;                     // See what's going on
    xl.workbooks.add(xl.XlWorksheet);       // Add Workbook (with one sheet)
    sht := xl.activeSheet.I_Worksheet;      // Get top sheet and fill cells
    sht.range("A1", null).putValue("One");
    sht.range("B1", null).putValue("Two");
    sht.range("C1", null).putValue("Three");
    sht.range("A2", null).putValue(10);
    sht.range("B2", null).putValue(5);
    sht.range("C2", null).putValue(3);
    rng := sht.range("A1", "C2");           // Select cells
    chrts := xl.charts;                     // Add a chart
    chrts.add(null, null, null, null);
    chrt := xl.activeChart;
    // Start chart wizard
    chrt.chartWizard(rng,                   // source
                     xl.Xl3DPie,            // gallery
                     7,                     // format
                     xl.XlRows,             // plotBy
                     1,                     // categoryLabels
                     0,                     // seriesLabels
                     2,                     // hasLegend
                     "Jade Example",        // title
                     null,                  // categoryTitle
                     null,                  // valueTitle
                     null);                 // xtraTitle
    // Output chart
    chrt.printOut(null,                     // first page
                  null,                     // last page
                  null,                     // copies
                  null,                     // preview
```

```
                    null,                       // printer
                    null,                       // print to file
                    null);                      // collate
epilog
    if xl <> null then
        xl.activeWorkbook.saved := true;    // Don't ask to save!
        xl.quit;                            // Close down Excel
        delete xl;
    endif;
end;
```

ActiveXInterface Class

The **ActiveXInterface** class is the abstract class that provides a superclass for all ActiveX interfaces imported into JADE.

ActiveX objects are manipulated by their interfaces. An *interface* consists of a set of properties that you can set or get, and a set of methods that can be called. The caller of an ActiveX object needs to know the property types and method names and parameters to make use of these interfaces.

A set of interface classes is generated when you import an ActiveX control or automation type library object. These generated classes map to each of the interfaces defined for that ActiveX object. You then access the functionality of the ActiveX object through these interface classes, by using standard JADE language constructs.

Most ActiveX objects have many different interfaces. The most important of these interfaces is the **IUnknown** interface, which all COM objects implement and all other ActiveX interfaces inherit. Although the **IUnknown** class is the only interface that all objects must support, most objects also support the **IDispatch** interface.

Note You can create neither transient nor persistent instances of **ActiveXInterface** subclasses.

For details about the **IUnknown** subclass of the **ActiveXInterface** class, see "IUnknown Class", later in this chapter. The **IUnknown** class provides the **IDispatch** subclass, in which the interface classes of the imported ActiveX object are created. (For details, see "IDispatch Class", later in this chapter.)

See also "Using ActiveX Control and Automation Server Libraries", in Chapter 4 of the *JADE External Interface Developer's Reference*.

Inherits From: Object

Inherited By: IUnknown

ActiveXInvokeException Class

The **ActiveXInvokeException** class is the transient class that defines behavior for exceptions that occur as a result of accessing an ActiveX property or invoking an ActiveX method.

An ActiveX exception is raised only if an internal exception occurs during a property access or method call of the ActiveX object. All other ActiveX errors are reported as user interface exceptions. For details about the properties defined in the **ActiveXInvokeException** class, see "ActiveXInvokeException Properties", in the following subsection.

Inherits From: UserInterfaceException

Inherited By: (None)

ActiveXInvokeException Properties

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

Property	Contains…
activeXErrorCode	An error number generated by the ActiveX object
description	A textual description of the error
helpContext	An associated context number for the ActiveX object
helpFile	The help file name for the ActiveX object
source	The name of the ActiveX object that caused the error

activeXErrorCode

Type: Integer

The **activeXErrorCode** property of the **ActiveXInvokeException** class contains the error number generated by the ActiveX object on which the exception was raised.

description

Type: String

The **description** property of the **ActiveXInvokeException** class contains a textual description of the error generated by the ActiveX object on which the exception was raised.

helpContext

Type: Integer

The **helpContext** property of the **ActiveXInvokeException** class contains an associated context number for the ActiveX object on which the exception was raised.

This property is used to provide context-sensitive help when accessing the help file of an ActiveX object.

helpFile

Type: String

The **helpFile** property of the **ActiveXInvokeException** class contains the help file name for the ActiveX object.

If this property is not set, no help file is opened.

source

Type: String

The **source** property of the **ActiveXInvokeException** class contains the name of the ActiveX object on which the exception was raised.

Application Class

The **Application** class provides a superclass for all user application classes. Each user application is defined as an instance of the **Application** class. The **Application** class defines standard properties and methods for the running of any application.

Each time you create a new schema, an instance of the **Application** class is created for that schema. When you load a schema from a file and there is application data (in the **.ddb** or **.ddx** form and data definition file), JADE creates an instance of the **Application** class for each application defined in this file.

Each schema also has a subclass of the **Application** class. A transient instance of this class is automatically made available to the runtime copy of the application. To access this transient instance, use the **app** system variable in your method logic; for example, use **app.name** to access the name of the application. This transient instance is unique to a specific copy of the application. Changes made to the properties are retained until the application copy is terminated. (This data is therefore not available to other copies of the application.)

Transient objects that are automatically created by JADE cannot be shared, including the application object and exclusive collections. (For details about specifying the creation of transient objects that can be shared across threads, see "*create* Instruction", in Chapter 1 of the *JADE Developer's Reference*.)

Notes Unpredictable results may occur when several processes concurrently access and modify transient objects that are not shared.

You can remove user-defined applications from a schema, providing that at least one application remains in the schema.

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

Inherits From: Object

Inherited By: RootSchemaApp, user-defined **Application** classes

Application Class Constants

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

Constant	Character or Integer Value
ApplicationType_GUI	'G'
ApplicationType_GUI_No_Forms	'F'
ApplicationType_Non_GUI	'S'
ApplicationType_Non_GUI_Rest	'T'
ApplicationType_Non_GUI_Web	'N'
ApplicationType_Rest_Services	'R'
ApplicationType_Web_Enabled	'W'
ThinClientEncryption_External	2
ThinClientEncryption_Internal	1
ThinClientEncryption_None	0
ThinClientEncryption_SSL	3

For details, see the **applicationType** property or **getThinClientEncryptionType** method.

Application Properties

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

Property	Description
aboutForm	Contains the form displayed when the Help menu **About** command is selected
appVersion	Documentation displayed in the default About box for the application
applicationType	Contains the type of the application
controlSpacing	Contains the number of dialog units between controls on forms in the application
currentLocale	Contains a reference to the locale under which the application is running
currentLocaleInfo	Contains information of the current locale
defaultMdi	Specifies whether forms are created as Multiple Document Interface (MDI) forms
finalizeMethod	Contains a reference to the finalize method for the application
fontBold	Contains the bold font style used for controls in the application
fontName	Contains the default font used for controls in the application
fontSize	Contains the default font size used for controls in the application
formMargin	Contains the number of dialog units required as a margin around the edge of forms
heightSingleLineControl	Contains the number of dialog units for the height of single-line controls
helpFile	Contains the help file name for the application that is running
icon	Contains the default icon for any form that does not have a defined icon
initializeMethod	Contains a reference to the initialize method for the application
mdiCaption	Contains the prefix of the caption for the default MDI frame
mdiFrame	Designates a specific form as being the next MDI frame
mousePointer	Controls the shape of the mouse pointer for all windows of the application
name	Contains the name of the application
printer	Contains the printer object for the application
showBubbleHelp	Specifies whether any bubble help defined for controls is displayed
startupForm	Contains the form initially created and displayed when the application is started
userSecurityLevel	Contains the numeric security level for the current user
webMinimumResponseTime	Contains the maximum time a Web browser user waits before a response must be sent back to that browser user

aboutForm

Type: Form

Availability: Read or write at any time

The **aboutForm** property of the **Application** class contains a reference to the form that is displayed when a user selects the **About** command from the Help menu; that is, the About form.

In the JADE development environment, specify the About form in the Define Application dialog **About Form** combo box. You can also specify the About form at run time, by assigning a persistent form reference. If no About form is assigned, the default About dialog is displayed.

When you use the JADE Painter menu designer to create a standard Help menu that contains an **About** menu item, JADE displays a message box at run time if you do not set the **Application** class **aboutForm** property. This message box displays the following information.

```
---------- Title:
"About Application:  application-name"

---------- Contents:
Application:   application-name

Release: application-release-property

Jade Version: nn.nn.nn
```

This information is displayed on the JADE Web Application monitor About form, for example.

appVersion

Type: String[30]

Availability: Read or write at any time

The **appVersion** property of the **Application** class is set at development time and serves only as documentation for the application.

This property is displayed on the default About form for the application.

applicationType

Type: Character

Availability: Read-only

The **applicationType** property of the **Application** class contains the type of the application.

The application type can be one of the values listed in the following table.

Class Constant	Value	Description
ApplicationType_GUI	G	GUI application type (the default), providing full Windows facilities.

Class Constant	Value	Description
ApplicationType_GUI_No_Forms	F	GUI application that does not display forms; for example, a background client that starts a print task from the **initialize** method and can perform other tasks that do not require the display of forms.
ApplicationType_Non_GUI	S	Application that can run in a client node or a server node. The **initialize** method of a non-GUI application should not use any GUI facilities and should not invoke printing services. A non-GUI application runs on the node on which the **Application** class **startApplication**, **startApplicationWithParameter**, **startAppMethod**, or **startApplicationWithString** method is executed (that is, if this method is executed on a server node, the application is started on that server node). Alternatively, you can start a non-GUI application by using the **ServerApplication** parameter in the [JadeServer] section or the [JadeAppServer] section of the JADE initialization file, which starts the application when the server node is initialized, or you can check the **Run As Server Application** check box in the JADE development environment Run Application dialog.
ApplicationType_Non_GUI_Rest	T	Web services application using HTTP that is based on the Representational State Transfer (REST) architectural style. The Web Application Monitor is not initiated.
ApplicationType_Non_GUI_Web	N	Application that can be accessed from the Internet as a background task, if required. The **Web Options** sheet of the Define Application dialog is then enabled. The start-up form defined for the application is the first Web page that is displayed when the application is invoked from the Web browser. For a Web services application, the type of Web service provided is based on Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). Application features such as Multiple Document Interface (MDI) forms and three-dimensional controls are ignored for Web-enabled applications. In addition, the Web Application Monitor window is not displayed (that is, the Web Application Monitor is not initiated).
ApplicationType_Rest_Services	R	Web services application using HTTP that is based on the Representational State Transfer (REST) architectural style.
ApplicationType_Web_Enabled	W	Application that can be accessed from the Internet, if required. The **Web Options** sheet of the Define Application dialog is then enabled. The start-up form defined for the application is the first Web page that is displayed when the application is invoked from the Web browser. For a Web services application, the type of Web service provided is based on Simple Object Access Protocol (SOAP) and Web Services Description Language (WSDL). Application features such as Multiple Document Interface (MDI) forms and three-dimensional controls are ignored for Web-enabled applications.

Applications of type **ApplicationType_GUI_No_Forms**, **ApplicationType_Non_GUI_Web**, and **ApplicationType_Non_GUI** terminate only after the JADE **terminate** instruction is executed or the application monitor window is closed.

Applications of type **ApplicationType_GUI** and **ApplicationType_Web_Enabled** terminate if no forms remain, if only forms created by using the **GUIClass** class **createPrintForm** method remain, or when the JADE **terminate** instruction is executed.

The **Application** class **startApplication**, **startAppMethod**, **startApplicationWithParameter**, and **startApplicationWithString** methods start only applications of type **ApplicationType_Non_GUI**, **ApplicationType_Non_GUI_Web**, and **ApplicationType_Non_GUI_Rest** if they are invoked from a server method or server application. (An exception is raised if they are invoked from a server method or a server application to start an application of a type other than non-GUI.) On a client node, they start all types of application.

You can use the **MaxWaitAppStart** parameter in the [JadeClient] or [JadeServer] section of the JADE initialization file to increase the time that JADE waits for a **GUI** or **GUI, No Forms** application to initiate on another thread before raising an exception when your system has a large number of applications to start and the default value of 45 seconds may not be sufficient for the loading on the machine during startup. For details, see your *JADE Initialization File Reference*.

Non-GUI applications (that is, those whose **Application** class **applicationType** property is set to **ApplicationType_Non_GUI**, **ApplicationType_Non_GUI_Web**, or **ApplicationType_Non_GUI_Rest**) for standard (fat) clients are run as GUI applications that do not display forms; that is, **ApplicationType_GUI_No_Forms**. Form creation raises an exception if the application is not in exception state.

When running JADE in standard mode, non-GUI applications behave as follows.

- The application displays an **Interrupt** button on the taskbar.
- Creation of a form is permitted while in exception state so that the **Debug** button on the exception dialog functions correctly.
- The JADE executable program (**jade.exe**) does not exit while non-GUI applications are running.

The following example shows a method used to initialize an **ApplicationType_Non_GUI** application.

```
initializeApp();
vars
    today : TimeStamp;
begin
    // Initialize method for a serverAppExample server application.  As
    // server applications must have an ApplicationType_Non_GUI (S)
    // application type and therefore cannot use GUI facilities, initialize
    // and finalize methods must be set.  When the server is started, the
    // serverAppExample application is run and this method is executed.
    write "Server Application Initialized on " & today.String;
end;
```

controlSpacing

Type: Integer

The **controlSpacing** property of the **Application** class contains the number of dialog units between controls on forms in the application. (With dialog units, you do not need to specify both vertical and horizontal spacing, as pixel spacing is a function of the dialog units and the application font.)

currentLocale

Type: Locale

Availability: Read-only

For an active application, the **currentLocale** property of the **Application** class contains a reference to the **Locale** object for the locale under which the application is running. This reference is an instance of **Locale** defined in the current schema, and it provides forms and translatable strings.

The **currentLocale** reference is set when an application is initiated and it is automatically updated if the locale of the workstation on which the application is running is changed.

When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **true**, the locale number of this instance can be different from the value returned by the **Schema** class **getCurrentLocaleId** method if the **setJadeLocale** method has been called with a locale not found in the schema. When the **EnhancedLocaleSupport** parameter is not defined or it is set to **false** and a locale is set using the **setJadeLocale** method of the **Application** class, the **currentLocale** property is changed but the **currentLocaleInfo** property remains unchanged.

Note When you have applications with imported packages, you must use the **Process** class **getProcessApp** method to determine the current locale of the process; that is, by calling **process**.**getProcessApp**.**currentLocale**.

currentLocaleInfo

Type: LocaleFullInfo

Availability: Read-only

The **currentLocaleInfo** property of the **Application** class contains a reference to a **LocaleFullInfo** object that provides information about the current locale.

Note When the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node is set to **false** or it is not specified, 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 enhanced locale support is not set:

- The **currentLocaleInfo** object is constructed during initialization of the application, using the locale information table for the initial locale of the application.
- If the locale is set using the **setJadeLocale** method of the **Application** class, the **currentLocale** property is changed but the **currentLocaleInfo** property is unchanged.

When enhanced locale support is set:

- The **currentLocaleInfo** object is constructed during initialization of the application, using the locale information table for the application initial thread locale. For a JADE thin client process, the information is obtained from the JADE thin client, including regional overrides. This value reflects regional overrides that have been applied to the session locale, if currently in use.
- If the locale is set using the **setJadeLocale** method of the **Application** class, the **currentLocaleInfo** property is changed. If it is called with the **requestedLcid** parameter set to **LCID_SessionWithOverrides**, the instance is changed to include regional settings that are in effect at the time of the method call.

defaultMdi

Type: Boolean

Availability: Read or write at any time

The **defaultMdi** property of the **Application** class specifies whether forms defined with the **mdiChild** property set to **MdiChild_UseAppDefault** (**0**) are created as Multiple Document Interface (MDI) forms. The **defaultMdi** property enables the style of application to be determined by the user.

The **mdiChild** property should be set to **MdiChild_UseAppDefault** (**0**) for all forms that do not specifically have to be MDI or non-MDI forms, so that the style of forms can be provided as a user option for the application.

You can also set this property in a method before a form **create** instruction, to control the style of form that is created when the form has the **mdiChild** property set to **MdiChild_UseAppDefault** (**0**).

finalizeMethod

Type: Method

Availability: Read-only at run time

The **finalizeMethod** property of the **Application** class contains a reference to the *finalize* method for the application; that is, the method that is executed when the application terminates.

Note The **finalize** method is not executed when an application is terminated through the JADE Monitor.

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **Application** class determines the bold font style used for controls in the application when the **fontName** property for a control on a form is set to **Default**. (See also the **Application** class **fontName** and **fontSize** properties.)

Any changes made to the application font attributes at run time are not permanent and affect only the current instance of the application. The application font also defines the default font for graphics text drawing methods. The default value for the **fontBold** property is **false**.

fontName

Type: String[31]

Availability: Read or write at any time

The **fontName** property of the **Application** class determines the default font used for controls in the application when the **fontName** property for a control on a form is set to **Default**. (See also the **Application** class **fontBold** and **fontSize** properties.)

Any changes made to the application font attributes at run time are not permanent and affect only the current instance of the application. The application font also defines the default font for graphics text drawing methods. The default value for the **fontName** property is **Tahoma**.

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property of the **Application** class determines the default font size used for controls in the application when the **fontName** property for a control on a form is set to **Default**. (See also the **Application** class **fontBold** and **fontName** properties.)

Any changes made to the application font attributes at run time are not permanent and affect only the current instance of the application. The application font also defines the default font for graphics text drawing methods.

The default value for the **fontSize** property is **8.25** (that is, 8.25 points).

formMargin

Type: Integer

Availability: Read or write at any time

The **formMargin** property of the **Application** class contains the number of dialog units that are required as a margin around the edge of forms (windows) in the application. (Form margins enable you to position controls so that they do not intrude into the margin of the form.)

heightSingleLineControl

Type: Integer

Availability: Read or write at any time

The **heightSingleLineControl** property of the **Application** class contains the number of dialog units required as the uniform height of button, text box, and combo box controls on forms in the application.

helpFile

Type: String

Availability: Read or write at any time

The **helpFile** property of the **Application** class contains the help file name for the application. This property defaults to the help file name specified in the **Help File** text box of the Define Application dialog in the JADE development environment.

In JADE thin client mode, help file processing is performed on the presentation client.

Although the help file name is usually established at development time, you can use this property to provide access at run time. The file name is not validated. Any changes made to the help file name at run time are not permanent and affect only the current instance of the application. If this property is not set, no help file is opened.

The help file can be Hypertext Markup Language (**.htm** or **.html**) files, an Adobe Acrobat Portable Document Format (.**pdf**) file, a Windows help (.**hlp**) file, or a compiled help (.**chm**) file.

When help is invoked directly via the **Window** class **showHelp** method or via the user pressing the help key (F1), a URL is created and the default browser is invoked to display the URL. For Web-based HTML help, the value of the **helpFile** property is set to a base URL, as shown in the following code fragment examples.

```
app.helpFile := "http://www.example.com/prodhelp";
```

```
app.helpFile := "http://www.example.com/prodhelp/" & app.version.String & "/";
```

icon

Type: Binary

Availability: Read or write at any time

The **icon** property of the **Application** class acts as the default icon for any form that does not have a defined icon. Use this property to specify the default custom icon for any form that can be minimized by the user at run time.

The icon of a form is displayed when the form is minimized and as the Control-Menu icon for JADE forms. JADE creates a large and a small icon for use with a form if they are present in the icon file when the **app.icon** and **form.icon** properties are set.

The icon is specified in the JADE development environment Define Application dialog **Icon** group box or by loading it at run time (by assigning it from another icon or by using the **loadPicture** method). If the icon is loaded from a file, it must have an icon format. If you do not specify a custom icon for a form, the application icon is used.

If no application icon was specified, the JADE default icon is used.

initializeMethod

Type: Method

Availability: Read-only at run time

The **initializeMethod** property of the **Application** class contains a reference to the *initialize* method for the application; that is, the method that is executed when the application starts.

mdiCaption

Type: String

Availability: Read or write at run time only

The **mdiCaption** property of the **Application** class contains the prefix of the caption for the default MDI frame that holds the MDI child forms. The **mdiCaption** string contains the name of the application appended to the MDI child form name. The caption does not apply to an MDI frame defined by a user.

The value of the **mdiCaption** property defaults to the application name. The property can be set and interrogated, regardless of whether an MDI frame is actually running.

mdiFrame

Type: Class

Availability: Read or write at run time only

The **mdiFrame** property of the **Application** class enables you to designate a specific form as being the next MDI frame. Any MDI child forms built after this point are placed in that user MDI frame. If the MDI frame form is not already active as an MDI frame, it is automatically created and displayed when the MDI child form is created and displayed. By default, MDI child forms are placed in a default MDI frame that is automatically supplied by JADE.

If the current **mdiFrame** form is created by your logic, it is built as an MDI frame regardless of the **mdiFrame** property setting of the form. Changing the next MDI frame does not affect any currently active forms. Setting the **mdiFrame** property instance to **null** causes the next MDI frame to return to the use of the JADE-supplied default.

The application can have any number of currently active MDI frames. Although forms that have the **mdiFrame** property set are created as MDI frames regardless of the value of **app.mdiFrame**, only the **mdiFrame** property determines the frame in which MDI child forms are placed.

Notes An MDI frame automatically creates a child client window that covers the non-border area of the frame. The child MDI forms are placed inside this client window. If the MDI frame is defined with controls, the child client window is automatically positioned in an empty area of the MDI frame.

An MDI frame form has sizable borders, regardless of the **borderStyle** property value for the form in Painter.

The **moveMdiClient** form method enables this client window to be positioned as required (for example, below toolbars and above status lines). The **moveMdiClient** method is usually called from the **resize** method of the form.

mousePointer

Type: Integer

Availability: Read or write at run time only

The **mousePointer** property of the **Application** class controls the shape of the mouse pointer for all windows of the application. Setting this property to a non-zero value overrides the **mousePointer** value of each window in the application. Set the **mousePointer** property to a non-zero value to determine the type of mouse pointer that is displayed when the mouse is positioned over any window belonging to the application at run time.

Tip Use this property to display the hourglass cursor (**Window.MousePointer_HourGlass**) when the application is performing an extended operation.

To restore the previous behavior, set the **mousePointer** property to the default value of **Window.MousePointer_Default** (**0**). If you set this property to the **Window** class **mousePointer** property default value, the mouse pointer that is displayed is determined by each individual window.

The settings of the **mousePointer** property are listed in the following table.

Window Class Constant	Value	Description
MousePointer_Default	0	Default value determined by the current window (for example, an arrow or hourglass).
MousePointer_Arrow	1	Arrow ( ).
MousePointer_Cross	2	Cross (cross-hair pointer).
MousePointer_IBeam	3	I-Beam ( ).
MousePointer_Cursor	4	User-defined cursor not provided by the system. (See the **Window** class **mouseCursor** property.)
MousePointer_Size	5	Size (four-pointed arrow pointing north, south, east, west).
MousePointer_NESW	6	Size NE SW (double arrow pointing north east and south west).
MousePointer_NS	7	Size N S (double arrow pointing north and south).
MousePointer_NWSE	8	Size NW SE (double arrow pointing north west and south east).

Window Class Constant	Value	Description
MousePointer_WE	9	Size W E (↔ double arrow pointing west and east).
MousePointer_UpArrow	10	Up arrow (↑).
MousePointer_HourGlass	11	Hourglass, or wait (⌛).
MousePointer_NoDrop	12	No drop (⃠).
MousePointer_Drag	13	Standard JADE drag cursor ( ).
MousePointer_HorizontalLine	14	Cursor used to drag a horizontal line ( ).
MousePointer_VerticalLine	15	Cursor used to drag a vertical line ( ).
MousePointer_HandPointing	16	Cursor showing a hyperlink ( ).

name

Type: String[100]

Availability: Read-only at run time

The **name** property of the **Application** class contains the name specified in the **Name** text box of the Define Application dialog in the JADE development environment.

An instance of the **Application** class is created with this name.

printer

Type: Printer

Availability: Read-only at run time

The **printer** property of the **Application** class contains a reference to the printer object for the application.

The printer object is automatically initialized when the application starts up.

showBubbleHelp

Type: Boolean

Availability: Read or write at run time only

The **showBubbleHelp** property of the **Application** class specifies whether any bubble help defined for controls is displayed. Bubble help is not displayed for a control if the **Window** class **bubbleHelp** property text contains spaces only.

Controls with bubble help text display that text in a bubble below or above the control when the mouse is positioned over the control for more than a half a second. The bubble help is removed when the mouse is moved off the control (see the **mouseLeave** event method) or when a mouse button is pressed. This function occurs only when the application is the active application and that form has focus.

If bubble help is currently displayed and the next window to which the mouse is moved also has bubble help text, there is no delay in the display of the bubble help for the next control. To turn off the display of bubble help or to provide the user with the ability to turn it off, you must supply the appropriate user logic.

See the **ComboBox** class or **ListBox** class in Chapter 2 for details about automatic bubble help that is displayed for combo boxes and list boxes if the combo box or list box does not have bubble help text defined for it by using the **Window** class **bubbleHelp** property.

The settings of the **showBubbleHelp** property are listed in the following table.

Value	Description
true	Bubble help is always displayed (the default)
false	Bubble help is not displayed

This property is ignored when dragging is in progress. (For details about dragging forms or controls, see the **Window** class **dragMode** property.)

startupForm

Type: Form

Availability: Read or write at any time

The **startupForm** property of the **Application** class contains a reference to the form that is initially created and displayed automatically when the application is started up. If the **initialize** method of the application creates a form that is not unloaded, the start-up form is ignored.

The **startupForm** property is specified in the **Startup Form** combo box of the Define Application dialog in the JADE development environment.

Any change to the property at run time is not retained when the application terminates.

userSecurityLevel

Type: Integer

Availability: Read or write at run time

The **userSecurityLevel** property of the **Application** class contains the security level for the current user. The default value is zero (**0**).

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 form.

- If **securityLevelEnabled > app.userSecurityLevel** for a form, control, or menu item, it is automatically disabled, regardless of the value of the **enabled** property.
- If **securityLevelVisible > app.userSecurityLevel** for a control or menu item, its **visible** property is set to **false**.
- If **securityLevelVisible > app.userSecurityLevel** for a form when the **show** or **showModal** method is called, the method call is rejected.

You can subsequently override the values of the **enabled** and the **visible** properties.

You should set the **userSecurityLevel** property during the **initialize** method or **getAndValidateUser** process before the creation of a form, as the setting of this property is actioned when forms and controls are created.

Changing the value of **app**.**userSecurityLevel** does not change the behavior of forms that have already been loaded.

Changing the value of the **securityLevelEnabled** property of a form, control, or menu item causes a reevaluation of its enabled status, based on the above rules.

Changing the value of the **securityLevelVisible** property of a form, control, or menu item causes a reevaluation of its visible status, based on the above rules.

webMinimumResponseTime

Type: Integer

Availability: Read or write at any time

The **webMinimumResponseTime** property of the **Application** class contains the maximum time a Web browser user has to wait before a response must be sent back to that user from the JADE application, triggered by a timer event. The minimum response time value represents the time in seconds.

By default, requests do not time out; that is, the default value of zero (**0**) indicates infinity.

When the timer event occurs, a default message is sent back to the browser and the current request is terminated. You can override the default message that is sent to the browser user by reimplementing the **Application** class **minimumResponseTimeExceededMsg** method in the **Application** subclass of your user-defined schema.

Note As it uses a timer event, JADE relies on you relinquishing control (by calling the **doWindowEvents** method). If your logic is in a tight loop, for example, the timer event may be unable to be executed at the required interval.

You can set the value of this property dynamically at run time or you can set it from the JADE development environment by using the **Minimum Response Time** text box in the **Web Options** sheet of the Define Application dialog.

Application Methods

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

Method	Description
activateApp	Activates the current form of the application if that application is running within the same copy of **jade.exe**
activeControl	Returns the control that currently has the focus
activeForm	Returns the form that currently has the focus
actualTime	Returns the current date and time as a timestamp value
actualTimeAppServer	Returns the current date and time of the application server
actualTimeServer	Returns the current date and time of the server node
actualTimeStampOffset	Returns the current date, time, and offset from UTC as a timestamp value

Method	Description
actualTimeStampOffsetAppServer	Returns the current date, time, and offset of the application server
actualTimeStampOffsetServer	Returns the current date, time, and offset of the server node
alert	Plays a waveform sound
asyncFinalize	Finalizes a worker application that has processed asynchronous method calls
asyncInitialize	Initializes a worker application to receive and process asynchronous method calls
beep	Plays a waveform sound
checkPictureFile	Checks the file contents to determine whether it contains a valid picture image
clock	Returns a relative time in milliseconds
clearWriteWindow	Clears the contents of the Jade Interpreter Output Viewer if it is open
closeWriteWindow	Closes the Jade Interpreter Output Viewer if it is open
computerName	Returns the name of the workstation on which the current method is executing
computerNameAppServer	Returns the name of the application server workstation when the application is running in JADE thin client mode
copyImageFromClipboard	Returns the binary image contained in the Windows clipboard
copyImageToClipboard	Copies the specified binary image to the Windows clipboard
copyStringFromClipboard	Returns the text contained in the Windows clipboard
copyStringToClipboard	Copies the specified string to the Windows clipboard
createSessionErrorMessage	Returns the message displayed on Web browsers when a Web session cannot be created
currentUTCBias	Obtains the current bias (in minutes) for local time translation on the specified node
dbPath	Obtains the database path
dbServerComputerName	Returns the name of the database server
debugApplication	Starts the specified application in debug mode
debugApplicationWithParameter	Starts the specified application in debug mode and passes an object parameter to the **initialize** method
defaultLocale	Returns the locale used if an attempt is made to run the application on a workstation operating in a locale not supported by the schema
doWindowEvents	Processes all pending Windows events for the application
enableThinClientConnBalancing	Enables balancing of connections from thin clients across a group of application servers
endOdbcSession	Called in an ODBC server application when an ODBC session is closed

Method	Description
executeMethodNotFoundMessage	Returns a default HTML string to a Web browser user when the method specified for execution cannot be found or it is invalid
finalize	Performs any terminate function common to all application users
finalizeOdbcSelect	Called in an ODBC server application at the end of the execution of an SQL Query
flushThinClient	Causes any commands queued in the application server for the JADE thin application to be passed immediately to the presentation client for processing
forms	Returns the form at the specified form number
formsCount	Returns the current number of active forms for the application
generateUuid	Generates and returns a binary Universally Unique Identifier (UUID) value
getApplicationSkin	Returns a reference to the **JadeSkinApplication** object currently set for the application
getCurrentSession	Returns a reference to the **WebSession** object of the specified Web session identifier
getCurrentSessionId	Returns a string of up to 16 characters that identifies the current Web session
getExternalDatabase	Returns a reference to the shared external database transient instance
getForm	Returns the first active form object with the specified name
getEnhancedLocaleSupport	Returns the current value (**true** or **false**) of the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node
getIniFileName	Returns the full path and file name of the JADE initialization file
getIniFileNameAppServer	Returns the name of the application server initialization file when the application is running in JADE thin client mode
getJadeInstallDir	Returns the directory in which the JADE binaries are installed
getJadeInstallDirAppServer	Returns the directory in which the JADE binaries are installed on the application server when the application is running in JADE thin client mode
getJadeTextEditGlobalSettings	Returns a string containing the global settings table compiled into the **JadeTextEdit** class library
getJadeTextEditOneSetting	Returns a string containing the specified **JadeTextEdit** class setting
getMessageText	Returns the error text associated with an exception with the specified error number
getMouseMoveTime	Returns the current **mouseMove** event time in use for the current application running on presentation clients
getOdbcSessionObject	Returns a reference to an application-maintained ODBC session object when called in an ODBC server application

Method	Description
getParamListTypeEntry	Returns the value of the parameter in the parameter list at the specified position
getParamListTypeLength	Returns the number of entries in a **ParamListType** pseudo type parameter list
getPersistentApp	Returns a reference to the persistent application from which the transient instance of the receiver was created
getProcess	Returns the current process
getProfileString	Retrieves a string from the specified section in an initialization file
getProfileStringAppServer	Retrieves a string from the specified section in an initialization file on the application server when the application is running in JADE thin client mode
getRootSchemaFormTranslation	Returns the control captions to be displayed on print-related system forms
getSchema	Returns the current schema
getSessionTimeout	Returns the Web session timeout value specified for the application
getSkin	Returns a reference to the **JadeSkin** object that is currently set for the application
getSkinCollection	Returns the global collection of skins
getSystemVersion	Returns a string containing the version of the JADE system
getTempDir	Returns the temporary directory on the client node
getTempDirAppServer	Returns the temporary directory on the application server when the application is running in JADE thin client mode
getThinClientEncryptionType	Returns the type of encryption being used by the thin client TCP connection for this application
getTransientDbPath	Returns a string containing the full path of the transient database file on the current node
getUTCTime	Returns the current UTC time for the machine on which the method executes
getWebMachineName	Returns the machine name to be used when generating HTML pages
getWebVirtualDirectory	Returns the virtual directory to be used when generating HTML pages
globalLockException	Provides a generic lock exception handler
htmlPageNotFoundMessage	Returns a string containing the error message that is sent to the receiver when the requested page for the Web application is not found
inactiveTimeout	Resets the inactive timeout period to zero (**0**) if there has been no user activity within the number of seconds specified by the **setInactiveTimeoutPeriod** method
initialize	Performs any initialization function common to all application users

Method	Description
initializeOdbcSelect	Called in an ODBC server application before the execution of an SQL query
invalidWebSessionMessage	Returns an HTML string for display on the Web browser when a session is invalid
isActiveXClassIdRegistered	Returns **true** if the specified ActiveX class is registered
isAppRunning	Indicates if the application is running within the same copy of **jade.exe**
isBeingDebugged	Returns **true** if the application is being run through the JADE debugger
isControlSupported	Returns **true** if the current presentation client supports the specified control type
isFormOpen	Returns **true** if a form of the specified class is open
isMultiUser	Returns **true** if the application is running in multiuser mode
isUnicode	Returns **true** if the application is running with Unicode characters and strings
isValidObject	Establishes if the object specified in the **obj** parameter exists
isWebService	Specifies whether the application is running as a Web service
jadeReportWriterAppName	Returns a string containing the name of the application
jadeReportWriterParamLiteral	Returns the literal to be reported when a JADE Report Writer parameter has had its "ignore in selection option" set on
jadeReportWriterParamObjects	Returns an array of objects to populate a list for selection when running a report with an object parameter
jadeReportWriterParameterIsSet	Updates any transient parameter instance holding the current parameter value
jadeReportWriterTimeDetails	Records JADE Report Writer timings
jadeWebServiceInputError	Enables you to log a message and return the SOAP exception text if the input to a Web service request is not encoded as valid UTF8
licencesExceededMessage	Returns the message that is displayed on Web browsers when your licences have been exceeded
loadPicture	Loads a picture (icon, bitmap, Jpeg, meta, png, tiff, or gif) from an external picture file
minimumResponseTimeExceededMsg	Returns a default HTML string to a Web browser user when the maximum wait time for a response is exceeded
msgBox	Displays a message in a dialog, and waits for the user to click a button
odbcWorkerFinalize	Finalizes an ODBC server application
odbcWorkerInitialize	Initializes an ODBC server application from information in a configuration file

Method	Description
paintIfRequired	Causes all forms of the application to be repainted if a repaint is required
playSound	Plays the specified .**wav** file and returns when the sound file has been played
playSoundAsync	Starts playing the specified .**wav** file and returns immediately
productionMode	Returns **true** if the database from which the application is runs has production mode set
random	Returns a random non-negative number in the range **0** through the value of the range **parameter**, inclusive, but not exceeding 32,767
random31	Returns a non-negative number in the range **0** through the value of the **range** parameter, inclusive
relativeMachineMicros	Returns a high-accuracy machine-relative time in microseconds
relativeMachineTime	Returns a high-accuracy machine-relative time in milliseconds
removeSessionMessage	Returns the message that is displayed on Web browsers when your Web session ends
repairCollection	Removes invalid object references and fixes up dictionary keys
rpsDataPumpFinalize	Performs termination functions for a user-defined RPS **Datapump** application
rpsDataPumpInitialize	Performs initialization functions for a user-defined RPS **Datapump** application and registers a receiver for callbacks required to control output to an RDBMS database
seedRandom	Initializes the random number generator or sets the random number generator with a new value
serverName	Returns the name of the server in use for the JADE application
setApplicationSkin	Defines the **JadeSkinApplication** object used for the application at run time
setEndpointForWebService	Redirects a Web service request received by a *gateway* Web service application to another Web service application
setInactiveTimeoutPeriod	Establishes a one-shot timeout period for user activity in a GUI application
setJadeLocale	Changes the locale from within the logic of the application
setMouseMoveTime	Sets the current **mouseMove** event time for the current application running on presentation clients
setOdbcSessionObject	Sets a reference to an application-maintained ODBC session object when called in an ODBC server application
setProfileString	Copies a string into the specified section of the JADE initialization file
setProfileStringAppServer	Copies a string into the specified section of the JADE initialization file on an application server when the application is running in JADE thin client mode

Method	Description
setSessionTimeout	Dynamically sets the period in minutes at which the Web session ends if no requests have been received within that time
setSkin	Defines the **JadeSkin** object used for the application at run time
setStatusLineDisplay	Dynamically changes scrolling text displayed in the Web browser status line
setWebMachineName	Sets the machine name to be used when generating HTML pages
setWebVirtualDirectory	Sets the virtual directory to be used when generating HTML pages
skinDelete	Deletes all entities that were loaded as part of the skin load process
skinExtract	Extracts skin images for a specified application skin to a directory structure
skinLoad	Loads skin images from the specified directory
skinMakeDirectory	Creates an empty directory structure into which skin images can be loaded
startAppMethod	Initiates another application in the same node, enabling you to specify the **initialize** method and pass a parameter to it
startApplication	Initiates another application in the same node
startApplicationWithParameter	Initiates another application in the same node and passes an object parameter to the **initialize** method
startApplicationWithString	Initiates another application in the same node and passes a string parameter to the **initialize** method
startOdbcSession	Called in an ODBC server application when an ODBC session is started
timedOutSessionMessage	Returns the message that is displayed on Web browsers when your Web session times out
updateJadeTextEditAppSettings	Adds or modifies one or more **JadeTextEdit** class settings associated with the current application
userName	Returns the name of the current user as a string
webApplicationDirectory	Returns the name of the Web application directory that contains transferred files when your JADE environment is behind a firewall

activateApp

Signature

```
activateApp(schemaName: String;
            appName:    String): Boolean;
```

The **activateApp** method of the **Application** class returns **true** if the application specified by the **schemaName** and **appName** parameters is running within the same copy of **jade.exe**. This method returns **false** if the specified application is not running.

Use the **schemaName** parameter to specify the schema for the application. If the application is running within the same copy, it activates the current form of the application; that is, it brings it to the top.

When **app.activateApp** is called for an application that is currently minimized, the window for that application is restored. See also the **Application** class **isAppRunning**, **startApplication**, and **startApplicationWithParameter** methods.

activeControl

Signature `activeControl(): Control;`

The **activeControl** method of the **Application** class returns a reference to the control that currently has the focus in the application that is running.

This method returns a null value if the focus is not on a control of a form of the application.

activeForm

Signature `activeForm(): Form;`

The **activeForm** method of the **Application** class returns a reference to the form that currently has focus in the receiver application. This method returns a null value if no form in the application currently has focus.

actualTime

Signature `actualTime(): TimeStamp;`

The **actualTime** method of the **Application** class returns the current date and time as a timestamp.

In JADE thin client mode, this method returns a reference to the date and time relative to the presentation client. (To return the current date and time of the application server, use the **actualTimeAppServer** method.)

To return the date and time relative to the server node, use the **actualTimeServer** method instead of calling **app.actualTime** with the **serverExecution** method option, which is less efficient.

actualTimeAppServer

Signature `actualTimeAppServer(): TimeStamp;`

The **actualTimeAppServer** method of the **Application** class returns the current date and time of the application server as a timestamp value. You can use this method, for example, to get the date and time that an event occurred on the application server for logging purposes. (In JADE thin client mode, calling **app.actualTime** returns the date and time relative to the presentation client, which may be in a different time zone.)

If the application is not running in JADE thin client mode, this method is equivalent to the **actualTime** method. Use the **actualTimeServer** method to return the current date and time of the server node if the application is not running in JADE thin client mode, as this is more efficient than calling the **actualTime** method with the **serverExecution** method option.

actualTimeServer

Signature `actualTimeServer(): TimeStamp;`

The **actualTimeServer** method of the **Application** class returns the current date and time of the server node as a timestamp value.

You can use this method, for example, to get the date and time that an event occurred on the server node for logging purposes.

Tip Although you can use this method when running in JADE thin client mode, it is more efficient to use this method to return the current date and time of the server node when the application is *not* running in JADE thin client mode instead of calling the **actualTime** method with the **serverExecution** method option.

Use the **actualTime** method to return the current date and time relative to the current node or to the presentation client when running the application in JADE thin client mode. Alternatively, call the **actualTimeAppServer** method to return the current date and time of the application server when the application is running in JADE thin client mode.

actualTimeStampOffset

Signature `actualTimeStampOffset(): TimeStampOffset;`

The **actualTimeStampOffset** method of the **Application** class returns the current date, time, and offset from Coordinated Universal Time (UTC) as a timestamp offset value.

In JADE thin client mode, this method returns a reference to the date, time, and offset from UTC relative to the presentation client. (To return the current date, time, and offset from UTC of the application server, use the **actualTimeStampOffsetAppServer** method.)

To return the date, time, and offset from UTC relative to the server node, use the **actualTimeStampOffsetServer** method instead of calling **app**.**actualTimeStampOffset** with the **serverExecution** method option, which is less efficient.

actualTimeStampOffsetAppServer

Signature `actualTimeStampOffsetAppServer(): TimeStampOffset;`

The **actualTimeStampOffsetAppServer** method of the **Application** class returns the current date, time, and offset from Coordinated Universal Time (UTC) of the application server as a timestamp offset value. You can use this method, for example, to get the date, time, and offset from UTC that an event occurred on the application server for logging purposes. (In JADE thin client mode, calling **app.actualTimeStampOffset** returns the date, time, and offset from UTC relative to the presentation client, which may be in a different time zone.)

actualTimeStampOffsetServer

Signature `actualTimeStampOffsetServer(): TimeStampOffset;`

The **actualTimeStampOffsetServer** method of the **Application** class returns the current date, time, and offset from Coordinated Universal Time (UTC) of the server node as a timestamp offset value. You can use this method, for example, to get the date, time, and offset from UTC that an event occurred on the server node for logging purposes.

Tip Use the **actualTimeStampOffsetAppServer** method to return the current date, time, and offset from UTC on an application server node. Use the **actualTimeStampOffset** method to return the current date, time, and offset from UTC on a presentation client or a standard client node.

alert

Signature `alert(soundName: Integer);`

The **alert** method of the **Application** class plays the waveform sound specified in the **soundName** parameter. In JADE thin client mode, this method always executes on the presentation client.

Tip Use this method in **LockException** handlers rather than the **Global**::**alert** method, to avoid *Object not available* exceptions occurring when global is locked.

Waveform sounds for sound types are identified by entries in the **Sounds** section of the registry.

Note Assign sounds to system events by using the **Sounds and Multimedia** program item of the standard Windows Control Panel.

You can use the **Sounds** category global constant values, listed in the following table, in the **soundName** parameter.

Global Constant	Integer Value
Snd_Asterisk	#40
Snd_Beep	-1
Snd_Default	0
Snd_Exclamation	#30
Snd_Hand	#10
Snd_Question	#20

asyncFinalize

Signature `asyncFinalize() updating;`

The **asyncFinalize** method of the **Application** class deletes internal transient objects used in processing asynchronous method calls.

You must call the **asyncFinalize** method to ensure clean shutdown of the underlying message queuing mechanism when your process terminates.

The method is typically called from the **finalize** method for a worker application that processes asynchronous method calls.

asyncInitialize

Signature `asyncInitialize() updating;`

The **asyncInitialize** method of the **Application** class constructs the internal structures required to process asynchronous method calls and prepares the application to receive and process requests.

The method is typically called from the **initialize** method for a worker application that processes asynchronous method calls. The following example shows an initialization method for a worker application coded in the **Application** subclass of the user schema.

```
initializeSearchWorker() updating;
begin
    app.asyncInitialize; // Turns application into asynchronous worker app
end;
```

beep

Signature `beep();`

The **beep** method of the **Application** class plays the .**wav** file associated with the **Default Beep** option (specified in the **Sound Events** list box on the **Sounds** sheet of the Sounds and Multimedia Properties dialog accessed by using the **Sounds and Multimedia** program item of the standard Windows Control Panel) of the current locale.

Tip Use this method in **LockException** handlers rather than the **Global**::**beep** method, to avoid *Object not available* exceptions occurring when global is locked.

Use this method to sound the beep at the workstation of the user who invoked the method.

Note The beep alert is sounded on a workstation regardless of whether a sound card is installed.

The following example shows the use of the **beep** method.

```
clock1_alarmSound(pClock: Clock) updating;
vars
    count : Integer;
begin
    foreach count in 1 to 10 do
        app.beep;
        app.doWindowEvents(100);
    endforeach;
end;
```

checkPictureFile

Signature `checkPictureFile(fileName: String): Integer;`

The **checkPictureFile** method of the **Application** class checks the contents of the file specified in the **fileName** parameter to determine whether it contains a valid picture image.

Note In JADE thin client mode, this method always refers to a file on the presentation client.

The return values from this method are listed in the following table.

Window Class Constants	**Integer**	**Picture Type**
PictureType_None	0	Not a valid picture
PictureType_Bitmap	1	Bitmap
	2	Not used
PictureType_Icon	3	Icon
PictureType_MetaFile	4	Metafile
PictureType_Cursor	5	Cursor
PictureType_Tiff	6	Tag Image File Format (.**tif** file)
PictureType_Jpeg	7	Joint Photographic Experts Group (JPEG)
PictureType_Jpeg2000	10	Joint Photographic Experts Group (JPEG 2000)

Window Class Constants	Integer	Picture Type
PictureType_Png	8	Portable Network Graphics (**png**)
PictureType_Gif	9	Graphics Interchange Format (.**gif** file)

The following example shows the use of the **checkPictureFile** method.

```
vars
    helix : String;
    icon  : Binary;
begin
    helix := " c:\jade\bin\Jade.bmp";
    if app.checkPictureFile(helix) = Window.PictureType_Bitmap then
        icon := app.loadPicture(helix);
        write icon.display;
    endif;
end;
```

clock

Signature `clock(): Integer;`

The **clock** method of the **Application** class returns the relative time in milliseconds since the operating system was started on the workstation that is executing the method.

Note As the return value is limited by the size of an integer, the value returns to zero (**0**) after approximately 24 days.

This method provides a mechanism to obtain deltas (or differences) in time, taken at different points of execution; for example, as a means of timing the execution of a logic sequence.

The following example shows the use of the **clock** method.

```
bttnClientExec_click(btn: Button input) updating;
begin
    // Invokes a method that updates 5000 entries in a collection
    // and executes on the client node.  The time taken by the method
    // is recorded using the app.clock method.
    startTime := app.clock;
    self.executeOnClient;
    textBoxDuration.text := ((app.clock - startTime)/1000).String &
                            "seconds";
end;
```

clearWriteWindow

Signature `clearWriteWindow();`

The **clearWriteWindow** method of the **Application** class clears the contents of the Jade Interpreter Output Viewer if it is open. If the Jade Interpreter Output Viewer is not open, this method is ignored.

closeWriteWindow

Signature `closeWriteWindow();`

The **closeWriteWindow** method of the **Application** class closes the Jade Interpreter Output Viewer if it is open. If the Jade Interpreter Output Viewer is not open, this method is ignored.

computerName

Signature `computerName(): String;`

The **computerName** method of the **Application** class returns the name of the device that executes the current method.

The code fragment in the following example shows the use of this method.

```
// Log the current time and the computer and user names
logFile.open;
logFile.writeLine(app.actualTime.String & " " & app.computerName & " " &
                  app.userName);
logFile.close;
```

In JADE thin client mode, this method returns a reference to the name of the presentation client workstation.

Use the **computerNameAppServer** method to return a reference to the name of the application server workstation.

Tip If the method that calls this method has the **serverExecution** method option, write a method with the **clientExecution** method option that calls this **computerName** method if you require the name of the workstation on which the application is running.

computerNameAppServer

Signature `computerNameAppServer(): String;`

The **computerNameAppServer** method of the **Application** class returns the computer name of the application server when the application is running in JADE thin client mode.

If the application is *not* running in thin client mode, this method functions like the **Application** class **computerName** method; that is, it returns the name of the workstation on which the application is running. The code fragment in the following example shows the use of the **computerNameAppServer** method.

```
if isMultiUser then
    return dbServerComputerName;
endif;
return computerNameAppServer;
```

copyImageFromClipboard

Signature `copyImageFromClipboard(): Binary;`

The **copyImageFromClipboard** method of the **Application** class returns the binary image contained in the Windows clipboard.

This method performs the following actions to copy an image from the Windows clipboard if it is available.

1. Checks whether a non-standard clipboard format called **PNG** is available. If it is available and the content is a .**png** file, a copy of that image is returned.
2. Checks whether an image in the standard Microsoft Windows **CF_DIB** format is available. If so, that image is copied from the clipboard and returned as a bitmap-type (.**bmp**) image.
3. Checks whether an image in the standard Microsoft Windows **CF_ENHMETAFILE** (enhanced meta file) format is available. If so, that image is copied from the clipboard and returned.

If the clipboard is locked by another Windows process (which is not a normal occurrence) or an image in any of the supported formats is not available, this method returns a null binary.

copyImageToClipboard

Signature `copyImageToClipboard(image: Binary): Integer;`

The **copyImageToClipboard** method of the **Application** class copies the binary image specified in the **image** parameter to the Windows clipboard. The image must be a .**bmp**, .**jpg**, .**jp2**, .**tiff**, .**gif**, .**png**, or enhanced meta file type image.

Except for the enhanced meta file type, JADE converts the file to a bitmap image and pastes it to the clipboard using the standard Windows clipboard style of **CF_DIB**. (There are no standard format types available for the other file types.)

An enhanced meta file type is added to the clipboard as a standard Microsoft Windows **CF_ENHMETAFILE** clipboard style.

If the image is a .**png** file, an additional clipboard entry is made under a non-standard clipboard format named **PNG**. This format is used and accepted by some other applications.

Note For the Windows **CF_DIB** style, Microsoft Windows usually creates additional clipboard copies of the image under other styles.

If the copy to the clipboard succeeded, the returned value from the method call is zero (**0**).

If the copying to the clipboard fails, one of the following errors is returned. (Method failure does not generate an exception.)

- 14015 (*File does not contain an image type that can be handled*), if the image is not one of the permitted styles or the image is invalid. (There is no standard clipboard style for icons or cursors.)
- 1042 (*Invalid handle*), if the clipboard is locked by some other Windows process. (This is not a normal occurrence.)

Note The rules regarding retrieving transparent images from the clipboard are vague, and applications have different implementations. The result is that the transparency can be lost by some applications, but not by JADE. For that reason, JADE adds a **PNG** clipboard entry for .**png** files so that transparency can be always retained if that format is used.

copyStringFromClipboard

Signature `copyStringFromClipboard(): String;`

The **copyStringFromClipboard** method of the **Application** class returns the text contained in the Windows clipboard.

If the clipboard is locked by another process or the clipboard does not contain text, a null string is returned.

copyStringToClipboard

Signature `copyStringToClipboard(str: String): Boolean;`

The **copyStringToClipboard** method of the **Application** class copies the string specified in the **str** parameter to the Windows clipboard.

If the process succeeds, this method returns **true**. If the process fails, it returns **false**. The action may fail if some other process has the clipboard locked.

createSessionErrorMessage

Signature `createSessionErrorMessage(): String;`

The **createSessionErrorMessage** method of the **Application** class displays a message on a Web browser when a Web session cannot be created by an application deployed in HTML thin client mode. The following default response is returned to the Web browser.

```
Session Error
Session could not be established. Please try your operation again.
```

Reimplement the **createSessionErrorMessage** method if you want to display a different session error message on the Web browser. The returned string should be in HTML format, for correct rendering on the browser.

currentUTCBias

Signature `currentUTCBias(location: Integer): Integer;`

The **currentUTCBias** method of the **Application** class returns the current bias, in minutes, for local time translation on the node specified in the **location** parameter.

The bias is the difference in minutes between Coordinated Universal Time (UTC) and local time (that is, **bias = UTC - 'local time'**). As the bias is current, it includes any daylight saving adjustment in effect at the time the value is obtained.

The **location** parameter values, provided by global constants in the **ExecutionLocation** category, are listed in the following table.

Global Constant	Integer Value	Method is executed…
CurrentLocation	0	In the current location
DatabaseServer	1	On the database server node
PresentationClient	2	On the presentation client (applicable to applications running in thin client mode)

The current bias defaults to **CurrentLocation** (**0**) if you specify any value other than those listed in this table. See also the **Application** class **getUTCTime** method and the **TimeStamp** primitive type **localToUTCTime**, **localToUTCTimeUsingBias**, **utcToLocalTimeUsingBias**, and **utcToLocalTime** methods.

dbPath

Signature `dbPath(): String;`

The **dbPath** method of the **Application** class returns the database path. The database path value corresponds to the database path that must be included in the program target (that is, the command line) when running the application. For details, see "JADE Configurations", in Chapter 1 of the *JADE Installation and Configuration Guide*.

Tip If your application requires an absolute path, ensure that you specify an absolute path in the **path** parameter of the program shortcut that starts the node.

You may require the database path when opening external files or when opening a common File Open dialog, as shown in the following example.

```
vars
    fopen : CMDFileOpen;
    name  : String;
begin
    create fopen;
    fopen.initDir := app.dbPath;          // set the initial directory
    if fopen.open = 0 then                // not canceled and no error
        name := fopen.fileTitle;          // use the returned value
    endif;
epilog
    delete fopen;                         // tidy
end;
```

dbServerComputerName

Signature `dbServerComputerName(): String;`

The **dbServerComputerName** method of the **Application** class returns the computer name of the database server.

debugApplication

Signature
```
debugApplication(schemaName:      String;
                 applicationName: String);
```

The **debugApplication** method of the **Application** class starts the application specified by the **applicationName** and **schemaName** parameters in debug mode.

An exception is raised if the JADE development environment is not already running in the same session (using the same **jade.exe** environment.

debugApplicationWithParameter

Signature
```
debugApplicationWithParameter(schemaName:      String;
                              applicationName: String;
                              passedObject:    Object);
```

The **debugApplicationWithParameter** method of the **Application** class starts the application specified in the **applicationName** parameter in JADE debug mode, passing the value specified in the **passedObject** parameter to the **initialize** method defined in the application. The **initialize** method must expect an object parameter; otherwise an exception is raised.

The JADE debugger stops on the first logic statement executed in the user application.

The conditions that apply to the **Application** class **debugApplication** method apply to the method; that is:

- To debug the application, the JADE development environment must be running.
- You must have a defined user profile in the JADE development environment.
- The schema specified in the **schemaName** parameter must exist.
- The application must be defined in that schema or a superschema.

defaultLocale

Signature `defaultLocale(): Locale;`

The **defaultLocale** method of the **Application** class returns a reference to the **Locale** object for the locale to be used if an attempt is made to run the application on a workstation operating in a locale that is not supported by the schema of the application.

If this method returns a **null** reference, JADE assumes that the first available locale in the schema is the default locale.

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**.

doWindowEvents

Signature `doWindowEvents(waitTime: Integer);`

The **doWindowEvents** method of the **Application** class processes all queued messages for the application until one of:

- The queue is empty *and* the period in milliseconds specified by the value of the **waitTime** parameter has expired.
- The **waitTime** period has expired and other callback-type messages (such as timers or notifications) are queued (the first one is always processed). In this case, paint requests and possibly user requests remain unprocessed.

When the **doWindowEvents** method has processed all pending Windows events, it then waits and processes any further Windows events that arrive until the specified time from when the method was initiated has expired.

Note You can call the **doWindowEvents** method from any type of application (for example, from a non-GUI application to allow the processing of timers and notifications) or from a server method.

For example, the **doWindowEvents** method allows:

- Any waiting timer and notification events to be processed.
- The Windows environment to respond to other actions; for example, keyboard or mouse clicks.

Do *not* use the **doWindowEvents** method in the following situations.

- When causing a repaint of a window. Use the **refreshNow** method to repaint a window.
- When allowing a **Cancel** button to be clicked during a lengthy process. Use the **Window** class **doWindowEvents** method.

- When involved in the processing of ActiveX controls and OLE objects.

  As the OLE object processes requests synchronously using Windows events, the **Application** class **doWindowEvents** method can cause asynchronous processing to be attempted, with resulting failure.

Call the **doWindowEvents** method from a server method to process server notifications and timers.

Caution Indiscriminate use of this method can cause unwanted side effects. For example, it can change the order of Windows event processing and can allow users to click on other controls, menus, or forms that could have an impact on the current process.

It can also cause recursive loops. For example, if a **keyDown** event calls a **doWindowEvents** method and the user is holding down the key, that method will invoke another **keyDown** event, and so on. JADE handles this situation by discarding messages for a specific window if there are already five such messages in the call stack.

Use of **app.doWindowEvents** can be dangerous because it can result in recursive calls and can mean events are processed out of order. It is also an expensive operation under thin client mode. Its use should be very restricted and only when the consequences are clearly understood.

The following example shows the use of the **doWindowEvents** method.

```
queryUnload(cancel: Integer io; reason: Integer) updating;
begin
    self.endTimer(101);               // ticker tape timer
    app.doWindowEvents(0);
    if self.bTotalWorth.value then
        self.endTimer(Graph_Timer);
        app.doWindowEvents(0);
    endif;
end;
```

See also the **Process** class **sleep** method. For more details, see "Windows Events and JADE Events", in Chapter 2.

enableThinClientConnBalancing

Signature `enableThinClientConnBalancing();`

The **enableThinClientConnBalancing** method of the **Application** class enables connections from presentation clients to be balanced across a group of application servers sharing the same value of the **AppServerGroupName** parameter in the [JadeAppServer] section of the JADE initialization file.

This method must be run successfully by at least one JADE process in the application server node. The application server is registered with the database server and the process becomes a redirection assistant. When all processes in an application server that are marked as redirection assistants terminate, the application server stops redirecting presentation clients.

If the method does not run successfully, exceptions are raised to report configuration errors (for example, a blank value for the **AppServerGroupName** parameter), and information is output to the JADE message log file. If the method runs successfully, an entry in the JADE message log file records that connection balancing is enabled.

Tip Call the **enableThinClientConnBalancing** method in the initialization method of a server application started by specifying the **ServerApplication** parameter in the [JadeAppServer] section of the JADE initialization file.

endOdbcSession

Signature `endOdbcSession(sessionObject: Object);`

The **endOdbcSession** method of the **Application** class is in an ODBC server application called when a session is closed. The value of the **sessionObject** parameter is the session object set by the application using the **setOdbcSessionObject** method or it is a null reference.

You can reimplement this method, if required.

executeMethodNotFoundMessage

Signature `executeMethodNotFoundMessage(): String;`

The **executeMethodNotFoundMessage** method of the **Application** class returns a default HyperText Markup Language (HTML) string that is sent to a Web browser user when the method specified for execution cannot be found or it is invalid.

You can override this method in the **Application** class of your user-defined schemas. However, it is your responsibility to ensure that the returned string is a correctly formatted HTML string if you override this method.

finalize

Signature `finalize() updating;`

The **finalize** event of the **Application** class is the default event that is called by the application before the close request of the application is invoked if you do not define your own **finalize** event to perform any function that is common to all users of this application.

Note The **finalize** event is performed once for each user of the application.

For non-GUI applications, this event method is replaced by the user-specified **finalize** method specified when the application is defined. (For details, see "Passing Parameters to non-GUI Applications using jadclient", in Chapter 1 of the *JADE Runtime Application Guide*.)

An exception that occurs during the **finalize** method is not displayed and is written to the **jommsg.log** file, unless it is processed by an active exception handler.

finalizeOdbcSelect

Signature
```
finalizeOdbcSelect(rv:       RelationalView;
                   username: String);
```

The **finalizeOdbcSelect** method of the **Application** class is called in an ODBC server application after executing an SQL query. The **rv** parameter specifies the relational view currently in use and the **username** parameter specifies the user code of the logged-on user.

You can reimplement this method, if required; for example, to clean up transient objects created during the execution of the query.

flushThinClient

Signature `flushThinClient();`

The **flushThinClient** method of the **Application** class causes any commands queued in the application server for the JADE thin application to be passed immediately to the presentation client for processing.

Queued commands are normally passed to the presentation client when the current Graphical User Interface (GUI) process that generated the processing is complete or when access to a method, property, or so on can be satisfied only by a call to the presentation client waiting for the return.

Use this method to force visual changes to be applied at some logical point before the current processing is complete (for example, a progress bar to update its display at a logical point).

Note This method is ignored if the client node is not a JADE thin client presentation client or if there are no queued commands.

forms

Signature `forms(formNumber: Integer): Form;`

The **forms** method of the **Application** class enables logic to access running forms in the application. This method returns a reference to the form at the number specified in the **formNumber** parameter, or **null** if the form number is invalid. This method returns an object of type **Form**, which enables the properties of the form to be accessed. The default MDI parent frame is not included in the list of returned forms.

To access a specific form control, you must convert the object to a form of the appropriate type, as shown in the following example that examines all active forms for the application and accesses the **text1** control of the menu form.

```
vars
    form : Form;
    indx : Integer;
begin
    foreach indx in 1 to app.formsCount do
        form := app.forms(indx);
        if form.name = "Menu" then
            form.Menu.text1.text := "The text for text box text1";
        endif;
    endforeach;
end;
```

Note You should *not* write logic like that shown in this example to cycle through forms in the range **1** through the value of **app.formsCount** when the logic involves the unloading of forms. (Unloading a form causes the **formsCount** property to change and the table of forms to be compressed.)

formsCount

Signature `formsCount(): Integer;`

The **formsCount** method of the **Application** class returns the current number of active forms for the application. As this count is dynamic, you should not store it for later use.

Note The default MDI parent frame is not included in the list of returned forms.

The following example shows the use of the **formsCount** method to examine all active forms and unload all form instances except for the Menu form.

```
vars
    form : Form;
    indx : Integer;
begin
```

```
    indx := app.formsCount;
    while indx >= 1 do
        form := app.forms(indx);
        if form.name <> "Menu" then
            form.unloadForm;
        endif;
        indx := indx - 1;
    endwhile;
end;
```

Notes If the logic loop results in a form being unloaded, the loop should process the forms in reverse order (as shown in the above example), as the value of **formsCount** is decreased by 1 after the unload and the forms array is contracted by the removal of the unloaded form.

A value of zero (**0**) is returned if this method is invoked from a server method.

generateUuid

Signature `generateUuid(variant: Integer): Binary;`

The **generateUuid** method of the **Application** class generates and returns a binary Universally Unique Identifier (UUID) value, which can be used as an attribute of an object so that it can be exposed to third-parties.

Use one of the following global constants in the **UUIDVariants** category for the **variant** parameter, to specify the layout of the UUID.

Global Constant	Integer Value	Description
VariantDce	2	Distributed Computing Environment, which is the scheme used by Qt C++ application development framework, and which is the recommended variant to pass to the **generateUuid** method
VariantMicrosoft	3	Reserved for Microsoft backward compatibility (GUID)
VariantNcs	1	Reserved for NCS (Network Computing System) backward compatibility

The following code example shows the use of the **generateUuid** method.

```
create() updating;
begin
    self.uuid := app.generateUuid(VariantDce);
end;
```

getApplicationSkin

Signature `getApplicationSkin(): JadeSkinApplication;`

The **getApplicationSkin** method of the **Application** class returns a reference to the **JadeSkinApplication** object that is currently set for the application or it returns **null** if there is no **JadeSkinApplication** object set. (See also the **Application** class **setApplicationSkin** method.)

Note The **getSkin** method of the **Application** class returns a reference to the **JadeSkin** object. Call the **getSkin** method only if the skin was set by using the **setSkin** method of the **Application** class.

getCurrentSession

Signature `getCurrentSession(sessionId: String): WebSession;`

The **getCurrentSession** method of the **Application** class returns a reference to the **WebSession** object of the session identifier specified in the **sessionId** parameter.

If the session is not valid (for example, the session has timed out and been removed) or the method is called when the application is not a Web-enabled application, the **getCurrentSession** method returns a **null** value.

For details about obtaining the session identifier, see the **Application** class **getCurrentSessionId** method.

getCurrentSessionId

Signature `getCurrentSessionId(): String;`

The **getCurrentSessionId** method of the **Application** class returns a string of up to 16 characters that identifies the current **WebSession** object.

You can store the returned value in a property defined in the **Application** class, for example. You should reimplement the **processRequest** method of the **WebSession** class and set the value in that method.

When you require the session object for this identifier, you can then call the **getCurrentSessionId** method on the **Application** class. If there is no current session, the **getCurrentSessionId** method returns null (**""**).

For details about obtaining the current Web session, see the **Application** class **getCurrentSession** method.

getExternalDatabase

Signature `getExternalDatabase(dbName: String): ExternalDatabase;`

The **getExternalDatabase** method of the **Application** class returns a reference to the shared transient instance of the external database specified in the **dbName** parameter or **null** if there is no active external database with the specified name.

getForm

Signature `getForm(formName: String): Form;`

The **getForm** method of the **Application** class enables logic to access the active forms in the application by name and returns a reference to the first active form object with the name specified in the **formName** parameter or **null** if there is no active form with the specified name.

This method returns an object of type **Form**, which is the superclass of all forms, enabling you to access all standard form properties.

To access the properties and methods specific to a particular form subclass, you must convert the returned object to a form object of the correct form type (by using a type guard operation).

The following example shows the use of the **getForm** method to retrieve the current period number from the first **MaintPeriods** form.

```
vars
    form          : Form;
    currentPeriod : Integer;
begin
    form := app.getForm("MaintPeriods");
```

```
    if form <> null then
        currentPeriod := form.MaintPeriods.currentPeriod;
    else
        currentPeriod := 0;
    endif;
    return currentPeriod;
end;
```

getEnhancedLocaleSupport

Signature `getEnhancedLocaleSupport(): Boolean;`

The **getEnhancedLocaleSupport** method of the **Application** class returns the current value (**true** or **false**) of the **EnhancedLocaleSupport** parameter in the [JadeEnvironment] section of the JADE initialization file on the database node.

The state for all nodes is set from the server state when the node initializes.

getIniFileName

Signature `getIniFileName(): String;`

The **getIniFileName** method of the **Application** 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 a filename **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**).

In JADE thin client mode, this method retrieves the initialization file from the presentation client. (Use the **Application** class **getIniFileNameAppServer** method to retrieve the file from the application server or **process.getIniFileName** to retrieve the file associated with the process of the receiver.)

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 the new application enables you to get the name of the initialization file that was used when the node started up.

---

getIniFileNameAppServer

Signature `getIniFileNameAppServer(): String;`

The **getIniFileNameAppServer** method of the **Application** class returns the name of the application server JADE initialization file when the application is running in JADE thin client mode.

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 a filename **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, **s:/jade/system/jade.ini**).

If the application is not running in thin client mode, this method is equivalent to the **Application** class **getIniFileName** method; that is, it returns the full path and file name of the JADE initialization file of the workstation on which the application is running. Use the **getIniFileName** method to return the presentation client initialization file or **node.getIniFileName** to return the initialization file associated with the current node.

getJadeInstallDir

Signature `getJadeInstallDir(): String;`

The **getJadeInstallDir** method of the **Application** class returns the directory in which the JADE binaries are installed; for example:

```
c:\jade\bin
```

In JADE thin client mode, this method retrieves the installation directory on the presentation client.

Use the **Application** class **getJadeInstallDirAppServer** method to obtain the string from the installation directory on the application server.

getJadeInstallDirAppServer

Signature `getJadeInstallDirAppServer(): String;`

The **getJadeInstallDirAppServer** method of the **Application** class returns the installation directory in which the JADE binaries are installed on the application server when the application is running in JADE thin client mode.

If the application is not running in JADE thin client mode, this method functions like the **Application** class **getJadeInstallDir** method; that is, it returns the directory in which the JADE binaries are installed on the workstation on which the application is running. (The **getJadeInstallDir** method returns the presentation client installation directory.)

getJadeTextEditGlobalSettings

Signature `getJadeTextEditGlobalSettings(): String;`

The **getJadeTextEditGlobalSettings** method of the **Application** class returns a string containing the contents of the global settings table compiled into the **JadeTextEdit** class library. See also the **Application** class **getJadeTextEditOneSetting** and **updateJadeTextEditAppSettings** methods and the **JadeTextEdit** class **applySettings** and **updateAppSettings** methods.

getJadeTextEditOneSetting

Signature `getJadeTextEditOneSetting(key: String): String;`

The **getJadeTextEditOneSetting** method of the **Application** class returns a string containing the **JadeTextEdit** class setting specified in the **key** parameter. This method searches the **AppSettings** table before it searches the **GlobalSettings** table. Text substitution (that is, **$(xxx)** replacement) is performed on the value text but it is not performed on the key text.

The following example shows the use of the **getJadeTextEditOneSetting** method to retrieve the C++ keywords.

```
vars
    str : String;
begin
    str := jte.getJadeTextEditOneSetting("keywords.$(file.patterns.cpp)");
    write str;
```

```
// Do some more processing here...
end;
```

See also the **Application** class **getJadeTextEditGlobalSettings** and **updateJadeTextEditAppSettings** methods and the **JadeTextEdit** class **applySettings** and **updateAppSettings** methods.

getMessageText

Signature `getMessageText(msgNumber: Integer): String;`

The **getMessageText** method of the **Application** class returns the error text associated with an exception that has an **errorCode** specified by the value of the **msgNumber** parameter.

Use the **getMessageText** method in contexts where you know the **errorCode** value but do not have the exception instance. The code fragment in the following example shows the use of the **getMessageText** method.

```
write app.getMessageText(1090);
// Outputs "Attempted access via null object reference"
```

getMouseMoveTime

Signature `getMouseMoveTime(): Integer;`

The **getMouseMoveTime** method of the **Application** class returns the current mouse move time that is in use for the current application running on presentation clients.

This style of mouse operation is transparent to most application operations and achieves a significant reduction of events that are sent.

If the application is running in standard fat client mode, this method returns zero (**0**).

By default, in JADE thin client mode, **mouseMove** and **dragOver** events are discarded when moving the mouse within the same window if the time since the execution of the last move event is less than the mouse move time defined for the current application, unless the mouse comes to rest. (The mouse comes to rest if no **mouseMove** events are received for the minimum of the specified mouse move time or the default value of **200** milliseconds.)

If the user moves the mouse slowly enough, the same results are achieved as those when running your application in standard fat client mode.

Note In JADE thin client mode, no **mouseMove** events are sent to the application server if there is no **mouseMove** event defined for that window.

The first **mouseMove** event received after left-clicking a control in thin client mode immediately generates a **mouseMove** event call to the application server (when that control has logic defined for that event). The **mouseMove** time processing then starts with the next **mouseMove** event that is received.

A user can set the **mouseMove** time for all applications run on a presentation client by using the **MouseMoveTime** parameter in the [JadeThinClient] section of the JADE initialization file or you can set it dynamically from your code, by using the **Application** class **setMouseMoveTime** method.

getOdbcSessionObject

Signature `getOdbcSessionObject(): Object;`

The **getOdbcSessionObject** method of the **Application** class returns a reference to an application-maintained session object when called in an ODBC server application.

Obtaining a saved session object reference using the **getOdbcSessionObject** method in an ODBC server application is analogous to dereferencing the **currentSession** environmental variable in a Web application.

getParamListTypeEntry

Signature `getParamListTypeEntry(index: Integer; paramList: ParamListType): Any;`

The **getParamListTypeEntry** method of the **Application** class returns the value of the parameter in the **ParamListType** pseudo type list specified in the **paramList** parameter at the position specified in the **index** parameter.

The first entry in the parameter list is at index **1**. If the value of the **index** parameter is outside the bounds of the parameter list, an exception is raised.

The following example retrieves the third entry from the parameter list then tests and displays its value.

```
doSomething(pl : ParamListType);
vars
    param : Any;
begin
    param := app.getParamListTypeEntry(3, paramList);
    if param.isKindOf(Integer) then
        write "Third parameter is an Integer= " & param.String;
    elseif param.isKindOf(Boolean) then
        write "Third parameter is a Boolean= " & param.String;
    elseif param.isKindOf(String) then
        write "Third parameter is a String= " & param.String;
    elseif param.isKindOf(Object) then
        write "Third parameter is an Object= " & param.String;
    else
        write "Third parameter is something else= " & param.String;
    endif;
end;
```

getParamListTypeLength

Signature `getParamListTypeLength(paramList: ParamListType): Integer;`

The **getParamListTypeLength** method of the **Application** class returns the number of entries in the **ParamListType** pseudo type parameter list specified in the **paramList** parameter. The following example displays the number of entries in the **p1** parameter list.

```
doSomething(pl : ParamListType);
vars
    count : Integer;
begin
    count := app.getParamListTypeLength(pl);
    write "parameter list length=" & count.String;
end;
```

getPersistentApp

Signature `getPersistentApp(): Application;`

The **getPersistentApp** method of the **Application** class returns a reference to the persistent application from which the transient instance of the receiver was created.

When using imported packages, for example, this method enables you to compare the **Process** class **persistentApp** reference of the process of the receiver with the persistent application to determine whether the application of the process is the same as you local package application.

getProcess

Signature `getProcess(): Process;`

The **getProcess** method of the **Application** class returns a reference to the **Process** object associated with the application; for example, when your application has multiple application objects when you are working with imported packages.

getProfileString

Signature
```
getProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 default:  String): String;
```

The **getProfileString** method of the **Application** 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.)

In JADE thin client mode, this method retrieves the initialization file string from the specified initialization file on the presentation client. (Use the **getProfileStringAppServer** method of the **Application** class to retrieve the string from the initialization file on the application server or **process**.**getProfileString** to retrieve the string from the initialization file of the process of the receiver.)

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.
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 or if the specified key is found in the initialization file but has the special value **<default>**.

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 := app.getProfileString(app.getIniFileName, "JadeClient",
                                   "ServerName", null);
    write "server name is " & server;
end;
```

getProfileStringAppServer

Signature

```
getProfileStringAppServer(fileName: String;
                          section:  String;
                          keyName:  String;
                          default:  String): String;
```

The **getProfileStringAppServer** method of the **Application** class returns 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.

If the application is not running in JADE thin client mode, this method functions like the **Application** class **getProfileString** method; that is, it returns the specified profile string from the workstation in which the application is running. (Use the **getProfileString** method to return the specified profile string from the presentation client or **node**.**getProfileString** to return the specified profile string from the current node.)

The **setProfileStringAppServer** method copies the string into the specified section of an initialization file on the presentation client.

The parameters for the **getProfileStringAppServer** 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 or if the specified key is found in the initialization file but has the special value **<default>**.

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*.

The following example shows the use of the **getProfileStringAppServer** method to determine the server mode for the current JADE initialization file.

```
vars
    server : String;
begin
    server := app.getProfileStringAppServer("\jade\system\JADE.ini",
              "InternalAS.JadeClient", "ServerName", null);
    write "server name is " & server;
end;
```

getRootSchemaFormTranslation

Signature `getRootSchemaFormTranslation(formName: String; entityName: String): String;`

The **getRootSchemaFormTranslation** method of the **Application** class returns the text to be displayed as the form, menu, or control caption specified by the **entityName** parameter on the form specified by the **formName** parameter.

You can translate captions for the following system forms and dialogs only (related to printing or the use of the **JadeRichText** control).

- Print Options dialog
- Select Pages To Print dialog
- Find Text dialog
- Print Preview Find message box
- Print dialog
- Find and Replace dialog (Rich text control)
- Paragraph formatting dialog (Rich text control)
- Insert Table dialog (Rich text control)
- Popup menu (Rich text control)

The **getRootSchemaFormTranslation** method dynamically loads the captions of all required entities when a form is created and when logic dynamically changes a caption.

To handle translation to another language, your application must re-implement the **getRootSchemaFormTranslation** method and return appropriate translations for each requested entity.

A copy of the default **getRootSchemaFormTranslation** method is provided in the **RootSchemaFormTranslation** subdirectory of the **examples** directory on the JADE release medium, with the **getRootSchemaFormTranslation.mth** file name. Any subsequent changes to this method will be documented in the JADE release information.

Notes A caption can include an accelerator key (for example; **'&Cancel'**). Translations must return a string with a unique accelerator that is appropriate to the language.

Captions with carriage return and line feed characters have **'%Cr%Lf'** markers included in the default text. You should retain these markers.

Some captions have parameter markers; for example, **'Print Preview page %1 of %2 pages'**. In this example, the **%<n>** value is a number in the range 1 through 9, to indicate where the parameter value is to be inserted. Your translations should retain these parameter markers so that subsequent logic can insert the required text.

getSchema

Signature `getSchema(): Schema;`

The **getSchema** method of the **Application** class returns a reference to the **Schema** object associated with the application; for example, when your application has multiple application objects when you are working with imported packages.

When working with imported packages, the **getSchema** method and the **currentSchema** system variable may not return the same value. In the following example, a schema called **ImportingSchema** imports a package from a schema called **PackageSchema**.

Compare the output from the code fragments from a non-imported method and from an imported package method.

```
// In a non-imported method
write "schema=" & app.getSchema.name;           // outputs "ImportingSchema"
write "schema=" & currentSchema.name;           // outputs "ImportingSchema"

// In an imported method
write "schema=" & app.getSchema.name;           // outputs "ImportingSchema"
write "schema=" & currentSchema.name;           // outputs "PackageSchema"
```

An exception is raised if the receiver of the **getSchema** method is not the application object of the schema context in which the code is executing. The application object changes as the process switches from one schema to another as a result of a call into a package.

The following method attempts to list all applications for a process. An exception is raised when the **getSchema** method is called and the **appObj** receiver references an application object other than that referenced by the **app** system variable.

```
listApplications();
vars
    appArray : ApplicationArray;
    appObj   : Application;
begin
    create appArray transient;
    process.getAllApps(appArray);
    // appArray contains the standard app object
    // and app objects for any imported packages
```

```
    foreach appObj in appArray do
        write "application=" & appObj.name;
        // next instruction raises an exception if appObj is not app
        write "schema=" & appObj.getSchema.name;
        write "schema=" & appObj.class.schema.name;
    endforeach;
epilog
    delete appArray;
end;
```

The exception can be avoided by not calling the **getSchema** method. Replace the call to **appObj.getSchema.name** with **appObj.class.schema.name**.

getSessionTimeout

Signature `getSessionTimeout(): Integer;`

The **getSessionTimeout** method of the **Application** class returns the Web session timeout value specified for the application.

By default, Web sessions do not time out; that is, the default value of zero (**0**) indicates infinity.

The **Web Options** sheet of the Define Application dialog provides the **Session Timeout** text box, which enables you to specify in minutes the period at which the Web session terminates if no requests have been received within that time.

See also the **Application** class **setSessionTimeout** method.

getSkin

Signature `getSkin(): JadeSkin;`

The **getSkin** method of the **Application** class returns a reference to the JADE skin that is currently set for the application. If no JADE skin is currently set for the application, a **null** value is returned.

Call this method only if the skin was set by using the **Application** class **setSkin** method.

Note The **Application**::**getApplicationSkin** method returns a reference to the **JadeSkinApplication** object.

See also the **Application** class **getSkinCollection** and **setSkin** methods and the **JadeSkin** class.

getSkinCollection

Signature `getSkinCollection(): JadeSkinsColl updating;`

The **getSkinCollection** method of the **Application** class returns the global collection of skins. This collection is global to all schemas and is a manual **ObjectArray** subclass that has no inverse references. The collection is automatically created by the first **app**.**getSkinCollection** call.

To implement your own selection facility, display the **name** property for each **JadeSkinApplication** and **JadeSkin** object in the collection.

See also the **Application** class **setSkin** method and the **JadeSkin** class.

Note This method is part of the implementation of skins for JADE release 5, which has been superseded. For more details, see "Using Skins to Enhance JADE Applications", in Chapter 9 of the *JADE Developer's Reference*.

getSystemVersion

Signature `getSystemVersion(): String;`

The **getSystemVersion** method of the **Application** class returns a string containing the JADE system version; for example:

```
vars
    jadver : String;
begin
    jadver := app.getSystemVersion;
    write jadver;                    // Outputs 7.1.03, for example
end;
```

getTempDir

Signature `getTempDir(): String;`

The **getTempDir** method of the **Application** class returns the temporary directory in which the JADE temporary files of the process are located including a trailing backslash character; for example:

```
c:\temp\
```

In JADE thin client mode, this method retrieves the temporary directory on the presentation client. Use the **Application** class **getTempDirAppServer** method to return the string from the installation directory on the application server.

getTempDirAppServer

Signature `getTempDirAppServer(): String;`

The **getTempDirAppServer** method of the **Application** class returns the temporary directory in which the JADE temporary files are located on the application server when the application is running in JADE thin client mode.

If the application is not running in JADE thin client mode, this method functions like the **Application** class **getTempDir** method; that is, it returns the temporary directory on the workstation on which the application is running. (The **getTempDir** method returns the presentation client temporary directory.)

getThinClientEncryptionType

Signature `getThinClientEncryptionType(): Integer;`

The **getThinClientEncryptionType** method of the **Application** class returns the type of encryption being used by the thin client TCP connection for this application. The encryption type can be one of the values listed in the following table.

Application Class Constant	Integer Value
ThinClientEncryption_External	2 (user-supplied encryption library)
ThinClientEncryption_Internal	1 (Windows 40 bit encryption)
ThinClientEncryption_None	0
ThinClientEncryption_SSL	3 (Secure Sockets Layer (SSL) encryption)

If the application is not running in JADE thin client mode, this method returns **ThinClientEncryption_None** (**0**).

getTransientDbPath

Signature `getTransientDbPath(): String;`

The **getTransientDbPath** method of the **Application** class returns a string containing the full path of the transient database file on the current node.

In JADE thin client mode, the **getTransientDbPath** method returns the path of the transient database file on the application server is returned. When the **getTransientDbPath** method is called from a **serverExecution** method, the full path of the transient database file on the database server node is returned.

See also "Transient Database File Analysis", in Chapter 3 of the *JADE Database Administration Guide*.

getUTCTime

Signature `getUTCTime(): TimeStamp;`

The **getUTCTime** method of the **Application** class returns the current UTC time for the machine on which the method executes.

Note Greenwich Mean Time (GMT) has been replaced as the world standard time by Coordinated Universal Time (UTC), which is based on atomic measurements rather than the rotation of the Earth. (GMT remains the standard time zone for the Prime Meridian, or zero longitude.)

For details about getting the current UTC bias (in minutes) of a specified node for local translation, see the **Application** class **currentUTCBias** method and the **TimeStamp** primitive type **localToUTCTime**, **localToUTCTimeUsingBias**, **utcToLocalTime**, and **utcToLocalTimeUsingBias** methods.

getWebMachineName

Signature `getWebMachineName(): String;`

The **getWebMachineName** method of the **Application** class returns the machine name to be used when generating HTML pages for the **JadeHTMLClass** class **buildFormActionOnly** and **buildLink** methods if you do not want to use the value specified in the working directory of the JADE application (set up in the Define Application dialog).

For details about specifying Internet server machine name and virtual directories for all of your Web-enabled applications or for a specific application, see the *JADE Initialization File Reference* for details about the **URLSpecifications** parameter in the [WebOptions] section of the JADE initialization file. For details about programmatically setting the machine name, see the **setWebMachineName** method.

getWebVirtualDirectory

Signature `getWebVirtualDirectory(): String;`

The **getWebVirtualDirectory** method of the **Application** class returns the virtual directory (URL) to be used when generating HTML pages for the **JadeHTMLClass** class **buildFormActionOnly** and **buildLink** methods if you do not want to use the value specified in the working directory of the JADE application (set up in the Define Application dialog).

For details about specifying Internet server machine name and virtual directories for all of your Web-enabled applications or for a specific application, see the *JADE Initialization File Reference* for details about the **URLSpecifications** parameter in the [WebOptions] section of the JADE initialization file. For details about programmatically setting the virtual directory, see the **setWebVirtualDirectory** method.

globalLockException

Signature `globalLockException(le: LockException io): Integer;`

The **globalLockException** method of the **Application** class implements a generic lock exception handler that displays the Lock Error dialog and continues retrying for locks until the lock is obtained (continues) or until the user cancels the retry operation (aborts). This exception handler cannot be armed from a server method.

The code fragment in the following example shows the use of the **globalLockException** method to arm a global lock exception handler.

```
on LockException do app.globalLockException(exception) global;
```

htmlPageNotFoundMessage

Signature `htmlPageNotFoundMessage(): String;`

The **htmlPageNotFoundMessage** method of the **Application** class returns a string containing the error message that is sent to the receiver when the requested page for the Web application is not found.

inactiveTimeout

Signature `inactiveTimeout() updating;`

The **inactiveTimeout** event method of the **Application** class is called automatically by the JADE executable program (**jade.exe**) if there has been no user activity (that is, mouse or key board action) within the number of seconds specified by the **setInactiveTimeoutPeriod** method **period** parameter. When the inactive time out occurs, the inactive timeout period is reset to zero (**0**), which prevents an automatic reoccurrence.

Note The automatic timeout does not occur if the user is in exception state.

The **inactiveTimeout** method defined in the **Application** class does nothing by default. Reimplement this method in your user applications if you want to define the handling of user timeouts in your applications. Logic in your **inactiveTimeout** event method must reset this value by using the **setInactiveTimeoutPeriod** method for another timeout event to occur.

The **inactiveTimeout** event method can then perform the required processing and re-establish the timeout period on completion. For example, the event could modally show the log-on form to force users to identify themselves after the inactive period, as follows.

```
inactiveTimeout();
vars
    form : LogonForm;
begin
    create form;
    form.showModal;                          // Force user to log on again
    self.setInactiveTimeoutPeriod(300);      // Reset timeout to 5 minutes
end;
```

initialize

Signature `initialize() updating;`

The **initialize** event of the **Application** class is the default event that is called by the application before the start-up form of the application is invoked if you do not define your own **initialize** event to perform any function that is common to all users of this application.

Note The **initialize** event is performed once for each user of the application.

If the event creates a form and does not subsequently unload it, the start-up form of the application is not invoked.

For non-GUI applications, this event method is replaced by the user-specified **initialize** method specified when the application is defined. (For details, see "Passing Parameters to Non-GUI Applications using jadclient", in Chapter 1 of the *JADE Runtime Application Guide*.)

initializeOdbcSelect

Signature `initializeOdbcSelect(rv: RelationalView; username: String);`

The **initializeOdbcSelect** method of the **Application** class is called in an ODBC server application before executing an SQL query. The **rv** parameter specifies the relational view currently in use and the **username** parameter specifies the user code of the logged-on user.

You can reimplement this method, if required; for example, to re-establish application state for the user executing the query.

invalidWebSessionMessage

Signature `invalidWebSessionMessage(): String;`

The **invalidWebSessionMessage** method of the **Application** class returns an HTML string to the Web browser when you reimplement this method in the **Application** subclass in a user-defined schema. For example, sending the following HTML string displays a message on the Web browser and provides the user with the option of restarting the application when a session is no longer valid.

```
<html>
<body>
Your session is no longer valid. Click <a
href='http://wilbur1a/jade/jadehttp.dll?NewWeb'>here</a> to restart
</body>
</html>
```

In this example, you would replace the **HREF** parameter value with the address of your Web site.

A null string (**""**) implements the default behavior; that is, the Web browser is refreshed with the application start-up form.

Note If you override the default behavior by implementing this method in the **Application** subclass of your user-defined schema, it is your responsibility to ensure that the returned string is a correctly formatted HTML string.

isActiveXClassIdRegistered

Signature `isActiveXClassIdRegistered(guid: String): Boolean;`

The **isActiveXClassIdRegistered** method of the **Application** class returns **true** if the ActiveX object specified in the **guid** parameter is registered in the operating system registry of the client workstation.

isAppRunning

Signature `isAppRunning(schemaName: String; appName: String): Boolean;`

The **isAppRunning** method of the **Application** class returns **true** if the application specified by the **schemaName** and **appName** parameters is running on the presentation client or standard (fat) client within the same copy of **jade.exe** and returns **false** if the specified application is not running in the same copy of **jade.exe**.

In thin client mode, this is **ApplicationType_GUI** or **ApplicationType_GUI_No_Forms**. On standard clients, the application is **ApplicationType_GUI**, **ApplicationType_GUI_No_Forms**, **ApplicationType_Non_GUI**, **ApplicationType_Non_GUI_Rest**, **ApplicationType_Non_GUI_Web**, **ApplicationType_Rest_Services**, **ApplicationType_AGL_NoState_G**, or **ApplicationType_AGL_NoState_N**.

See also the **Application** class **startApplication** and **startApplicationWithParameter** methods.

isBeingDebugged

Signature `isBeingDebugged(): Boolean;`

The **isBeingDebugged** method of the **Application** class returns **true** if the application is being run through the JADE debugger or **false** if the application is not being debugged. This method enables you to execute additional debugging code at run time when an application is being debugged.

isControlSupported

Signature `isControlSupported(class: GUIClass): Boolean;`

The **isControlSupported** method of the **Application** class returns **true** if the subclass of the **Control** class specified by the **class** parameter is supported by the current presentation client.

If the **class** parameter is not a supported control type or is not a subclass of the **Control** class, the method returns **false**.

The following example shows the use of the **isControlSupported** method.

```
boolean := app.isControlSupported(ActiveXControl);
```

For standard Windows presentation clients, this method returns **true** for all control classes except **JadeXamlControl**. Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, and Windows Server 2008 support **JadeXamlControl** only if .NET 3 is installed. If .NET 3 is not installed, **false** is returned.

isFormOpen

Signature `isFormOpen(cls: Class): Boolean;`

The **isFormOpen** method of the **Application** class returns **true** if a form for the class specified in the **cls** parameter is open (running) for this application.

isMultiUser

Signature `isMultiUser(): Boolean;`

The **isMultiUser** method of the **Application** class returns **true** if the application is running in multiuser mode. This method returns **false** if the application is running in single user or read-only mode.

isUnicode

Signature `isUnicode(): Boolean;`

The **isUnicode** method of the **Application** class returns **true** if the application is running with Unicode characters and strings or it returns **false** if the application is running with ANSI characters and strings.

isValidObject

Signature `isValidObject(obj: Object): Boolean;`

The **isValidObject** method of the **Application** class is used to establish if the object specified in the **obj** parameter exists, by returning **true**. This method returns **false** if the specified object has been deleted.

Tip Use this method in **LockException** handlers rather than the **Global**::**isValidObject** method, to avoid *Object not available* exceptions occurring when **global** is locked.

isWebService

Signature `isWebService(): Boolean;`

The **isWebService** method of the **Application** class returns **true** if the application is running as a Web service or it returns **false** if the application is not running as a Web service.

jadeReportWriterAppName

Signature `jadeReportWriterAppName(): String;`

The **jadeReportWriterAppName** method of the **Application** class is called by the JADE Report Writer Designer application to return the name of the application.

Although this method returns the value of the **name** property of the receiver application by default, you can reimplement this method in the **Application** class of your user schemas if you want to return a specific value; that is, a system identifier (variable) that may depend on the current JADE application or user, as shown in the following example.

```
jadeReportWriterAppName(): String;
vars
begin
    return "My Test Company":
end;
```

jadeReportWriterParamLiteral

Signature `jadeReportWriterParamLiteral(): String;`

The **jadeReportWriterParamLiteral** method of the **Application** class is called automatically from the JADE Report Writer Designer application before a report is run. This method returns the literal to be reported when a JADE Report Writer parameter has had its "ignore in selection option" set (that is, you have checked the **Ignore** check box for that parameter on the **Parameters** sheet of the Report Properties dialog).

The **jadeReportWriterParamLiteral** method returns a null value (**""**), by default. To report an ignore status value, re-implement this method in your user schema to return an appropriate **String** value (for example, **"<All>"**), which will be reported instead of the parameter value when the parameter has the ignore status set. The **String** value returned by this method is used only if it is not null.

jadeReportWriterParameterIsSet

Signature `jadeReportWriterParameterIsSet(reportName: String; parameterName: String; parameterValue: Any);`

The **jadeReportWriterParameterIsSet** method of the **Application** class is called automatically from the JADE Report Writer Designer application before a report is run. This method is called once for each parameter used in the report and passes the current parameter value so that any transient holding this parameter value can be updated.

Reimplement this method in the **Application** class if you run the report programmatically and you want to save a parameter value (for example, a date) for subsequent use in your application, to achieve the same result as running the report from the JADE Report Writer Designer application.

jadeReportWriterParamObjects

Signature `jadeReportWriterParamObjects(className: String; reportName: String; parameterName: String): ObjectArray;`

The **jadeReportWriterParamObjects** method of the **Application** class is called by the JADE Report Writer Designer application to return an array of objects from which a user can select an object as a parameter for a report.

The value of the **parameterName** parameter is the name of the object parameter defined in the report specified by the value of the **reportName** parameter.

The value of the **className** parameter specifies the class of defined for the object parameter.

The following example shows a **jadeReportWriterParamObjects** method for a report that prints or extracts data on the monthly sales of a branch of a company. When the report is run, the user selects a branch from the object array returned by the **jadeReportWriterParamObjects** method.

```
begin
    if className = "Branch"
    and reportName = "Monthly Branch Report"
    and parameterName = "branch" then
        return app.myCompany.allBranchesArray;
    endif;
end;
```

Reimplement this method in the **Application** class of your schemas if your want to reference the name of the application in which the report is running.

jadeReportWriterTimeDetails

Signature `jadeReportWriterTimeDetails(reportName: String; cpuTime: Real; startTime: TimeStamp; endTime: TimeStamp);`

The **jadeReportWriterTimeDetails** method of the **Application** class records timings of each JADE Report Writer report (contained in the **reportName** parameter).

Reimplement this method in the **Application** class of your schemas if you require timings of JADE Report Writer reports (for example, if you want to log timing details for performance or billing purposes).

The **jadeReportWriterTimeDetails** method is called each time a report designed, by using the JADE Report Writer is run, with the exception of previewing the form at design time. This method is also called when a report is run programmatically (by using the **JadeReportWriterReport** class **run** or **runWithStatus** method) and when the JADE Report Writer Designer application terminates when the **reportName** parameter contains ***DESIGN*** (the CPU time does not include running time previewing, printing, or extracting the report) or ***PREVIEW*** (the CPU time is the total amount of preview time).

As the times apply to the node on which the database is located, the CPU time and the start and end times are those of the server node (or application server, if your application is running in JADE thin client mode).

The **cpuTime** parameter value is in seconds and the **startTime** and **endTime** values are timestamps of the workstation locale on which the database is located. For example, if this method is called from a presentation client in New Zealand accessing report data on an application server in the United Kingdom, the **startTime** and **endTime** parameters record Greenwich Mean Time (GMT) timestamp values.

jadeWebServiceInputError

Signature `jadeWebServiceInputError(message: Binary): String;`

The **jadeWebServiceInputError** method of the **Application** class enables you to log a message when the input to a Web service request that is being processed by a JADE Web services provider application is not correctly encoded in the UTF8 format.

A JADE Web service provider application requires all input to be encoded in UTF8. When input is received that cannot be decoded as valid UTF8 format, an exception is raised from the provider application. As part of the exception handling, the **jadeWebServiceInputError** method is called.

The **message** parameter contains the input message as a **Binary** character sequence, and the return value from the method is a string containing the error message detail to be sent back to the external consumer in the SOAP exception text. The default implementation of this method does nothing with the **message** parameter and returns the following message.

```
Input is not encoded as valid UTF-8
```

Reimplement this method in the **Application** subclass in your schema if you want to log a message and return the SOAP exception text.

licencesExceededMessage

Signature `licencesExceededMessage(): String;`

The **licencesExceededMessage** method of the **Application** class reimplements the **RootSchema** method that displays a message on a Web browser when the number of registered licenses for an organization is exceeded (that is, error code 5503 or 5504 is raised). For example, you could reimplement this method to state the following at a site running applications deployed using HTML thin clients over the Internet.

```
Number of Licenses Exceeded.  Please try again later.
```

loadPicture

Signature `loadPicture(fileName: String): Binary updating;`

The **loadPicture** method of the **Application** class loads a picture (icon, bitmap, JPEG, meta, PNG, TIFF, or GIF file) from an external picture file specified in the **fileName** parameter.

This method returns an object of type **Binary**, which is examined to ensure that it is a valid picture file. If the **fileName** parameter is not specified, a **null** image (that is, no picture) is returned.

The following example shows the use of the **loadPicture** method.

```
vars
    graphic : String;
begin
    graphic := "apict.bmp";
    icon := app.loadPicture("c:\my.icon");            // set form icon
    pictureIcon.picture := app.loadPicture(graphic);  // set picture
end;
```

minimumResponseTimeExceededMsg

Signature `minimumResponseTimeExceededMsg(): String;`

The **minimumResponseTimeExceededMsg** method of the **Application** class returns a default HTML string of the receiver to a Web browser user when the maximum wait time contained in the **Application** class **webMinimumResponseTime** property is exceeded.

For example, the default HTML message is sent to the Web browser user if the JADE application hangs and therefore does not return a response to the Web browser within the specified number of seconds.

You can override this method in the **Application** subclass of your user-defined schema. However, if you override this method, it is your responsibility to ensure that the returned string is a correctly formatted HTML string.

msgBox

Signature
```
msgBox(msg:   String;
       title: String;
       flags: Integer): Integer;
```

The **msgBox** method of the **Application** class displays a message in a dialog and waits for the user to click a button.

Note When you are working with multiple monitors on one workstation, by default, Microsoft Windows displays a message box on the monitor where the application last had focus.

The **msgBox** method returns a value indicating the button that the user has clicked. (For details, see "msgBox Method Return Values", later in this section.)

The **msgBox** method parameters are listed in the following table.

Parameter	Description
msg	Specifies the message displayed in the dialog
title	Specifies the dialog title
flags	Specifies the buttons and icons that are to be displayed in the dialog and the default button

The **msgBox** method is always executed on the client node or presentation client workstation, even if it is called from a server method.

While a message box is displayed, notifications and timer events continue to be processed.

An exception is raised (that is, *1291 - GUI request not allowed in this context*) if this method is called in a non-GUI application.

Tip It is not good programming practice to display message boxes while in transaction state (that is, you should abort or commit the transaction *before* calling the **msgBox** method).

In JADE thin client mode, this method executes on the presentation client workstation. The following example shows the use of the **msgBox** method in which the **OK** and **Cancel** buttons are displayed.

```
vars
    number : Integer;
begin
    number := app.msgBox("Delete this object?", "Deletion
                         Confirmation", MsgBox_OK_Cancel);
    if number = MsgBox_Return_Cancel then                // canceled?
        return;
    endif;
    ...
end;
```

Using the flags Parameter

The **flags** parameter of the **msgBox** method is a numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality.

The flags parameter values and the associated global constants provided by the **MessageBox** category are listed in the following table.

Global Constant	Integer Value	Description
MsgBox_OK_Only	0	Display **OK** button only
MsgBox_OK_Cancel	1	Display **OK** and **Cancel** buttons
MsgBox_Abort_Retry_Ignore	2	Display **Abort**, **Retry**, and **Ignore** buttons
MsgBox_Yes_No_Cancel	3	Display **Yes**, **No**, and **Cancel** buttons
MsgBox_Yes_No	4	Display **Yes** and **No** buttons
MsgBox_Retry_Cancel	5	Display **Retry** and **Cancel** buttons
MsgBox_Stop_Icon	16	Display Stop icon
MsgBox_Question_Mark_Icon	32	Display Question Mark icon
MsgBox_Exclamation_Mark_Icon	48	Display Exclamation Mark icon
MsgBox_Information_Icon	64	Display Information icon
MsgBox_Default_First	0	First button is the default
MsgBox_Default_Second	256	Second button is the default
MsgBox_Default_Third	512	Third button is the default
MsgBox_App_Modal	0	Application modal (the user must respond to the message box before continuing work in the current application)
MsgBox_System_Modal	4096	System modal (all applications are suspended until the user responds to the message box)

In this table, the first group of values (**0** through **5**) describes the number and type of buttons displayed in the dialog. The second group of values (**16**, **32**, **48**, and **64**) describes the icon style. The third group of values (**0**, **256**, and **512**) determines the button that is the default. The fourth group of values (**0** and **4096**) determines the modality of the message box. The default message box title is the application name.

Note When adding numbers to create a final value for the **flags** parameter, use only one number from each group.

For application modal message boxes, the **msgBox** method displays a maximum of 1024 characters (longer messages are truncated after 1024 characters). Message strings longer than 255 characters with no intervening spaces are truncated after 255 characters. For system modal message boxes, the number of characters that you can display depends on screen resolution and if the string to be displayed is on one or more lines.

The **msgBox** method breaks lines automatically at the right edge of the dialog. If you want to set line breaks, place a linefeed (ANSI character **10**) or **CrLf** before the first character of the text that is to begin each new line.

msgBox Method Return Values

The value returned by the **msgBox** method indicates which button has been selected, as shown in the following table.

MessageBox Category Global Constant	Integer Value	Selected Button
MsgBox_Return_OK	1	OK
MsgBox_Return_Cancel	2	Cancel
MsgBox_Return_Abort	3	Abort
MsgBox_Return_Retry	4	Retry
MsgBox_Return_Ignore	5	Ignore
MsgBox_Return_Yes	6	Yes
MsgBox_Return_No	7	No

If the message box dialog displays a **Cancel** button, pressing the Esc key has the same effect as clicking the **Cancel** button.

Caution When using this method for a Web page, you should use **app.msgBox** sparingly and with care as there is no modal support with Web applications, and any code following **app.msgBox** in your JADE method continues to be executed.

Handling Translatable Strings on Message Box Button Captions

The **msgBox** method uses translatable strings for the button captions if they are available for non-English translations.

The construction of a message box is as follows.

- If an application form skin is not in use and no message box translated captions are available (or the current locale is English), the standard Windows message box is displayed.

If an application form skin is in use or message box translated captions are available, a custom message box is displayed. If no message box translated captions are available and the locale is not English, the button captions from the resources in the Windows **user32.dll** are used. This means the captions will be displayed using the language that is currently installed on the user's workstation.

If a button translation is available, the text of the translation is used. The custom message box is built so that its size does not exceed the displayable screen area. If the text displayed exceeds the displayable area, a vertical scroll bar is displayed so that the entire message can be accessed.

Message Box Translatable String Handling for Button Captions

When an application is initiated or the presentation client current locale is changed and the locale is not English, the following translatable strings are read, if available, and sent to the presentation client.

- MsgBox_Caption_OK
- MsgBox_Caption_Cancel
- MsgBox_Caption_Abort
- MsgBox_Caption_Retry
- MsgBox_Caption_Ignore
- MsgBox_Caption_Yes
- MsgBox_Caption_No
- MsgBox_Caption_TryAgain
- MsgBox_Caption_Continue

Note It is not necessary to define these translatable strings unless a message box needs to be shown in a locale other than the language installed on the user's workstation.

odbcWorkerFinalize

Signature `odbcWorkerFinalize() updating;`

The **odbcWorkerFinalize** method of the **Application** class must be called by the **finalize** method in an ODBC server application.

The **odbcWorkerFinalize** method ensures a clean shutdown of the ODBC server application.

odbcWorkerInitialize

Signature `odbcWorkerInitialize() updating;`

The **odbcWorkerInitialize** method of the **Application** class must be called by the **initialize** method in an ODBC server application. The method processes the ODBC server configuration parameters, starting additional copies of the application as required.

When ODBC initialization has completed successfully, the service is ready to accept client connections on the configured TCP listen port. Each worker application joins the transport group associated with the ODBC service for a schema and relational view combination.

paintIfRequired

Signature `paintIfRequired() clientExecution;`

The **paintIfRequired** method of the **Application** class causes all forms of the application to be repainted if a repaint is required; for example, while performing a long processing loop, to ensure that the user presentation is updated after the user brings another application to the front and then returns to JADE.

The JADE executable calls the **DisableProcessWindowsGhosting()** Microsoft API on initiation, which disables Windows' ghosting so that a non-responsive form does not show *Not Responding*, nor does it have the ghosting effect applied by Windows. However, the form will still not automatically paint itself when the presentation thread is busy processing JADE logic. Windows automatically redraws that part of the form or forms that need refreshing from a saved copy of the previously painted image or images.

A **refreshNow** event is performed on that part of the form that needed refreshing. If **paint** events are not required, no action is performed.

The **paintIfRequired** method performs any repainting required without having to perform an **app.doWindowEvents** method call, and therefore does not allow the user interface to be active.

Other than any **paint** events, no other events, notifications, or timer events will be processed as a result of this **paintIfRequired** method call.

Note After a repaint, any clicked button that initiated the processing loop will be drawn in the up position, so it will be important that the user is given a visual indication that the processing is still progressing by some other means; for example, by using the **app.mousePointer := 11** (busy) property value.

You will need to add a call to your logic loop that is regularly performed; for example, call it when the **Cancel** button is checked for a **click** event, when a progress bar update ticks over a percentage, or at a specified number of seconds, as shown in the following code fragment.

```
cancelled := false;
    while not cancelled do
        // ... logic
    // the click event sets the cancelled property
    btnCancel.doWindowEvents(0);
    app.paintIfRequired();
endwhile;
```

playSound

Signature `playSound(waveFileName: String);`

The **playSound** method of the **Application** class plays the .**wav** file specified in the **waveFileName** parameter and returns when the complete sound file has been played. (The **app.playSoundAsync** method starts playing the specified .**wav** file and returns immediately.) In JADE thin client mode, this method always executes on the presentation client.

The **waveFileName** parameter must include the full path and file name, as shown in the code fragment in the following example.

```
app.playSound("s:\jademedia\arlo.wav");
```

This method does not raise an exception if the sound cannot be played or if the specified file is invalid, so that code can function silently on a workstation without a sound card.

See also the **Sound** class **play** method, which transfers a sound wave image from the receiver object to memory and then creates a thread to play the sound.

playSoundAsync

Signature `playSoundAsync(waveFileName: String);`

The **playSoundAsync** method of the **Application** class starts the playing of the .**wav** file specified in the **waveFileName** parameter and returns immediately. (The **app.playSound** method waits for the complete sound file to be played before returning.)

In JADE thin client mode, the **playSoundAsync** method always executes on the presentation client.

Calling this method causes any existing sound that is playing to be cancelled. Calling this method with a **null** file name also cancels any sound that is being played.

The **waveFileName** parameter must include the full path and file name, as shown in the code fragment in the following example.

```
app.playSoundAsync("s:\jademedia\arlo.wav");
```

This method does not raise an exception if the sound cannot be played or the specified file is invalid, so that code can function silently on a workstation without a sound card. (See also the **Sound** class **play** method, which transfers a sound wave image from the receiver object to memory and then creates a thread to play the sound.)

productionMode

Signature `productionMode(): Boolean;`

The **productionMode** method of the **Application** class returns **true** if the database from which the application is running has production mode set.

For details, see "Running JADE Production Mode Databases", in Chapter 1 of the *JADE Runtime Application Guide*.

random

Signature `random(range: Integer): Integer;`

The **random** method of the **Application** class returns a random positive number in the range **0** through the value of the **range** parameter, inclusive.

If the supplied range value exceeds 32,767, it is reset internally to **32,767**. If the supplied range value is less than zero (**0**), it is internally reset to zero (**0**), which forces the **random** method to return to zero.

The code fragment in the following example shows the use of the **random** method.

```
rand := app.random(100); // get a random number between 0 and 100, inclusive
```

Note The sequence of generated random numbers is architecture-dependent.

See also the **Application** class **random31** and **seedRandom** methods.

random31

Signature `random31(seed: Integer io;`
`range: Integer): Integer;`

The **random31** method of the **Application** class returns a random positive number in the range **0** through the value of the **range** parameter, inclusive. Use this method to generate random numbers with a larger range of values compared to the **Application** class **random** method. In addition, the **random31** method enables you to generate multiple sets of random numbers within a single application, by maintaining multiple values of the **seed** parameter.

The valid values for the **seed** parameter are in the range **1** through **Max_Integer**, inclusive. The valid values for the **range** parameter are in the range **0** through **Max_Integer - 1**, inclusive.

If the supplied value of the:

- **seed** parameter is not valid, JADE selects a new random seed value
- **range** parameter exceeds **Max_Integer - 1** (that is, 2,147,483,646), it is internally reset to **2,147,483,646**
- **range** parameter is less than zero (**0**), it is internally reset to zero (**0**), which forces the **random31** method to return zero

The method in the following example calls a method that uses the **random31** method to generate a random letter.

```
testRandomLetters();
vars
    i : Integer;
    seed : Integer;
begin
    seed := app.clock;
    foreach i in 1 to 10 do
        getRandomLetter(seed);
    endforeach;
end;
```

Preserve the value of the **seed** parameter between calls to the **random31** method by using a local variable as shown in the following example, or alternatively by making the **seed** parameter an attribute of an object.

```
getRandomLetter(seed: Integer io);
constants
    Letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
begin
    write Letters[app.random31(seed, 25) + 1];
end;
```

See also the **Application** class **random** and **seedRandom** methods.

relativeMachineMicros

Signature `relativeMachineMicros(): Decimal;`

The **relativeMachineMicros** method of the **Application** class returns a high-accuracy machine-relative time in microseconds (that is, millionths of a second). This method is based on a hardware timer. (Internal ticks per second can vary, depending on hardware.)

The time is relative to the machine that is executing the method, which is the application server host when the application is run in JADE thin client mode. This relative time value may wrap around after a period of machine up-time. The period of up-time is hardware-dependent.

As the maximum precision of the timer is 19 digits, you should use a **Decimal[19]** (or higher) precision decimal for storage and computation.

relativeMachineTime

Signature `relativeMachineTime(): Decimal;`

The **relativeMachineTime** method of the **Application** class returns a high-accuracy machine-relative time in milliseconds. This method is based on a hardware timer. (Internal ticks per second can vary, depending on hardware.)

The time is relative to the machine that is executing the method, which is the application server host when the application is run in JADE thin client mode. This relative time value may wrap around after a period of machine up-time. The period of up-time is hardware-dependent.

As the maximum precision of the timer is 19 digits, you should use a **Decimal[19]** (or higher) precision decimal for storage and computation.

removeSessionMessage

Signature `removeSessionMessage(): String;`

The **removeSessionMessage** method of the **Application** class reimplements the **RootSchema** method that displays a message on a Web browser when the Web session running an application deployed in HTML thin client mode ends. For example, you could reimplement this method to state the following at a site running applications deployed using HTML thin clients over the Internet.

```
Your session has now ended. Please sign on again.
```

repairCollection

Signature `repairCollection(coll: Collection input);`

The **repairCollection** method of the **Application** class removes invalid object references and fixes up dictionary keys in the collection specified in the **coll** parameter. The **repairCollection** method enables you to perform an online repair of a collection while it is concurrently updated, as shown in the following example.

```
app.repairCollection(aColl);
```

The **repairCollection** method implemented by the **Application** class creates and executes a background process that invokes a **CollClass** method to perform the repair.

To perform a repair synchronously, you can invoke the **CollClass** method directly, as shown in the following example.

```
Collection.repairCollection(aColl);
```

The **repairCollection** method records progress information and information about entries that have been corrected, in the **jommsg.log** file.

This method iterates the collection specified in the **coll** parameter and performs the following actions.

- Sets, arrays, and external key dictionaries:

  Removes references to non-existent objects or references that are not type-compatible with the membership of the collection.

- Member key dictionaries and dynamic dictionaries (**DynaDictionary** instances).

  Removes references to non-existent objects or references that are not type-compatible with the membership and checks that dictionary keys match the member keys. When they do not, the entry is removed and reinserted with the correct keys.

No action is taken with arrays of primitive types.

The **repairCollection** method differs from the **Collection**::**rebuild** method used by the JADE Logical Certifier utility in the following ways.

- Does not have to be enclosed in a transaction to repair persistent collections but instead it iterates the target collection and performs a database transaction for each entry that requires repair.
- Does not attempt to restore structural integrity of the collection itself as it deals only with entries within the collection.

When there are a relatively small number of entries to be repaired in a large collection, the **Application** class **repairCollection** method is significantly faster than the **Collection**::**rebuild** method used by the JADE Logical Certifier utility and it is designed to be invoked online.

Notes Use the **Application**::**repairCollection** method only when you know that the collection is structurally sound and that only the entries in that collection are invalid. You can obtain an indication of the structural integrity of the collection by iterating the collection and counting entries. If the iteration completes without encountering an exception *and* the number of entries is equal to the value of the **Collection**::**size** property, it is likely the collection is structurally sound. When in doubt about the structural integrity, use the JADE Logical Certifier utility, which uses the **Collection**::**rebuild** method. For details, see Chapter 5, "JADE Logical Certifier Diagnostic Utility", of the *JADE Object Manager Guide*.

Because the repair is done by a background process, you can use the **repairCollection** method only with committed persistent or shared transient collections.

rpsDataPumpFinalize

Signature `rpsDataPumpFinalize() updating;`

The **rpsDataPumpFinalize** method of the **Application** class performs functions required to stop incrementally replicating objects for a user-defined RPS **Datapump** application. This method must be called from the **finalize** method of the user-defined RPS **Datapump** application.

The **rpsDataPumpFinalize** method is valid only on RPS nodes.

rpsDataPumpInitialize

Signature `rpsDataPumpInitialize(userCallbackReceiver: JadeRpsDataPumpIF) updating;`

The **rpsDataPumpInitialize** method of the **Application** class performs functions required to initialize a user-defined RPS **Datapump** application.

The **userCallbackReceiver** parameter, if non-null, represents an instance of a user class that implements the **JadeRpsDataPumpIF** interface. This instance receives callbacks for a row before the row is output to the RDBMS, provided the corresponding table for the row is defined to receive such callbacks in the RPS mapping. For details about selecting classes for **Output Callback**, see "Mapping Classes to Tables", in Chapter 15 of the *JADE Development Environment User's Guide*.

This method must be called from the **initialize** method of the user-defined RPS **Datapump** application.

The **rpsDataPumpInitialize** method is valid only on RPS nodes.

seedRandom

Signature `seedRandom(seed: Integer);`

The **seedRandom** method of the **Application** class initializes the random number generator. To set the random number generator with a new value, call this method, specifying the required value in the **seed** parameter.

Note Each application starts with a random **seed** value.

See also the **Application** class **random** and **random31** methods.

serverName

Signature `serverName(): String;`

The **serverName** method of the **Application** class returns the name of the server that is in use for the JADE application, as specified in the program target (command line) options when JADE was executed.

An exception is raised if this method is invoked from a server method.

setApplicationSkin

Signature `setApplicationSkin(skin: JadeSkinApplication);`

The **setApplicationSkin** method of the **Application** class defines the **JadeSkinApplication** object used for the current application at run time. The skin consists of a collection of form and control skins. (One of the collections can be empty.)

When you call this method to set a skin, any existing skin for the application is replaced. When the **Window** class **ignoreSkin** property is set to **false**, each form that is active or subsequently loaded applies the form skin that matches the value of the **skinCategoryName** for the form. If no matching form skin is found, a skin is not applied to that form.

When the **setApplicationSkin** method is first called, the collected skin data is stored as a blob on the **JadeSkinApplication** instance. Subsequent calls use this stored information and do not need to retrieve the skin information.

In addition, a presentation client caches the skin information. As a result, subsequent calls of the **setApplicationSkin** method only need to request the creation of the skin from the presentation client cache file without having to transmit the skin data.

When you change a skin definition using the **JadeSkinMaintence** form or by loading a form and data definition (**.ddb** or **.ddx**) file, the timestamp of all **JadeSkinApplication** instances is updated, which requires a rebuild of the skin information the first time each skin is set for an application or form. If you change JADE skin information by any other means, you must call the **updateSkinTimeStamp** method on the **JadeSkinApplication** instance, to reset the instance timestamp and cause the skin build data to be rebuilt.

Each control that is active or subsequently loaded applies the corresponding control skin matching the value of the **skinCategoryName** property. If there is no matching control skin or if the value of the **ignoreSkin** property is **true**, the control is not drawn with a skin.

Note Changing a skin object after this method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call the **app.setApplicationSkin** method again.

To clear the application skin (cancel the skin display), call this method again with a null value, as follows.

```
app.setApplicationSkin(null);
```

See also the **Application** class **getApplicationSkin** method, the **Form** class **setApplicationSkin** method, and Chapter 9, "Using Skins to Enhance JADE Applications", of the *JADE Developer's Reference*.

setEndpointForWebService

Signature

```
setEndpointForWebService(className:  String;
                         methodName: String;
                         message:    String): String;
```

Reimplement the **setEndpointForWebService** method of the **Application** class in a user schema to redirect an incoming Web service request to another Web service application. The response is provided by the Web service application receiving the redirected request, which could be running in a separate JADE system.

The Web service application that receives Web service requests from consumers initially redirects them to other Web service applications is acting as a *gateway* Web service.

The parameters for the **setEndpointForWebService** method are listed in the following table.

Parameter	Specifies the …
className	Name of the **JadeWebServiceProvider** class, which is in the schema of the Web service provider application to which the Web service request is redirected
methodName	Name of the Web service method
message	Incoming SOAP message

The following example shows a reimplementation of the **setEndpointForWebService** method in the schema of the gateway Web service.

```
vars
    //  GatewayConsumer is a JadeWebServiceConsumer class in current schema
    ws: GatewayConsumer;
begin
    create ws;
    if className = "CustomerService" then
        // CustomerWebService is a JadeWebServiceProvider class in the
        // schema to which the request will be redirected
        ws.setEndpointURL("http://host1/jade/jadehttp.dll?WebApp1&amp;"
                          & "serviceName=CustomerWebService");
    elseif className = "ProductService" then
        // ProductWebService is a JadeWebServiceProvider class in the
        // schema to which the request will be redirected
        ws.setEndpointURL("http://host2/jade/jadehttp.dll?WebApp2&amp;"
                          & "serviceName=ProductWebService");
    endif;
```

```
    // The JadeWebServiceConsumer::invoke method redirects the request
    return ws.invoke(message);
epilog
    delete ws;
end;
```

For more details, see "Creating a Web Service Gateway", in Chapter 11 in your *JADE Developer's Reference*.

setInactiveTimeoutPeriod

Signature `setInactiveTimeoutPeriod(period: Integer);`

The **setInactiveTimeoutPeriod** method of the **Application** class establishes a one-shot timeout period for user activity in a GUI application by defining the period after which the application times out the user if no activity occurs (either mouse or key board action).

The **period** parameter specifies the number of seconds that users can be inactive before being timed out. The default value of zero (**0**) indicates that there is no timeout.

See also the **Application** class **inactiveTimeout** method.

setJadeLocale

Signature `setJadeLocale(requestedLcid : Integer) updating;`

The **setJadeLocale** method of the **Application** class dynamically changes the effective locale to the Locale Identifier (LCID) specified in the **requestedLcid** parameter for forms and translatable strings. For details about the commonly used locale identifier (LCID) global constants, see the **JadeLocaleIdNumbers** category, in Appendix A of the *JADE Encyclopaedia of Primitive Types*.

For example, if you want to translate to **English (United States)** and retain the **English (United Kingdom)** locale of your operating system, you could use another locale such as Belgian Dutch (the **LCID_Dutch_Belgium** global constant) for the translation.

Note 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.

If the value of the **requestedLcid** parameter is:

- A valid locale identifier, the **Application** class **currentLocale** is changed.

  If enhanced locale support is enabled, the **currentLocaleInfo** properties and the current thread locale are updated. The current thread locale identifier affects date, time, numeric, and currency parsing and formatting, and is returned by the **Schema** class **getCurrentLocaleId** method.

- Invalid or not installed, an exception is raised.

  If enhanced locale support is enabled, validation occurs on the application server and the presentation client.

- Zero (**0**), locale information for the current locale is refreshed.

- Not contained in the schema, the **currentLocale** property is set to the current schema default locale. (The value of the **currentLocale** property will be different from the current thread locale identifier obtained from the **Schema** class **getCurrentLocaleId** method.)

- The current thread locale identifier, regional overrides for the session locale, if currently in use, are not applied.
- The **LCID_SessionWithOverrides** global constant, regional overrides for the session locale, if currently in use, are applied.

setMouseMoveTime

Signature `setMouseMoveTime(time: Integer);`

The **setMouseMoveTime** method of the **Application** class enables you to dynamically set the current mouse move time for presentation clients. This style of mouse operation is transparent to most application operations and achieves a significant reduction of the number of events that are sent.

If your application is not running in JADE thin client mode, this method does nothing.

By default, in JADE thin client mode, **mouseMove** and **dragOver** events are discarded when moving the mouse within the same window if the time since the execution of the last move event is less than the mouse move time defined for the current application, unless the mouse comes to rest. (The mouse comes to rest if no **mouseMove** events are received for the minimum of the specified mouse move time or the default value of **200** milliseconds.)

If the user moves the mouse slowly enough, the same results are achieved as those when running your application in standard fat client mode.

Note In JADE thin client mode, no **mouseMove** events are sent to the application server if there is no **mouseMove** event defined for that window.

The first **mouseMove** event received after left-clicking a control in thin client mode immediately generates a **mouseMove** event call to the application server (when that control has logic defined for that event). The **mouseMove** time processing then starts with the next **mouseMove** event that is received.

A user can set the **mouseMove** time for all applications run on a presentation client by using the **MouseMoveTime** parameter in the [JadeThinClient] section of the JADE initialization file. Use the **Application** class **getMouseMoveTime** method to return the current mouse move time.

setOdbcSessionObject

Signature `setOdbcSessionObject(object: Object);`

The **setOdbcSessionObject** method of the **Application** class sets a reference to an application-maintained session object when called in an ODBC server application.

This method would typically be called in the **Application** class **startOdbcSession** method, to save an application-defined context object for the user. The ODBC server maintains this object on behalf of the client that is currently executing a query.

setProfileString

Signature
```
setProfileString(fileName: String;
                 section:  String;
                 keyName:  String;
                 string:   String): Boolean;
```

The **setProfileString** method of the **Application** class copies a string into the section of an initialization file specified in the **section** parameter.

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	Specifies the …
fileName	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	Initialization file section containing the key (parameter) name.
keyName	Name of the key (parameter) whose associated string is to be stored.
string	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 presentation client.

Use the **Application** class **setProfileStringAppServer** method to set the string in the JADE initialization file on the application server or **process**.**setProfileString** to set the string in the JADE initialization file of the application server process.

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
    app.setProfileString(app.getIniFileName, "mySection",
                         ProfileRemoveSection, "");
    // If the user has moved the window, reset it to the default values
    app.setProfileString(app.getIniFileName, "JadeAppServer", "WindowPos",
                         ProfileRemoveKey);
end;
```

setProfileStringAppServer

Signature

```
setProfileStringAppServer(fileName: String;
                          section:  String;
                          keyName:  String;
                          string:   String): Boolean;
```

The **setProfileStringAppServer** method of the **Application** class copies a string into the section of an initialization file specified in the **section** parameter on an application server workstation when the application is running in JADE thin client mode. This method returns **true** if it succeeds in storing the specified string.

If the application is not running in JADE thin client mode, this method functions like the **Application** class **setProfileString** method or **process**.**setProfileString**; that is, it copies a string into the section of an initialization file specified in the **section** parameter on the workstation on which the application is running. To retrieve a stored string, use the **getProfileStringAppServer** method.

The parameters for the **setProfileStringAppServer** 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 client node is used. If this parameter does not contain a full path to the file, Windows searches for the file in the Windows directory on the client node.
section	Initialization file section containing the key (parameter) name.
keyName	Name of the key (parameter) whose associated string is to be stored.
string	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 same application server 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.

To remove a key parameter (**keyName**) from an initialization file, set the **string** parameter to **ProfileRemoveKey** (a global constant in the **JadeProfileString** category). To remove an entire section in an initialization file, set the **keyName** parameter to **ProfileRemoveSection** (a global constant in the **JadeProfileString** category).

The following example shows the use of the **setProfileStringAppServer** method.

```
begin
    // JADE system adds a value of name= (for example, "InternalAS.")
    app.setProfileStringAppServer(app.getIniFileName, "JadeAppServer",
                                  "PictureCacheFile", ProfileRemoveKey);
    // Initialization file name not IDENTICAL, so we can supply a prefix
    // If the user has moved the window, reset it to the default values
    app.setProfileStringAppServer("\jade\system\JADE.ini",
                                  "Test.JadeAppServer", "WindowPos",
                                  ProfileRemoveKey);
    // JADE uses the single-level (top level) section name
    app.setProfileStringAppServer("c:\jade\system\jade.ini",
                                  ".JadeAppServer", "EnableAppRestrictions",
                                  "true");
end;
```

setSessionTimeout

Signature `setSessionTimeout(timeoutValue: Integer) updating;`

The **setSessionTimeout** method of the **Application** class enables you to dynamically set the timeout period for all Web sessions that are subsequently created.

Use the **timeoutValue** parameter to specify in minutes the period at which the Web session ends if no requests have been received within that time. By default, Web sessions do not time out.

Note The maximum time out value is **1439**, which corresponds to 23 hours and 59 minutes.

See also the **Application** class **getSessionTimeout** method.

setSkin

Signature `setSkin(skin: JadeSkin);`

The **setSkin** method of the **Application** class defines the **JadeSkin** object to be used by the application by setting the skin that applies to the application that is currently running.

To cancel skin usage for the application, pass a **null** value as the skin object; that is, **app.setSkin(null)**. See also the **Application** class **getSkin** and **getSkinCollection** methods and the **JadeSkin** class.

Note This method applies only to JADE release 5.1 and 5.2 applications.

setStatusLineDisplay

Signature `setStatusLineDisplay(str: String) updating;`

The **setStatusLineDisplay** method of the **Application** class enables your logic to dynamically change the scrolling text that is displayed in the Web browser status line to the specified **str** parameter value.

setWebMachineName

Signature `setWebMachineName(machineName: String) updating;`

The **setWebMachineName** method of the **Application** class programmatically sets the machine name to be used when generating HTML pages for the **JadeHTMLClass** class **buildFormActionOnly** and **buildLink** methods if you do not want to use the value specified in the working directory of the JADE application (set up in the Define Application dialog). This method applies at run time only.

For details about specifying Internet server machine name and virtual directories for all of your Web-enabled applications or for a specific application, see the *JADE Initialization File Reference* for details about the **URLSpecifications** parameter in the [WebOptions] section of the JADE initialization file.

For details about getting the machine name, see the **Application** class **getWebMachineName** method.

setWebVirtualDirectory

Signature `setWebVirtualDirectory(vd: String) updating;`

The **setWebVirtualDirectory** method of the **Application** class programmatically sets the virtual directory (URL) to be used when generating HTML pages for the **JadeHTMLClass** class **buildFormActionOnly** and **buildLink** methods if you do not want to use the value specified in the working directory of the JADE application (set up in the Define Application dialog).

For details about specifying Internet server machine name and virtual directories for all of your Web-enabled applications or for a specific application, see the JADE *Initialization File Reference* for details about the **URLSpecifications** parameter in the [WebOptions] section of the JADE initialization file. This method applies at run time only.

For details about getting the virtual directory, see the **Application** class **getWebVirtualDirectory** method.

skinDelete

Signature `skinDelete(skinName: String);`

The **skinDelete** method of the **Application** class deletes all skin entity images that were loaded as part of the skin with the name specified in the **skinName** parameter. These images are deleted from JADE and from the directories in the skin directory structure that was created by calling the **skinMakeDirectory** method. (See also the **Application** class **skinLoad** method.)

skinExtract

Signature `skinExtract(skinName: String);`

The **skinExtract** method of the **Application** class displays the common Browse for Folder dialog, which prompts you for a directory in which a directory structure will be created for the skin images for the skin specified by the **skinName** parameter. The value of the **skinName** parameter name of is the application name for the skin. (A list of the application skin names is displayed on the Jade Skin Maintenance dialog.)

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

For details about the structure of the skin directories, see "Directory Structure Example for the Button Control", "Directory Structure Example for the Combo Box Control", or "Naming Convention when Loading JADE Skins", in Chapter 9 of the *JADE Developer's Reference*.

The code shown in the following fragment invokes the **skinExtract** method from a Workspace.

```
app.skinExtract("DemoSkin");
```

skinLoad

Signature `skinLoad();`

The **skinLoad** method of the **Application** class displays the common Browse for Folder dialog, which prompts you for the root directory in which the skin images are located. The name of the root directory is used as the application name for the skin.

Although you can load a partial set of skins, you must load the complete set of skins for a specific control, menu item, or form because any existing skins with the same name are deleted prior to loading in the new skins. For more details, see "Using the JADE Skin Loader", in Chapter 9 of the *JADE Developer's Reference*.

If you want to load skins individually, use the Jade Skin Maintenance dialog. (For details, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.)

Notes The skins are not loaded unless the skins directory has the specified structure. For details and examples, see "Directory Structure Example for the Button Control", "Directory Structure Example for the Combo Box Control", or "Naming Convention when Loading JADE Skins", in Chapter 9 of the *JADE Developer's Reference*. For details about creating an empty skin directory structure into which you can load your skin image files of a specified name in a selected root directory, see the **Application** class **skinMakeDirectory** method.

The **skinLoad** method loads images only. You must change other settings by using the Jade Skin Maintenance dialog after the load is complete.

At the end of the load process, a validation phase logs details about the load. The file name is the skin name with a .**log** file suffix (for example, **DemoSkin.log**), and it is located in the root directory in which the skin images are located (for example, the **DemoSkin** directory). You should review this file, because it lists skins that have not been loaded into the system.

The code shown in the following fragment invokes the **skinLoad** method from a Workspace.

```
app.skinLoad;
```

Tip The **FormAdminMdi** class **zSetupSkinSelectMenu** and **mnuSkin_click** methods in the **ErewhonInvestmentsViewSchema** schema, included in the **erewhon** subdirectory of the **examples** directory on the JADE release medium, provides an example of adding a skin for selection by users in runtime applications.

skinMakeDirectory

Signature `skinMakeDirectory(skinName: String);`

The **skinMakeDirectory** method of the **Application** class creates an empty directory structure into which the images are loaded (by calling the **Application** class **skinLoad** method).

The value specified in the **skinName** parameter is used as the root directory for this skin, as shown in the following code fragment.

```
app.skinMakeDirectory("DocSkins");
```

The **skinMakeDirectory** method invokes the common Browse for Folder dialog, which prompts you to select the root directory in which the specified skin directory structure is created as an immediate descendant of the directory selected in the Browse for Folder dialog.

Tip You can embed category names within the directory name by appending the category name to the control name; for example, by calling **app.skinMakeDirectory("DemoSkin_Flowery");** from a Workspace. The total length of the skin name and category name cannot exceed 20 characters.

The skin name must be unique across the JADE system so that it is the application skin name, optionally appended by the category name, and followed by sufficient acronyms to make it unique.

startApplication

Signature
```
startApplication(schemaName: String;
                 appName:    String): Process;
```

The **startApplication** method of the **Application** class enables your logic to initiate an application in the same JADE database as the initiating application. (Use the **terminate** instruction to terminate the current application.)

The code fragment in the following example shows the use of the **startApplication** method.

```
if eventType = Start_Server_App then
    app.startApplication('ServerApps', 'SApp03');
endif;
```

The parameters for the **startApplication** method are listed in the following table.

Parameter	Description
schemaName	Specifies the name of the schema in which the application is located
appName	Specifies the name of the application to start

This method starts only applications of type **ApplicationType_Non_GUI_Web**, **ApplicationType_Non_GUI_Rest**, or **ApplicationType_Non_GUI** if this method is invoked from a server method or server application. (An exception is raised if this method is invoked from a server method or a server application to start an application of a type other than a non-GUI application.) On a client node, this method starts all types of application. For details about running non-GUI applications in standard (or fat client) mode, see the **Application** class **applicationType** property.

You can use the **MaxWaitAppStart** parameter in the [JadeClient] or [JadeServer] section of the JADE initialization file to increase the time that JADE waits for a **GUI** or **GUI, No Forms** application to initiate on another thread before raising an exception, when your system has a large number of applications to start and the default value of 45 seconds may not be sufficient for the loading on the machine during startup. For details, see your *JADE Initialization File Reference*.

If one application is terminated (when all forms of that application are closed), the database remains open, as each application uses the same open instance of that database.

This method returns the process of the application that was started. The application that calls this method continues executing after JADE has successfully created a new process. If the application is not initiated (for example, because of **EnableAppRestrictions** security restrictions) an exception is raised in the application requesting the initiation.

The types of application that you can start by using the **startApplication** method on client nodes or server nodes are listed in the following table, which also lists their termination behavior.

Start-up Location	GUI	Non_GUI and Non_GUI_Web	GUI_No_Forms
Client nodes	Stays while application has forms	Always stays running	Always stays running
	Executes **finalize** method	Executes **finalize** method	Executes **finalize** method
Server nodes	Not available	Always stays running	Not available
		Executes **finalize** method	

In this table:

- "Stays while application has forms" indicates that the application is terminated automatically when it has no remaining forms or there is explicit programmatic termination
- "Always stays running" indicates that the application requires explicit termination (by using the **terminate** instruction)

startApplicationWithParameter

Signature

```
startApplicationWithParameter(schemaName:          String;
                              appName:             String;
                              initializeParameter: Object input): Process;
```

The **startApplicationWithParameter** method of the **Application** class enables your logic to initiate another application on the same node as the initiating application. This method enables you to share transient objects between JADE applications.

Notes If you use this method to share transient objects between applications (that is, by passing transients to the **initialize** method), ensure that the transients are created as shared transient objects. (For details, see "*create* Instruction", in Chapter 1 of the *JADE Developer's Reference*.)

If you use this method from within a **serverExecution** method, the **initializeParameter** object must be persistent.

The parameters for the **startApplicationWithParameter** method are listed in the following table.

Parameter	Description
schemaName	Specifies the name of the schema in which the application is located
appName	Specifies the name of the application to start
initializeParameter	Passed to the **initialize** method of the application

The object specified in the **initializeParameter** parameter is passed to the **initialize** method of the application, specified by using the JADE development environment Define Application dialog **Initialize Method** combo box. The parameter must be a persistent object or a shared transient object (except in **serverExecution** methods). If the parameter is a non-shared transient object or a shared transient object and the **startApplicationWithParameter** method is used in a server execution method to start a server application, an exception (1000 - Invalid parameter type) exception is raised.

The **initialize** method must have a signature that contains only the **initializeParameter** parameter; that is:

```
method-name(initializeParameter: Object);
```

The following example shows the use of the **startApplicationWithParameter** method.

```
addWorker() updating;
vars
    worker : Worker;
begin
    // create a WORKER shared transient object and
    // start it up as a separate application
    beginTransientTransaction;
        create worker sharedTransient;
        worker.central:= self;
        worker.name:= "Worker " & workers.size.String;
    commitTransientTransaction;
    app.startApplicationWithParameter("Threads", "Threads", worker);
end;
```

This method starts only applications of type **ApplicationType_Non_GUI_Web**, **ApplicationType_Non_GUI_Rest**, or **ApplicationType_Non_GUI** if this method is invoked from a server method or server application. (An exception is raised if this method is invoked from a server method or a server application to start an application of a type other than a non-GUI application.)

On a client node, this method starts all types of application. For details about running non-GUI applications in standard (or fat client) mode, see the **Application** class **applicationType** property.

When multiple applications are initiated, they run independently of each other. The JADE application program switches the context between applications, based on the form or control that is causing an event. If one application is terminated (when all forms of that application are closed), the database remains open, as each application uses the same open instance of that database.

You can use the **MaxWaitAppStart** parameter in the [JadeClient] or [JadeServer] section of the JADE initialization file to increase the time that JADE waits for a **GUI** or **GUI, No Forms** application to initiate on another thread before raising an exception, when your system has a large number of applications to start and the default value of 45 seconds may not be sufficient for the loading on the machine during startup. For details, see the *JADE Initialization File Reference*.

This method returns the process of the application that was started. The application that calls the method continues executing after JADE has successfully created a new process. If the application is not initiated (for example, because of **EnableAppRestrictions** security restrictions) an exception is raised in the application requesting the initiation.

The types of application that you can start by using the **startApplicationWithParameter** method on client nodes or server nodes are listed in the following table, which also lists their termination behavior.

Start-up Location	GUI	Non_GUI and Non_GUI_Web	GUI_No_Forms
Client nodes	Stays while application has forms	Always stays running	Always stays running
	Executes **finalize** method	Executes **finalize** method	Executes **finalize** method
Server nodes	Not available	Always stays running	Not available
		Executes **finalize** method	

In this table:

- "Stays while application has forms" indicates that the application is terminated automatically when it has no remaining forms or there is explicit programmatic termination
- "Always stays running" indicates that the application requires explicit termination (by using the **terminate** instruction)

startApplicationWithString

Signature

```
startApplicationWithString(schemaName:          String;
                           appName:             String;
                           initializeParameter: String): Process;
```

The **startApplicationWithString** method of the **Application** class enables your logic to initiate another JADE application on the same node as the initiating application and to pass a single string to the new application.

The parameters for the **startApplicationWithString** method are listed in the following table.

Parameter	Description
schemaName	Specifies the name of the schema in which the application is located
appName	Specifies the name of the application to start
initializeParameter	Passed to the **initialize** method of the application

The string specified in the **initializeParameter** parameter is passed to the **initialize** method of the application, specified by using the JADE development environment Define Application dialog **Initialize Method** combo box.

The **initialize** method must have a signature that contains only the **initializeParameter** parameter, as shown in the following example.

```
init(custName: String);
vars
    cust: Customer;
begin
    app.root := Root.firstInstance();
    cust := root.customers.getAtKey(custName);
```

```
        if not cust = null then
            cust.sendLetter();
        endif;
        terminate;
    end;
```

The following code fragment shows the use of the **startApplicationWithString** method.

```
app.startApplicationWithString("BankSystem", "UpdateCustomer", "ZQ32112");
```

This method starts only applications of type **ApplicationType_Non_GUI_Web**, **ApplicationType_Non_GUI_Rest**, or **ApplicationType_Non_GUI** if this method is invoked from a server method or server application. (An exception is raised if this method is invoked from a server method or a server application to start an application of a type other than a non-GUI application.)

On a client node, this method starts all types of application. For details about running non-GUI applications in standard (or fat client) mode, see the **Application** class **applicationType** property.

When multiple applications are initiated, they run independently of each other. The JADE application program switches the context between applications, based on the form or control that is causing an event. If one application is terminated (when all forms of that application are closed), the database remains open, as each application uses the same open instance of that database.

You can use the **MaxWaitAppStart** parameter in the [JadeClient] or [JadeServer] section of the JADE initialization file to increase the time that JADE waits for a **GUI** or **GUI, No Forms** application to initiate on another thread before raising an exception, when your system has a large number of applications to start and the default value of 45 seconds may not be sufficient for the loading on the machine during startup. For details, see the *JADE Initialization File Reference*.

This method returns the process of the application that was started. The application that calls the method continues executing after JADE has successfully created a new process. If the application is not initiated (for example, because of **EnableAppRestrictions** security restrictions) an exception is raised in the application requesting the initiation.

The types of application that you can start by using the **startApplicationWithString** method on client nodes or server nodes are listed in the following table, which also lists their termination behavior.

Start-up Location	GUI	Non_GUI and Non_GUI_Web	GUI_No_Forms
Client nodes	Stays while application has forms	Always stays running	Always stays running
	Executes **finalize** method	Executes **finalize** method	Executes **finalize** method
Server nodes	Not available	Always stays running	Not available
		Executes **finalize** method	

In this table:

- "Stays while application has forms" indicates that the application is terminated automatically when it has no remaining forms or there is explicit programmatic termination
- "Always stays running" indicates that the application requires explicit termination (by using the **terminate** instruction)

startAppMethod

Signature
```
startAppMethod(schemaName:    String;
               appName:       String;
               methodName:    String;
               methodParam:   Object input;
               checkSecurity: Boolean): Process;
```

The **startAppMethod** method of the **Application** class enables your logic to initiate another application on the same node as the initiating application. This method enables you to share transient objects between JADE applications and to specify a method to be invoked on the new application.

The parameters for the **startAppMethod** method are listed in the following table.

Parameter	Description
schemaName	Specifies the name of the schema in which the application is located.
appName	Specifies the name of the application to start.
methodName	Specifies the method that is to be invoked on the application.
methodParam	Passed to the method that is invoked.
checkSecurity	If set to **true**, invokes the **getAndValidateUser** method to validate user codes and passwords. If set to **false**, inherits the security profile from the invoking application. (If set to **false** and the invoking application terminates before the start up of the called application has completed, a JADE exception is raised.)

If you use this method to share transient objects between applications (that is, by passing transients in the method specified in the **methodName** parameter), ensure that the transients are created as shared transient objects. (For details, see "*create* Instruction", in Chapter 1 of the *JADE Developer's Reference*.) The method specified in the **methodName** parameter must have a signature that contains only the **methodParam** parameter; that is:

```
method-name(methodParam: Object);
```

This method starts only applications of type **ApplicationType_Non_GUI_Web**, **ApplicationType_Non_GUI_Rest**, or **ApplicationType_Non_GUI** if this method is invoked from a server method or server application. (An exception is raised if this method is invoked from a server method or a server application to start an application of a type other than a non-GUI application.)

On a client node, this method starts all types of application. For details about running non-GUI applications in standard (or fat client) mode, see the **Application** class **applicationType** property.

You can use the **MaxWaitAppStart** parameter in the [JadeClient] or [JadeServer] section of the JADE initialization file to increase the time that JADE waits for a **GUI** or **GUI, No Forms** application to initiate on another thread before raising an exception, when your system has a large number of applications to start and the default value of **45** seconds may not be sufficient for the loading on the machine during startup. For details, see the *JADE Initialization File Reference*.

The object that is specified in the **methodParam** parameter of the **startAppMethod** method is passed to the method specified in the **methodName** parameter, as shown in the following example.

```
okButton_click(btn: Button input) updating;
vars
    count : Integer;
begin
    app.mousePointer := self.MousePointer_Hourglass;
    while count.bump <= number.text.Integer do
```

```
            if lockgen.value then
                app.startApplication('LockTest', 'LockGen');
            endif;
            if lockTestOthers.value then
                app.startAppMethod('LockTest', 'LockTestOthers', "otherMethod",
                                   Records.firstInstance, false);
            endif;
            if locktest.value then
                app.startApplication('LockTest', 'LockTest');
            endif;
        endwhile;
        app.mousePointer := self.MousePointer_Default;
    end;
```

When multiple applications are initiated, they run independently of each other. The JADE application program switches the context between applications, based on the form or control that is causing an event. If one application is terminated (when all forms of that application are closed), the database remains open, as each application uses the same open instance of that database.

Note If a client node application creates a shared transient instance, it *cannot* pass it as a parameter of the **startAppMethod** method if this method is to be initiated on the server node, as the application will be initiated on the server node. (Shared transient instances belong to a node.)

This method returns the process of the application that was started. The application that calls the method continues executing after JADE has successfully created a new process. If the application is not initiated (for example, because of **EnableAppRestrictions** security restrictions) an exception is raised in the application requesting the initiation.

The types of application that you can start by using the **startAppMethod** method on client nodes or server nodes are listed in the following table, which also lists their termination behavior.

Start-up Location	GUI	Non_GUI and Non_GUI_Web	GUI_No_Forms
Client nodes	Stays while application has forms	Does not stay running	Stays while application has forms
	No execution of **finalize** method	No execution of **finalize** method	No execution of **finalize** method
Server nodes	Not available	Does not stay running	Not available
		No execution of **finalize** method	

In this table, "Stays while application has forms" indicates that the application is terminated automatically when it has no remaining forms or there is explicit programmatic termination.

startOdbcSession

Signature `startOdbcSession(rv: RelationalView; username: String);`

The **startOdbcSession** method of the **Application** class is called in an ODBC server application after the **isUserValid** method of the **Global** class, to indicate a new session has been established.

You can reimplement this method, if required.

timedOutSessionMessage

Signature `timedOutSessionMessage(): String;`

The **timedOutSessionMessage** method of the **Application** class reimplements the **RootSchema** method that displays a message on a Web browser when the Web session running an application deployed in HTML thin client mode times out. For example, reimplement this method to state the following at a site running applications deployed using HTML thin clients over the Internet.

```
Your session has timed out. Please sign on again.
```

updateJadeTextEditAppSettings

Signature `updateJadeTextEditAppSettings(text: String): Integer;`

The **updateJadeTextEditAppSettings** method of the **Application** class adds or modifies one or more **JadeTextEdit** class settings specified in the **text** parameter in the current application settings table.

Separate settings in the **text** parameter with the **CrLf** end-of-line sequence. Each setting has the format *key=value*, as shown in the example in the following code fragment.

```
int := app.updateJadeTextEditAppSettings("lead=tin" & CrLf & "tin=gold");
```

This method returns zero (**0**) if the application settings were successfully updated or it returns a JADE error code if the action was unsuccessful. (For details about the causes and actions of JADE error codes, see the appropriate error code in the **JADEMsgs.pdf** file.)

See also the **Application** class **getJadeTextEditGlobalSettings** and **getJadeTextEditOneSetting** methods and the **JadeTextEdit** class **updateAppSettings** method.

userName

Signature `userName(): String;`

The **userName** method of the **Application** class returns the name of the current user as a string.

In JADE thin client mode, this method returns a reference to the user name of the user on the presentation client.

webApplicationDirectory

Signature `webApplicationDirectory(): String;`

The **webApplicationDirectory** method of the **Application** class returns the name of the Web application directory that contains transferred files over a TCP/IP connection when your JADE environment is behind a firewall. To configure a firewall, the **Firewall** parameter in the [WebOptions] section of the JADE initialization file is set to **true**.

To configure the Web server end of the connection for a firewall:

- For Microsoft Internet Information Server (IIS), the **Firewall** parameter in the [**Jadehttp Files**] section of the JadeHttp initialization file must be set to **true**.
- For Apache HTTP Server, the **Firewall** directive in the **Apache Configuration Directives File** must be set to **True**.

When a file is transferred to JADE from a Web browser using the default connection mechanism, the text for the text box that generated the transfer is changed by the **JadeHttp** library to:

```
<original-file-name>;<temporary-file-name-and-path>
```

If you have set the **Firewall** parameter in the [Jadehttp Files] section in the **jadehttp.ini** file to **true**, the text is set to:

```
<original-file-name>;<temporary-file-name>
```

Your application logic that accesses the file must append the Web application directory to the temporary file name to form the actual path. Use the **webApplicationDirectory** method to get the name of the Web application directory.

For more details, see "Firewall for the JADE Internet Environment", in Chapter 3 of the *JADE Initialization and Configuration Guide*. See also the **WebSession** class **createVirtualDirectoryFile** method and the **JadeWebServiceProvider** class **createVirtualDirectoryFile** method.

ApplicationContext Class

The **ApplicationContext** class provides the initial context of packages when a process begins, by creating transient instances of this class (along with other environmental objects such as **app**, **appContext**, and **global**) for the main application in which the package is imported and for each package application. (See also the **Object** class **invokeMethod** method.)

A transient instance of this class is automatically made available to the runtime copy of the application. To access this transient instance, use the **appContext** system variable in your method logic; for example, use **appContext.initialSchema** to access the schema in which the package is defined. This transient instance is unique to a specific copy of the application.

Changes made to the properties are retained until the application copy is terminated. (This data is therefore not available to other copies of the application.)

For details about switching between application contexts, see "Switching Application Contexts When Invoking a Method", in Chapter 8 of the *JADE Developer's Reference*.

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

Inherits From: Object

Inherited By: (None)

ApplicationContext Properties

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

Property	Contains a reference to the …
initialApp	**Application** instance of the context
initialPackage	**JadePackage** instance of the context
initialProcess	**Process** instance of the context; that is, the current process
initialSchema	**Schema** instance of the context

initialApp

Type: Integer

The **initialApp** property of the **ApplicationContext** class contains a reference to the **Application** instance of the context.

initialPackage

Type: Integer

The **initialPackage** property of the **ApplicationContext** class contains a reference to the **JadePackage** instance of the context.

initialProcess

Type: Integer

The **initialProcess** property of the **ApplicationContext** class contains a reference to the **Process** instance of the context; that is, the current process.

initialSchema

Type: Integer

The **initialSchema** property of the **ApplicationContext** class contains a reference to the **Schema** instance of the context.

Array Class

An array is an ordered collection of objects in which the member objects are referenced by their position in the collection.

Notes Unlike other primitive types, a corresponding subclass of **Array** for **MemoryAddress** values does not exist in the **RootSchema**. If you require such an array, subclass the **Array** class in your user schema, selecting **MemoryAddress** as the membership.

For **Array** classes that are internal pseudo arrays (that is, arrays of GUI-related information only in the JADE run time module), the only **Array** class methods that are implemented are **at**, **atPut**, and **createIterator**. The **atPut** method is implemented only for primitive type elements on a **ListBox**, **ComboBox**, or **Table** control; that is, it is not implemented for a collection of controls and menu items.

For details about array subscripts and the **Array** methods, see "Using Subscripts in Arrays" and "Array Methods", in the following subsections.

Inherits From: List

Inherited By: BinaryArray, BooleanArray, ByteArray, CharacterArray, DateArray, DecimalArray, HugeStringArray, Integer64Array, IntegerArray, ObjectArray, PointArray, RealArray, StringArray, StringUtf8Array, TimeArray, TimeStampArray, TimeStampIntervalArray, user-defined **Array** classes

Using Subscripts in Arrays

The bracket (**[]**) subscript operators enable you to assign values to and read values from an array. The code fragments in the following examples show the syntax of bracket subscript operators in **Array** methods.

```
stringArray[5] := "Five";

str := stringArray[5];
```

Note The index value for an array must be of type **Integer**. The other numeric types are not allowed in this context.

However, for large arrays it is more efficient to use an iterator to traverse an array than it is to index your way through. For example, the code shown in the first of the following code fragments is preferable to that shown in the second code fragment.

```
foreach str in array do
    string := str;
endforeach;

while number <= array.size do
    string := array[number];
    number := number + 1;
endwhile;
```

Array Methods

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

Method	Description
add	Adds an entry to the end of the array
at	Returns the entry at a specified index in the array
atPut	Places an entry at a specified index in the array
countOf	Returns the number of times the specified entry occurs in the array
countOf64	Returns the number of times the specified entry occurs in the array as an **Integer64** value
createIterator	Creates an iterator for the array
first	Returns the first entry in the array
getStatistics	Analyzes the array and returns structural statistics
includes	Returns **true** if the array contains a specified object
indexOf	Returns the index of a specified entry if it exists in the array
indexOf64	Returns the index of a specified entry if it exists in the array as an **Integer64** value
initialise	Initializes an array with null entries up to a size specified by the value of the **count** parameter
insert	Inserts an entry at a specified index in the array and moves up all higher entries
last	Returns the last entry in the array
remove	Removes a specified entry from an array, and moves all higher entries down
removeAt	Removes an entry at a specified index from an array, and moves all higher entries down
replace	Replaces an existing entry in an array with another entry

add

Signature `add(value: MemberType) updating;`

The **add** method of the **Array** class adds an entry to the end of an array, thus increasing the size of the array; for example:

```
setAllEntries(labels:  StringArray;
              numbers: IntegerArray;
              colors:  IntegerArray) updating;
// This method passes all data required by an XYGraph object and labels is
// an array containing a label for each entry on the graph, numbers contains
// the numbers to be graphed, and colors contains the colors for each entry.
vars
    entry : Integer;
begin
    entry := 1;
    while entry <= labels.size do
        labelsArray.add(labels.at(entry));
        entry := entry + 1;
    endwhile;
    entry := 1;
```

```
    while entry <= numbers.size do
        dataArray.add(numbers.at(entry));
        entry := entry + 1;
    endwhile;
    entry := 1;
    while entry <= colors.size do
        colorArray.add(colors.at(entry));
        entry := entry + 1;
    endwhile;
    currentList.add(cust);
end;
```

at

Signature `at(index: Integer64): MemberType;`

The **at** method of the **Array** class returns a reference to the entry in the array at the position specified by the **index** parameter; for example:

```
firstCustomer := currentList.at(1);
```

The following examples show the use of the bracket (**[]**) operators to return an entry from an array.

```
firstCustomer := currentList[1];
display(): String;
vars
    temp, alphaNumber : String;
    entry : Integer;
begin
    temp := self.faultNumber.String;
    if temp.length > 4 then
        alphaNumber := temp;
    else
        entry := 0;
        while entry < 4 - temp.length do
            alphaNumber := alphaNumber & (0).String;
            entry := entry + 1;
        endwhile;
        alphaNumber := alphaNumber & temp;
    endif;
    return alphaNumber & " " & self.history.at(1)[1:100];
end;
```

If there is no entry at the specified index, an exception is raised.

atPut

Signature
```
atPut(index: Integer64;
      value: MemberType) updating;
```

The **atPut** method of the **Array** class places an entry in the array of the object specified in the **value** parameter at the position specified by the **index** parameter; for example:

```
foreach fault in app.myCompany.allFaults do
    if fault.isOpen then
        days := fault.getDaysOpen;
```

```
            if days < 8 then
                ia.atPut(1, ia.at(1) + 1);
            elseif days < 31 then
                ia.atPut(2, ia.at(2) + 1);
            elseif days < 61 then
                ia.atPut(3, ia.at(3) + 1);
            else
                ia.atPut(4, ia.at(4) + 1);
            endif;
        endif;
    endforeach;

    currentList.atPut(100, cust);
```

Note You must specify a positive value in the **index** parameter.

The following example shows the use of the bracket (**[]**) operators to assign values to an array.

```
currentList[100] := cust;
```

If the specified index is greater than the size of the array, the array is expanded.

countOf

Signature `countOf(value: MemberType): Integer;`

The **countOf** method of the **Array** class returns the number of times the entry specified in the **value** parameter occurs in the array.

Note Use the **countOf64** method instead of the **countOf** method, if the number of entries could exceed the maximum integer value of 2,147,483,647.

countOf64

Signature `countOf64(value: MemberType): Integer64;`

The **countOf64** method of the **Array** class returns the number of times the entry specified in the **value** parameter occurs in the array as an **Integer64** value.

createIterator

Signature `createIterator(): Iterator;`

The **createIterator** method of the **Array** class creates iterators for the **Array** class and its subclasses.

For details about iterators, see the **Iterator** class.

first

Signature `first(): MemberType;`

The **first** method of the **Array** class returns a reference to the first entry in the array.

This method is equivalent to using the bracket (**[]**) operators with an index value of **1**.

getStatistics

Signature `getStatistics(stats: JadeDynamicObject input);`

The **getStatistics** method of the **Array** class analyzes the array and returns structural statistics in the attributes of a **JadeDynamicObject**.

The attributes of the dynamic object containing array statistics are defined and interpreted as follows.

Attribute	Description
blockSize	Entries per block
keyLength	Size of the key in bytes (always **4** for an array)
entrySize	Size of each array entry in bytes
size	Number of entries in the array (that is, the size of the array itself)
blockCount	Total number of blocks in the array
height	Number of levels (always **1** for an array)
minEntries	Minimum number of entries found in any block
maxEntries	Maximum number of entries found in any block
avgEntries	Average number of entries in array blocks
loadFactor	Actual average percent loading of array 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 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

includes

Signature `includes(value: MemberType): Boolean;`

The **includes** method of the **Array** class returns **true** if the array contains the object specified in the **value** parameter.

This method returns **false** if the array does not contain the specified object.

indexOf

Signature `indexOf(value: MemberType): Integer;`

The **indexOf** method of the **Array** class returns the index of the entry specified in the **value** parameter if it exists in the array or it returns zero (**0**) if it does not exist. If the specified value occurs more than once in the array, the index of the first occurrence is returned.

The code fragment in the following example shows the use of the **indexOf** method.

```
epilog
    listBoxScrollBar.value := self.theArray.indexOf (myProduct);
    labelRight.caption     :="Time Taken := " & ((app.clock.Time -
                              startTime).Integer/1000).String & " Seconds";
    app.mousePointer:= self.MousePointer_Default;
end;
```

Note Use the **indexOf64** method instead of the **indexOf** method, if the number of entries in the collection could exceed the maximum integer value of 2,147,483,647.

indexOf64

Signature `indexOf64(value: MemberType): Integer64 abstract;`

The **indexOf64** method of the **Array** class returns the index of the entry specified in the **value** parameter if it exists in the array as an **Integer64** value or it returns zero (**0**) if it does not exist.

If the specified value occurs more than once in the array, the index of the first occurrence is returned.

initialise

Signature `initialise(count: Integer64) updating;`

The **initialise** method of the **Array** class initializes an array with null entries up to a size specified by the value of the **count** parameter.

If you know that an array will contain a large number of entries, use the **initialise** method to preallocate space for the array rather than have it grow incrementally.

You can also use the **initialise** method to reinitialize an array efficiently without needing to call the **clear** method.

insert

Signature `insert(index: Integer64;
value: MemberType) updating;`

The **insert** method of the **Array** class inserts the object specified in the **value** parameter into the array at the position specified in the **index** parameter.

Any entry above the insertion point is moved up one slot, thus increasing the size of the array.

last

Signature `last(): MemberType;`

The **last** method of the **Array** class returns a reference to the last entry in the array.

This method is equivalent to using the bracket (**[]**) operators with an index value equal to the size of the array (**array.size**).

remove

Signature `remove(value: MemberType) updating;`

The **remove** method of the **Array** class removes the entry specified in the **value** parameter from an array. Any entry at a higher index is moved down one slot to fill the gap.

If the specified entry occurs more than once in the array, only the first entry is removed. If the specified entry does not exist, an exception is raised.

removeAt

Signature `removeAt(index: Integer64): MemberType updating;`

The **removeAt** method of the **Array** class removes an entry from an array at the position specified in the **index** parameter and moves all entries at a higher index down one slot to fill the gap.

If the specified index does not exist, an exception is raised.

replace

Signature `replace(index: Integer64;
value: MemberType) updating;`

The **replace** method of the **Array** class replaces an existing entry in an array at the position specified by the **index** parameter with the entry specified in the **value** parameter.

If the specified index does not exist, an exception is raised.

BinaryArray Class

The **BinaryArray** class is an ordered collection of **Binary** values with a length less than or equal to 128 bytes. The values are referenced by their position in the collection.

Binary arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a Binary array.

Inherits From: Array

Inherited By: (None)

BooleanArray Class

The **BooleanArray** class is an ordered collection of **Boolean** values in which the values are referenced by their position in the collection.

Boolean arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a Boolean array.

Inherits From: Array

Inherited By: (None)

Btree Class

The **Btree** class is the abstract class that encapsulates behavior required to access entries in a collection by a key, or index.

For details about the method defined in the **Btree** class that is not is not inherited from the **Collection** class, see "Btree Method", in the following subsection.

Inherits From: Collection

Inherited By: Dictionary, Set

Btree Method

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

Method	Description
setLoadFactor	Modifies the default load factor for a Btree-based collection

setLoadFactor

Signature `setLoadFactor(loadFactor: Integer) updating;`

The **setLoadFactor** method of the **Btree** class modifies the default load factor for a Btree-based collection; that is, for dictionaries and sets. (For more details, see "Collections Behavior and Tuning", in Chapter 4 of the *JADE Developer's Reference*.)

The value specified in the **loadFactor** parameter determines the ratio of entries (as a percentage factor) that are moved to a new block when a **Btree** block splits.

Statistically, a 66 percent load factor provides optimal loading when entries are added in random key order and a higher load factor (for example, **95** percent) provides better loading when entries are added in sequential key order.

The default value, specified at the collection class level, is **66** (percent). For details about specifying a sequential load pattern for **Btree** classes if you do not want the default random load pattern, see "Tuning Collection Classes", in Chapter 3 of the *JADE Development Environment User's Guide*.

You can call the **setLoadFactor** method at any time, even on a non-empty collection. When you change the load factor of a collection, an immediate restructure of the Btree does not occur.

Note To adjust the load factor at the class level, use the **Expected Population** text box on the **Tuning** sheet of the Define Class dialog.

ByteArray Class

The **ByteArray** class is an ordered collection of **Byte** values in which the values are referenced by their position in the collection.

Byte arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a Byte array.

For details about the methods defined in the **ByteArray** class, see "ByteArray Methods", in the following section.

Inherits From: Array

Inherited By: (None)

ByteArray Methods

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

Method	Description
binarySearch	Specifies whether the element exists at the position specified by an **Integer** value
binarySearch64	Specifies whether the element exists at the position specified by an **Integer64** value

binarySearch

Signature
```
binarySearch(search: Byte;
             index:  Integer io): Boolean;
```

The **binarySearch** method of the **ByteArray** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter if found or to the position at which it should be added if it does not exist. This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch** method.

```
if not bytes.includes(byte) then
    bytes.binarySearch(byte, pos);
    bytes.insert(pos + 1, byte);
endif;
```

Note Use the **binarySearch64** method instead of the **binarySearch** method, if the number of bytes in the array could exceed the maximum integer value of 2,147,483,647.

binarySearch64

Signature
```
binarySearch64(search: Byte;
               index:  Integer64 io): Boolean;
```

The **binarySearch64** method of the **ByteArray** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter as an **Integer64** value if found or to the position at which it should be added if it does not exist. This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch64** method.

```
if not bytes.includes(byte) then
    bytes.binarySearch64(byte, pos);
    bytes.insert(pos + 1, byte);
endif;
```

CharacterArray Class

The **CharacterArray** class is an ordered collection of **Character** values in which the values are referenced by their position in the collection.

Character arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a Character array.

Inherits From: Array

Inherited By: (None)

Class Class

The **Class** class, a subclass of the **Type** class, is the metaclass of all other JADE classes; that is, all user-defined JADE classes are themselves instances of the **Class** class.

For details about handling class instances, see "Caveat When Handling Persistent Class Instances" and "Caveat When Handling Shared Transient Class Instances", in the following subsections. For details about the properties and methods defined in the **Class** class, see "Class Properties" and "Class Methods", later in this section.

Inherits From: Type

Inherited By: CollClass, ExceptionClass, GUIClass

Caveat When Handling Persistent Class Instances

The **instances** property and the **allInstances**, **firstInstance**, and **lastInstance** methods of the **Class** class employ a database method that retrieves references to instances of a class (and optionally its subclasses) from the physical database.

Note You can use the **allInstances**, **firstInstance**, and **lastInstance** methods and the **instances** property in a production system if you have implemented a mechanism that ensures that there are no uncommitted creates or deletes of instances at the time of execution of the method or during use of the **instances** property.

The collection of instances from the database may not be consistent with updates made by existing uncommitted transactions that are still in a client node cache. For example, if overflow has occurred, the collection may contain instances that have been created, and deleted instances may be missing.

In addition, this type of access has no form of concurrency control that can guarantee a consistent view or "snapshot of instances" for the operation as a whole in a multiuser environment.

In delta database mode, the **instances** property and the **allInstances**, **firstInstance**, and **lastInstance** methods of the **Class** class are executed on both the root and delta database but the merged result set may not be the same as the result set obtained outside of delta mode.

The differences in behavior when using class extent methods in delta mode are as follows.

- Any class extent method can return a reference to an object deleted in delta mode; for example, if the first instance of a class is deleted in delta mode, a reference to this object is still returned by the **Class** class **firstInstance** method.
- The **Class** class **countPersistentInstances** method and calls to **Class.instances.size** include instances that were deleted in delta mode.

For these reasons, unless you have implemented a mechanism to block updates, as stated in the note earlier in this topic, the **instances** property and the **allInstances**, **firstInstance**, and **lastInstance** methods are not recommended for production use in a JADE application. Such use is appropriate only as a development diagnostic or testing aid.

Note To find the number of non-exclusive persistent instances of a class, use the **countPersistentInstances** method of the **Class** class in preference to the **size** method on the **instances** pseudo collection, as shown in the following code fragments.

```
Customer.instances.size;

Customer.countPersistentInstances;  // significantly faster
```

Caveat When Handling Shared Transient Class Instances

The **allSharedTransientInstances**, **firstSharedTransientInstance**, and **lastSharedTransientInstance** methods of the **Class** class employ a method that retrieves references to shared transient instances of a class (and optionally its subclasses) from the transient database.

Note You can use the **allSharedTransientInstances**, **firstSharedTransientInstance**, and **lastSharedTransientInstance** methods in a production system if you have implemented a mechanism that ensures that there are no uncommitted creates or deletes of instances at the time of execution of the method.

The collection of instances from the database may not be consistent with updates made by existing uncommitted transactions that are still in a client node cache. For example, if overflow has occurred, the collection may contain instances that have been created, and deleted instances may be missing.

In addition, this type of access has no form of concurrency control that can guarantee a consistent view or "snapshot of instances" for the operation as a whole in a multiuser environment.

For these reasons, unless you have implemented a mechanism to block updates, as stated in the note earlier in this topic, the **allSharedTransientInstances**, **firstSharedTransientInstance**, and **lastSharedTransientInstance** methods are not recommended for production use in a JADE application. Such use is appropriate only as a development diagnostic or testing aid.

Class Properties

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

Property	Description
implementedInterfaces	Contains references to the interfaces implemented by the class
instances	Contains persistent instances of the class
properties	Contains the dictionary of properties in the class
subclasses	Contains the dictionary of subclasses in the class
superclass	Contains the superclass of the class
transient	Specifies whether instances of the class are transient

implementedInterfaces

Type: JadeInterfaceNDict

The **implementedInterfaces** property of the **Class** class contains references to the JADE interfaces implemented by the receiver.

instances

Type: VirtualColl

The read-only **instances** property of the **Class** class contains a reference to persistent instances of the class that it represents. (See also "Caveat When Handling Persistent Class Instances", earlier in this section.) This property enables you to access persistent class instances as though they were in a class collection without having to populate a collection; for example:

```
vars
    product : Product;
begin
    foreach product in Product.instances do
        product.printDetails;
    endforeach;
end;
```

The order in which instances are returned when iterating a virtual collection is not significant.

The following examples show how the **instances** property, which is a virtual collection, can be used with normal collection methods.

```
count := Employee.instances.size;            // count instances

Employee.instances.purge;                    // delete all instances

emp := Employee.instances.first.Employee;    // first instance

cust := Customer.instances.last.Customer;    // last instance

Product.instances.copy(tempColl);            // copy to temporary collection
```

properties

Type: PropertyNDict

The **properties** property of the **Class** class is a collection of properties in the class.

subclasses

Type: ClassNDict

The **subclasses** property of the **Class** class is a collection of subclasses in the class.

superclass

Type: Class

The **superclass** property of the **Class** class contains a reference to the superclass of the class.

transient

Type: Boolean

The read-only **transient** property of the **Class** class specifies whether instances of the class are transient by default. This default value can be overridden when an object is created. Transient objects exist only for some predetermined time within an application session. They are not stored in the database and are destroyed when explicitly deleted or when the application terminates. (See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.) The following example shows the use of the **transient** property.

```
load() updating;
vars
    cls : Class;
begin
    self.centreWindow;
    app.mousePointer := self.MousePointer_HourGlass;
    foreach cls in currentSchema.getAllClasses(false) do
        if cls.transient = true and
            cls.inheritsFrom(Form) = true then
            comboBoxScreen.addItem(cls.name);
            comboBoxScreen.itemObject[comboBoxScreen.newIndex] := cls;
        endif;
    endforeach;
    app.mousePointer := self.MousePointer_Arrow;
end;
```

Class Methods

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

Method	Description
addDynamicProperty	Adds and returns a new dynamic property to a specified dynamic property cluster in the receiving class
addDynamicPropertyCluster	Adds and returns a new dynamic property cluster to the receiving class
allInstancesInPartition	Fills an array with instances of the receiver class stored in the specified database partition
allInstances	Fills an array with all persistent instances of the receiver class and its subclasses
allLocalSubclasses	Adds all subclasses in the current schema of the receiver to a collection
allProcessTransientInstances	Fills an array with all transient instances of the receiver class created by the current process
allProperties	Returns all properties of the class
allPropertiesUpTo	Returns all properties of superclasses of the receiver up to the specified class
allSharedTransientInstances	Fills an array with all shared transient instances of the receiver class
allSubclasses	Adds all subclasses in the current schema to a collection
allSubclassesInSubschemas	Adds all subclasses in subschemas of the receiver to a collection

Method	Description
allSubclassesUpToSchema	Adds all subclasses of the receiver up to those in the specified schema to a collection
allSuperclassesUpTo	Adds all superclasses of the receiver up to those in the specified class to a collection
anyInstance	Returns **true** if any instances of the receiver class exist
causeClassEvent	Causes a class event
compactDynamicPropertyClusters	Compacts dynamic property clusters in which dynamic property instances were deleted
countPersistentInstances	Returns the number of non-exclusive persistent instances of the receiver class
countPersistentInstances64	Returns the number of non-exclusive persistent instances of the receiver class as an **Integer64** value
countPersistentInstancesLim64	Returns the number of non-exclusive persistent instances of the receiver class up to a specified limit as an **Integer64** value
countPersistentInstancesLimit	Returns the number of non-exclusive persistent instances of the receiver class up to a specified limit
createPartition	Creates a new empty database partition and returns the partition identifier
deleteDynamicProperty	Deletes a dynamic property from a cluster in the receiving class
deleteDynamicPropertyCluster	Deletes a dynamic property cluster from the receiving class
findConstant	Returns the constant with the specified name from the receiver class
findDynamicPropertyCluster	Returns the dynamic property cluster with the specified name from the class of the receiver
findLocalSubclass	Returns the subclass with the specified name from the current schema
findMethodInSubclasses	Returns the method with the specified name from all subclasses
findProperty	Returns the property with the specified name
findPropertyInSubClasses	Returns the property with the specified name from all subclasses
firstInstance	Returns the first persistent instance of the class
firstProcessTransientInstance	Returns the first transient instance of the class in the current process
firstSharedTransientInstance	Returns the first shared transient instance of the class
getConstantInHTree	Returns the constant with the specified name
getDbFile	Returns the database file to which the class is mapped
getImplementors	Adds the classes from all schemas in the hierarchy that implement the specified method to a collection
getMethodInHTree	Returns the method with the specified name
getNextSubClasses	Adds the classes that are subclasses of the current class from the specified collection to a collection
getProperties	Returns the properties of the class of the receiver

Method	Description
getProperty	Returns the specified property from the class of the receiver
getPropertyInHTree	Returns the property with the specified name
getRpsMappingRefs	Adds RPS mappings (instances of the **RelationalView** class) in which the receiver is used
getSubclass	Returns the subclass of the specified receiver
getSubclasses	Adds all subclasses of the receiver class to a collection
getSubclassesUpToSchema	Adds all subclasses of the receiver up to those in the specified schema to a collection
getSuperclass	Returns the superclass of the receiver in the highest level of the schema hierarchy
hasInstance	Returns **true** if an instance of the class exists
hasRpsReferences	Returns **true** if the receiver is referenced by one or more RPS mappings
hasSubclasses	Returns **true** if the class has subclasses
implementsInterface	Returns **true** if the receiver class implements the specified interface
instancesExist	Returns **true** if any instances of the class or its subclasses exist
lastInstance	Returns the last persistent instance of an object in the class
lastProcessTransientInstance	Returns the last transient instance of an object in the class in the current process
lastSharedTransientInstance	Returns the last shared transient instance of an object in the class
needsReorg	Returns **true** if the class requires reorganization
resynchInstances	Discards the replicas of instances of the class
withAllSubclasses	Adds all subclasses of the receiver class to a collection
withAllSuperclasses	Adds all superclasses of the receiver class to a collection

addDynamicProperty

Signature

```
addDynamicProperty(clusterName:  String;
                   propertyName: String;
                   propertyType: Type;
                   length:       Integer;
                   scaleFactor:  Byte): Property;
```

The **addDynamicProperty** method of the **Class** class adds a new runtime dynamic property with the name specified by the **propertyName** parameter name to the dynamic property cluster specified by the **clusterName** parameter in the receiving class and returns the newly created property.

If the type of the property is:

- **Binary**, **String**, or **StringUtf8**, the length must be specified in the **length** parameter and zero (**0**) must be specified for the **scaleFactor** parameter.
- **Decimal**, the precision and number of decimal places must be specified in the **length** and **scaleFactor** parameters, respectively.

If the property is not one of these types, zero (**0**) must be specified as the value for the **length** and **scaleFactor** parameters.

The property name must be unique; that is, different from the names of other static and dynamic properties in the class and in all superclasses and subclasses in the current schema and in all superschemas and subschemas.

The property name must start with a lowercase letter. It can contain alphanumeric characters including underscore characters, but not spaces.

The process must be in transaction state when a dynamic property definition is added. This transaction must be committed before the property can be used.

addDynamicPropertyCluster

Signature `addDynamicPropertyCluster(name: String): JadeDynamicPropertyCluster;`

The **addDynamicPropertyCluster** method of the **Class** class adds a new dynamic property cluster with a name specified in the **name** parameter to the receiving class and returns the dynamic property cluster instance that was created.

The cluster name must be unique within the class and the superschema branch, and within all classes and subschemas. The cluster name can include numbers and underscore characters, but it cannot include punctuation, spaces, or other non-alphanumeric characters.

allInstances

Signature
```
allInstances(objArray:          ObjectArray;
             maxInsts:          Integer64;
             includeSubclasses: Boolean);
```

The **allInstances** method of the **Class** class adds all persistent instances of the receiver class to the array specified in the **objArray** parameter. (Note that the object array is not cleared before instances are added.)

The **maxInsts** parameter specifies the maximum number of instances returned in the **objArray** parameter.

Note The maximum value for the **maxInsts** parameter is 4,294,967,295 (2^{32}-1), which corresponds to the maximum number of entries that can be stored in the **objArray** collection.

If the **includeSubclasses** parameter is set to **true**, all subclasses of the receiver class are also included in the array.

Note As the JADE Inspector uses the **allInstances** method, it is therefore subject to these restrictions. See also "Caveat When Handling Persistent Class Instances", earlier in this section.

The following is an example of the **allInstances** method in which there is no maximum number of instances and logic is requesting instances of **Company** and its subclasses.

```
Company.allInstances(coll, 0, true);
```

allInstancesInPartition

Signature `allInstancesInPartition(partID: Integer64;`
`objArray: ObjectArray;`
`maxInstances: Integer64);`

The **allInstancesInPartition** method of the **Class** class adds instances of the receiver class stored in the partition specified in the **partID** parameter to the array specified in the **objArray** parameter. (Note that the object array is not cleared before instances are added.)

The **maxInstances** parameter specifies the maximum number of instances returned in the **objArray** parameter. A **maxInstances** parameter value of zero (**0**) indicates that there is no maximum number of instances.

Note The maximum value for the **maxInsts** parameter is 4,294,967,295 (2^{32}-1), which corresponds to the maximum number of entries that can be stored in the **objArray** collection.

An exception is raised if the specified partition identifier is out of range or if the file is not partitioned.

allLocalSubclasses

Signature `allLocalSubclasses(subs: ClassColl input);`

The **allLocalSubclasses** method of the **Class** class adds all subclasses in the current schema of the receiver class to the collection specified in the **subs** parameter. (Note that the collection is not cleared before instances are added.)

allProcessTransientInstances

Signature `allProcessTransientInstances(objArray: ObjectArray;`
`maxInsts: Integer;`
`includeSubclasses: Boolean);`

The **allProcessTransientInstances** method of the **Class** class adds all transient instances of the receiver class (and optionally the subclasses of the receiver class) that were created by the current process to the array specified in the **objArray** parameter. (Note that the object array is not cleared before instances are added.)

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.

All transient instances of the subclasses of the receiver class are included in the array when the **includeSubclasses** parameter is set to **true**.

The code fragment in the following example shows the use of the **allProcessTransientInstances** method.

```
create coll transient;
foreach class in classColl do
    class.allProcessTransientInstances(coll, 0, false);
    foreach object in coll do
        if not (object = self or object = caller) then
            count.bump;
            found := true;
            if count = 1 then
                write 'Class - ' & class.name;
            endif;
            write '  Transient - ' & object.String;
        endif;
```

```
        endforeach;
        coll.clear;
        count := 0;
    endforeach;
```

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

allProperties

Signature `allProperties(): PropertyColl;`

The **allProperties** method of the **Class** class returns a reference to all properties of the class of the receiver.

The code fragment in the following example shows the use of the **allProperties** method.

```
// Go through the properties and find the ones that have text.
// Format the text and then write it.
vars
    propertyColl : PropertyColl;
    prop : Property;
begin
    propertyColl := class.allProperties;
    foreach prop in propertyColl do
        if prop.text <> null then
            write prop.name.toUpper & CrLf & prop.text & CrLf;
        endif;
    endforeach;
epilog
    delete propertyColl;
end;
```

allPropertiesUpTo

Signature `allPropertiesUpTo(cls: Class): PropertyColl;`

The **allPropertiesUpTo** method of the **Class** class returns a reference to all properties of superclasses of the receiver in the current schema up to those of the class specified in the **cls** parameter.

Note Only references to superclass properties in the current schema are returned, and not those in superschemas.

allSharedTransientInstances

Signature
```
allSharedTransientInstances(objArray:          ObjectArray;
                            maxInsts:          Integer;
                            includeSubclasses: Boolean);
```

The **allSharedTransientInstances** method of the **Class** class adds all transient instances of the receiver class (and optionally the subclasses of the receiver class) that were created with the **sharedTransient** qualifier (that is, transient objects that can be shared between processes) to the array specified in the **objArray** parameter.

Note The object array is not cleared before instances are added.

The **maxInsts** parameter specifies the maximum number of instances. A **maxInsts** parameter value of zero (**0**) indicates that there is no maximum number of instances.

All transient instances of the subclasses of the receiver class are included in the array when the **includeSubclasses** parameter is set to **true**.

The code fragment in the following example shows the use of this method.

```
create coll transient;
foreach class in classColl do
    class.allSharedTransientInstances(coll, 0, true);
    foreach object in coll do
        if not (object = self or object = caller) then
            count := count + 1;
            found := true;
            if count = 1 then
                write 'Class - ' & class.name;
            endif;
            write '  Transient - ' & object.String;
        endif;
    endforeach;
    coll.clear;
    count := 0;
endforeach;
```

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

allSubclasses

Signature `allSubclasses(subs: ClassColl input);`

The **allSubclasses** method of the **Class** class adds all subclasses in the current schema and superschemas of the receiver class to the collection specified in the **subs** parameter.

allSubclassesInSubschemas

Signature `allSubclassesInSubschemas(subs: ClassColl input);`

The **allSubclassesInSubschemas** method of the **Class** class adds all subclasses in subschemas of the receiver class to the collection specified in the **subs** parameter.

allSubclassesUpToSchema

Signature
```
allSubclassesUpToSchema(topSchema: Schema;
                        subs:      ClassColl input);
```

The **allSubclassesUpToSchema** method of the **Class** class adds all subclasses of the receiver up to those in the schema specified in the **topSchema** parameter to the collection specified in the **subs** parameter.

allSuperclassesUpTo

Signature
```
allSuperclassesUpTo(coll: ClassColl input;
                    cls:  Class);
```

The **allSuperclassesUpTo** method of the **Class** class adds all superclasses of the receiver up to those in the class specified in the **cls** parameter to the collection specified in the **coll** parameter.

anyInstance

Signature `anyInstance(): Boolean;`

The **anyInstance** method of the **Class** class returns **true** if any instances of the receiver class exist and is equivalent to the **firstInstance** **<>** **null** expression.

causeClassEvent

Signature
```
causeClassEvent(eventType: Integer;
                immediate: Boolean;
                userInfo:  Any);
```

The **causeClassEvent** method of the **Class** class causes a class event.

Any client waiting for the specified event for the receiver class receives a user notification event.

The parameters for the **causeClassEvent** method are listed in the following table.

Parameter	Description
eventType	Any number (integer value) selected by the user, in the range **User_Base_Event** through **User_Max_Event**. You can define your own event types in the range **User_Base_Event** through **Max_Integer**.
immediate	Indicates when the event is actioned. If this value is **false**, the notification occurs at the end of the transaction. If this value is **true**, the notification occurs immediately. If the client is not within a begin/commit transaction cycle, the notification waits for the next commit on that client.
userInfo	A value of **Any** primitive type (that is, a string, integer, or character) or object reference that is passed to the **causeClassEvent** event handler when the event is notified. (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.) You should not use a transient object reference across nodes, but you can use a shared transient object reference between applications on the same node.

The following table lists the notification event types.

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

The code fragment in the following example shows the use of the **causeClassEvent** method, where a notification is sent immediately.

```
Customer.causeClassEvent(Refresh_Customer_Views, true, name);
```

compactDynamicPropertyClusters

Signature `compactDynamicPropertyClusters(interval: Integer): receiver: Object; callback: Method; statistics: JadeDynamicObject);`

The **compactDynamicPropertyClusters** method of the **Class** class compacts dynamic property clusters in which dynamic property instances were deleted. This method iterates all instances of the receiving class and subclass. For each instance, it locates any clusters that may contain deleted dynamic property values. If such a cluster exists, deleted values are removed and the cluster is compacted.

You can use this method to reduce the size of dynamic property clusters that contain values for dynamic property definitions that have been deleted.

The **interval**, **receiver**, and **callback** parameters enable you to invoke your own callback to allow the transaction to be committed and restarted periodically. The values of these parameters can all be null, in which case no callback is invoked. The **interval** parameter specifies the period in milliseconds between successive calls to the callback. The **receiver** parameter specifies the receiver of the callback.

The **statistics** parameter, which is also optional, enables some simple statistics to be returned (for example, the number of instances, the number of deleted values, and so on).

The signature of the callback method is:

```
callback(count: Integer): Boolean;
```

The **count** parameter specifies the number of instances in the current class that have been read. The return value is a continue or stop indicator; that is, continue when **true** or stop when **false**.

The following is an example of the **compactDynamicPropertyClusters** method.

```
vars
    statistics    : JadeDynamicObject;
    instanceCount : Integer;
    deletedValues : Integer;
begin
    create statistics;
    beginTransaction;
    // must be in transaction state
        MyClass.compactDynamicPropertyClusters(60000, app,  Application::callback,
                 statistics);
    commitTransaction;
    instanceCount := statistics.getPropertyValue("instanceCount").Integer64;
    // number of instances of MyClass processed
    deletedValues := statistics.getPropertyValue("deletedValues").Integer64;
    // number of deleted dynamic properties removed
end;
```

The following is an example of the **callback** method defined in the **Application** class.

```
callback(count: Integer): Boolean;
// callback called once a minute (60,000 milliseconds) when compacting
// dynamic property clusters
vars
    continueProcessing : Boolean;
begin
```

```
    continueProcessing := true;
    // limit the number of updates in the current transaction: close
    // and re-start the transaction
    commitTransaction;
    beginTransaction;
    // display progress information
    // allow the operation to be cancelled
    return continueProcessing;
end;
```

countPersistentInstances

Signature `countPersistentInstances(): Integer;`

The **countPersistentInstances** method of the **Class** class returns the number of non-exclusive persistent instances of the receiver class.

The **countPersistentInstances** method is significantly faster than the **size** method on an **instances** pseudo-collection, as shown in the following code fragments.

```
write Customer.instances.size;

write Customer.countPersistentInstances;        // significantly faster
```

Note This method impacts other users of the database file where the instances of the receiver class are stored. The best time to run this method is therefore when there are few other active users.

countPersistentInstances64

Signature `countPersistentInstances64(): Integer;`

The **countPersistentInstances64** method of the **Class** class returns the number of non-exclusive persistent instances of the receiver class.

The **countPersistentInstances64** method is significantly faster than the **size** method on an **instances** pseudo-collection, as shown in the following code fragments.

```
write Customer.instances.size;

write Customer.countPersistentInstances64;        // significantly faster
```

Note This method impacts other users of the database file where the instances of the receiver class are stored. The best time to run this method is therefore when there are few other active users.

countPersistentInstancesLim64

Signature `countPersistentInstancesLim64(limit: Integer64): Integer64;`

The **countPersistentInstancesLim64** method of the **Class** class returns the number of non-exclusive persistent instances of the receiver class up to the **Integer64** value specified by the **limit** parameter.

The **countPersistentInstancesLim64** method is significantly faster than the **size** method on an **instances** pseudo-collection, as shown in the following code fragments.

```
write Customer.instances.size;

write Customer.countPersistentInstancesLim64(500000);      // significantly faster
```

This **countPersistentInstancesLim64** method can be used instead of the **countPersistentInstances64** method to limit the time used, but the result will not be correct if the actual count is greater than the value of the **limit** parameter.

Note This method impacts other users of the database file where the instances of the receiver class are stored. The best time to run this method is therefore when there are few other active users.

countPersistentInstancesLimit

Signature `countPersistentInstancesLimit(limit: Integer): Integer;`

The **countPersistentInstancesLimit** method of the **Class** class returns the number of non-exclusive persistent instances of the receiver class up to the value specified by the **limit** parameter.

The **countPersistentInstancesLimit** method is significantly faster than the **size** method on an **instances** pseudo-collection, as shown in the following code fragments.

```
write Customer.instances.size;

write Customer.countPersistentInstancesLimit(500000);      // significantly faster
```

This **countPersistentInstancesLimit** method can be used instead of the **countPersistentInstances** method to limit the time used, but the result will not be correct if actual count is greater than the value of the **limit** parameter.

Note This method impacts other users of the database file where the instances of the receiver class are stored. The best time to run this method is therefore when there are few other active users.

createPartition

Signature `createPartition(): Integer64;`

The **createPartition** method of the **Class** class creates a new empty database partition and returns the partition identifier.

The **createPartition** operation is audited within a database transaction ensuring it is atomic and recoverable. Multiple related **createPartition** operations can be made atomic by containing them in the same database transaction. For example, if related classes such as **Order** and **OrderItem** are both partitioned, instance creation for both of them can be switched to a new current partition within the same database transaction.

A number of related **createPartition** operations can be made atomic by containing them in the same database transaction, as shown in the following example.

```
// execute just before midnight at end of current period
beginTransaction;
Order.createPartition;
OrderItem.createPartition;
// execute just after midnight at start of next period
commitTransaction;
```

An exception is raised if the database file is locked for reorganization or if the file is not partitioned.

The following restrictions apply to the use of the **createPartition** method.

- Partitions can only be created within a transaction
- No other partition creation operation can be in progress

- Persistent objects cannot be created or updated in the transaction that creates a partition
- Persistent objects cannot be created in a partitioned file by any user while a new partition for that file is being created

Note For a production application, developers should implement a synchronization mechanism to prevent the creation of objects stored in a partitioned file while a new partition is created.

deleteDynamicProperty

Signature `deleteDynamicProperty(propertyName: String);`

The **deleteDynamicProperty** method of the **Class** class deletes the runtime dynamic property with the name specified in the **propertyName** parameter from a cluster in the receiving class.

Note If instances of the class exist, you cannot delete a dynamic property. A non-embedded blob, slob, or slobutf8 dynamic property has a size or length greater than 540.

The maximum length of an embedded **Binary** type is 540 characters, an embedded **StringUtf8** type is 540 characters, and an embedded **String** type is 539 characters.

You can delete a runtime dynamic property only if the class in which it is defined is not being used by any other process. If production mode is set, a runtime dynamic property can be deleted in single-user mode only.

An exception is raised if the runtime dynamic property is not defined in the class.

deleteDynamicPropertyCluster

Signature `deleteDynamicPropertyCluster(name: String);`

The **deleteDynamicPropertyCluster** method of the **Class** class deletes the dynamic property cluster specified in the **name** parameter from the receiving class.

You can delete a dynamic property cluster only if the class in which it is defined is not being used by any other process and there are no instances of this class or any subclass. If production mode is set, a dynamic property cluster can be deleted in single user mode only.

findConstant

Signature `findConstant(str: String): Constant;`

The **findConstant** method of the **Class** class returns a reference to the constant with the name specified in the **str** parameter from the receiver.

findDynamicPropertyCluster

Signature
```
findDynamicPropertyCluster(name: String):
            JadeDynamicPropertyCluster;
```

The **findDynamicPropertyCluster** method of the **Class** class returns the dynamic property cluster with the name specified in the **name** parameter from the class of the receiver or it returns null if no cluster with the specified name is defined.

findLocalSubclass

Signature `findLocalSubclass(subName: String): Class;`

The **findLocalSubclass** method of the **Class** class returns a reference to the subclass with the name specified in the **subName** parameter.

A recursive search down through the subclass hierarchy in the current schema only is performed.

findMethodInSubclasses

Signature `findMethodInSubclasses(str: String): Method;`

The **findMethodInSubclasses** method of the **Class** class returns a reference to the method with the name specified in the **str** parameter.

A recursive search down through the subclass hierarchy is performed.

findProperty

Signature `findProperty(str: String): Property;`

The **findProperty** method of the **Class** class returns a reference to the property with the name specified in the **str** parameter if the property exists in the receiving class or any of its superclasses.

findPropertyInSubClasses

Signature `findPropertyInSubClasses(str: String): Property;`

The **findPropertyInSubClasses** method of the **Class** class returns a reference to the property with the name specified in the **str** parameter if the property exists in any of the subclasses of the receiving class.

firstInstance

Signature `firstInstance(): InstanceType;`

The **firstInstance** method of the **Class** class returns a reference to the first instance of the class. (See also "Caveat When Handling Persistent Class Instances", earlier in this section.)

The following examples show the use of the **firstInstance** method.

```
deleteAllInvestors() updating;
vars
    object : Object;
begin
    beginTransaction;
        app.setMarket(Market.firstInstance);
        foreach object in app.myMarket.allInvestors do
            delete object;
        endforeach;
    commitTransaction;
end;

// Check for first company setup
app.myCompany := Company.firstInstance;
if app.myCompany = null then
```

```
        beginTransaction;
            create coy;
            app.myCompany := coy;
        commitTransaction;
    endif;
```

firstProcessTransientInstance

Signature `firstProcessTransientInstance(): InstanceType;`

The **firstProcessTransientInstance** method of the **Class** class returns a reference to the first transient instance of the class that was created by the current process.

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

firstSharedTransientInstance

Signature `firstSharedTransientInstance(): InstanceType;`

The **firstSharedTransientInstance** method of the **Class** class returns a reference to the first shared transient instance of the class that was created with the **sharedTransient** qualifier; that is, a transient object that can be shared between processes.

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

getConstantInHTree

Signature `getConstantInHTree(name: String): Constant;`

The **getConstantInHTree** method of the **Class** class returns a reference to the constant with the name specified in the **name** parameter. If the constant does not exist in the receiver class, superclasses are searched.

getDbFile

Signature `getDbFile(): DbFile;`

The **getDbFile** method of the **Class** class returns a reference to the database file to which the class is mapped.

getImplementors

Signature
```
getImplementors(methodName: String;
                methSet:    MethodSet input);
```

The **getImplementors** method of the **Class** class adds the classes from all schemas in the hierarchy that implement the method specified in the **methodName** parameter to the collection specified in the **methSet** parameter.

getMethodInHTree

Signature `getMethodInHTree(name: String): Method;`

The **getMethodInHTree** method of the **Class** class returns a reference to the method with the name specified in the **name** parameter.

If the method does not exist in the receiver class, superclasses are searched.

getNextSubClasses

Signature `getNextSubClasses(currentColl: ClassColl; subclassColl: ClassColl input);`

The **getNextSubClasses** method of the **Class** class adds the classes that are subclasses of the current class from the collection specified in the **currentColl** parameter to the collection specified in the **subclassColl** parameter. (Note that the collection is not cleared before instances are added.)

getProperties

Signature `getProperties(): PropertyNDict;`

The **getProperties** method of the **Class** class returns a collection in the root type object of the properties of the class of the receiver.

getProperty

Signature `getProperty(propName: String): Property;`

The **getProperty** method of the **Class** class returns a reference to the property with the name specified in the **propName** parameter from the class of the receiver.

Use the **findProperty** method if you want to find the property in the class of the receiver or any of its superclasses.

As a subschema copy of the class is used if it exists, call the **getPropertyInHTree** method if you want to use the property from the root definition.

getPropertyInHTree

Signature `getPropertyInHTree(name: String): Property;`

The **getPropertyInHTree** method of the **Class** class returns a reference to the property with the name specified in the **name** parameter.

If the property does not exist in the receiver class, superclasses are searched.

getRpsMappingRefs

Signature `getRpsMappingRefs(rpsMapSet: ObjectSet input);`

The **getRpsMappingRefs** method of the **Class** class adds any RPS mappings (that is, instances of the **RelationalView** class) in which the receiver is used to the input **rpsMapSet** parameter. The **rpsMapSet** collection is not cleared before the RPS mappings are added.

getSubclass

Signature `getSubclass(name: String): Class;`

The **getSubclass** method of the **Class** class returns a reference to the subclass of the receiver specified in the **name** parameter.

getSubclasses

Signature `getSubclasses(subs: ClassColl input);`

The **getSubclasses** method of the **Class** class adds the immediate subclasses of the receiver (that is, the next lowest level of classes in the hierarchy) to the collection specified in the **subs** parameter.

getSubclassesUpToSchema

Signature
```
getSubclassesUpToSchema(topSchema: Schema;
                        subs:      ClassColl input);
```

The **getSubclassesUpToSchema** method of the **Class** class adds all immediate subclasses of the receiver up to those in the schema specified in the **topSchema** parameter to the collection specified in the **subs** parameter. (Note that the collection is not cleared before instances are added.)

getSuperclass

Signature `getSuperclass(): Class;`

The **getSuperclass** method of the **Class** class returns a reference to the superclass of the receiver in the highest level of the schema hierarchy.

hasInstance

Signature `hasInstance(object: Object): Boolean;`

The **hasInstance** method of the **Class** class returns **true** if the value specified in the **object** parameter is or has been an instance of the receiver class.

This method is similar to the **isKindOf** method of the **Object** class, except that the **object** parameter can refer to a deleted object; for example, in a delete notification. However the **isKindOf** method executes faster than the **hasInstance** method.

hasRpsReferences

Signature `hasRpsReferences(): Boolean;`

The **hasRpsReferences** method of the **Class** class returns **true** if the receiver is referenced by one or more RPS mappings.

hasSubclasses

Signature `hasSubclasses(): Boolean;`

The **hasSubclasses** method of the **Class** class returns **true** if the class has subclasses.

implementsInterface

Signature `implementsInterface(jinf: JadeInterface): Boolean;`

The **implementsInterface** method of the **Class** class returns **true** if the receiver class implements the interface specified in the **jinf** parameter.

instancesExist

Signature `instancesExist(): Boolean;`

The **instancesExist** method of the **Class** class returns **true** if any instances of the class or its subclasses exist.

lastInstance

Signature `lastInstance(): InstanceType;`

The **lastInstance** method of the **Class** class returns a reference to the last instance of the class.

The code fragment in the following example shows the use of the **lastInstance** method.

```
if Customer.firstInstance = null then
    instancesTable.text := "0010";
else     // add 10 to last one used; that is, sequentially allocate
         // the next free Customer number
    instancesTable.text := (Customer.lastInstance.customerNumber +
                            10).String.padLeadingZeros(4);
endif;
```

See also "Caveat When Handling Persistent Class Instances", earlier in this section.

lastProcessTransientInstance

Signature `lastProcessTransientInstance(): InstanceType;`

The **lastProcessTransientInstance** method of the **Class** class returns a reference to the last transient instance of the class that was created by the current process.

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

lastSharedTransientInstance

Signature `lastSharedTransientInstance(): InstanceType;`

The **lastSharedTransientInstance** method of the **Class** class returns a reference to the last shared transient instance of the class that was created with the **sharedTransient** qualifier; that is, a transient object that can be shared between processes.

See also "Caveat When Handling Shared Transient Class Instances", earlier in this section.

needsReorg

Signature `needsReorg(): Boolean;`

The **needsReorg** method of the **Class** class returns **true** if the class of the receiver requires reorganization.

For details about reorganizing your JADE database, see Chapter 14 of the *JADE Developer's Reference*.

resynchInstances

Signature `resynchInstances();`

The **resynchInstances** method of the **Class** class discards the replicas of instances of the receiver class from cache.

The replicas are discarded immediately and a new copy of an instance is replicated from the server when it is required on the client.

withAllSubclasses

Signature `withAllSubclasses(coll: ClassColl input);`

The **withAllSubclasses** method of the **Class** class adds all subclasses of the receiver class up to and including the receiver class to the collection specified in the **coll** parameter. (Note that the collection is not cleared before instances are added.)

The following code fragment adds subclasses of the **BankAccount** class to a **ClassColl** collection.

```
vars
    classColl: ClassColl;
begin
    create classColl transient;
    BankAccount.withAllSubclasses(classColl);
```

When a class name is coded in a method (and colored green in the editor) the reference is evaluated to the root type of the class. In the following code fragment, only subclasses of the **MemberKeyDictionary** class from the **RootSchema** are added to the **ClassColl** collection; that is, local subclasses are not included.

```
vars
    classColl: ClassColl;
begin
    create classColl transient;
    MemberKeyDictionary.withAllSubclasses(classColl);
```

To find local subclasses of **MemberKeyDictionary**, use the **getClass** method of the **Schema** class to specify that the copy of **MemberKeyDictionary** in the current schema is to be used, as shown in the following code fragment.

```
vars
    classColl: ClassColl;
begin
    create classColl transient;
    currentSchema.getClass("MemberKeyDictionary").withAllSubclasses(classColl);
```

withAllSuperclasses

Signature `withAllSuperclasses(coll: ClassColl input);`

The **withAllSuperclasses** method of the **Class** class adds all superclasses of the receiver class to the collection specified in the **coll** parameter. (Note that the collection is not cleared before instances are added.)

CMDialog Class

The **CMDialog** class encapsulates behavior for the common dialog subclasses, which provide an interface between JADE logic and the environmental dialogs (Windows **comdlg32** library file). To initiate a common dialog using these classes, the **comdlg32** library file must be in your path.

To invoke a common dialog, perform the following actions

1. Define a local instance of the class.
2. Create the instance.
3. Set any options for the common dialog, by using its properties.
4. Call the **open** method for the instance.
5. Optionally delete the instance when it is no longer required.

The following example shows the invocation of the common Color dialog.

```
vars
    colDlg : CMDColor;
begin
    create colDlg;
    colDlg.color := backColor;            // set initial displayed color
    if colDlg.open = 0 then               // not cancelled and no error
        backColor := colDlg.color;        // now use the returned value
    endif;
epilog
    delete colDlg;                        // tidy
end;
```

The common classes that are subclasses of the **CMDialog** class are listed in the following table.

Subclass	Description
CMDColor	Color dialog
CMDFileOpen	File Open dialog
CMDFileSave	Save dialog
CMDFont	Font dialog
CMDPrint	Print or Printer Selection dialog

The created common dialog instance is transient and does not require transaction state. The created instance is discarded on completion of the application, if it is not deleted.

Notes This class is not available in a server node.

In JADE thin client mode, common dialogs execute on the presentation client and return information relative to the presentation client.

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

Inherits From: Object

Inherited By: CMDColor, CMDFileOpen, CMDFileSave, CMDFont, CMDPrint

CMDialog Properties

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

Property	Description
dialogTitle	Contains the string displayed in the title bar of the dialog for the common dialogs
helpContextId	Specifies the keyword of the topic displayed when the user clicks the **Help** button
helpFile	Causes the **Help** button to be displayed
helpKeyword	Contains the keyword of the help topic to be displayed

dialogTitle

Type: String

Default Value: Common dialog type

The **dialogTitle** property of the **CMDialog** class contains the string displayed in the title bar of the dialog for the common dialogs. Use the **dialogTitle** property to display your own title in the title bar of the common dialogs rather than the default title.

helpContextId

Type: Integer

Default Value: 0

The **helpContextId** property of the **CMDialog** class contains an associated context number for an object. This property is used to provide context-sensitive help for your common dialogs. 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 (that is, a .**hlp** or .**chm** file) for your application, when a user presses F1, JADE automatically calls help and requests the topic identified by the current context number (or the **helpKeyword** property).

If this property is set to zero (**0**) and its **helpKeyword** property value is **null**, 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.

helpFile

Type: String

Default Value: None

The **helpFile** property of the **CMDialog** class causes the **Help** button to be displayed on the common dialogs if one of the **helpKeyword** or **helpContextId** properties is also set. (If not, it is ignored.) The **helpFile** value provides the dialog with the help file from which the help topic is drawn. The help file can be Hypertext Markup Language (**.htm** or **.html**) files, an Adobe Acrobat Portable Document Format (.**pdf**) file, a Windows help (.**hlp**) file, or a compiled help (.**chm**) file.

Note Building a help file requires Adobe Acrobat, the Microsoft Windows Help Compiler, or any other Windows help compiler.

helpKeyword

Type: String

Default Value: Null

If a help keyword is provided for a common dialog, the **helpKeyword** property of the **CMDialog** class contains the text that is used to access the help file when the user presses F1 for help while the focus is on that common dialog. This property must be set before the dialog is initiated by using the **open** method of the **CMDialog** subclass. The current keyword is the value of the **helpKeyword** property for the object that has the focus. If the **helpKeyword** property is empty and its **helpContextId** property is set to zero (**0**), 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 help keyword is used in preference to the help context number. When a user presses F1 to request help, 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 Adobe 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 of 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 a help request other than one using 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.

- The **helpKeyword** property can contain a help file name before the keyword, separated by a semicolon. This help file (which can be a .**pdf**, .**hlp**, or .**chm** file) is specific to this **helpKeyword** property, and overrides the default value, as shown in the following examples.

```
btnHelp_click(btn: Button input) updating;
vars
begin
    if fldFolder.topSheet = shtSelect then
        btn.helpKeyword := "DevRef.pdf;selectinglibraryacxautomationdrg10";
    elseif fldFolder.topSheet =  shtLibrary then
        btn.helpKeyword := "DevRef.pdf;namelibrary_activex";
    elseif fldFolder.topSheet =  shtObjects then
        btn.helpKeyword := "DevRef.pdf;namingobjectclassesacxautomationdrg10";
    elseif fldFolder.topSheet =  shtInterfaces then
        btn.helpKeyword := "DevRef.pdf;naminginterfacesacxautomationdrg10";
    elseif fldFolder.topSheet =  shtConstants then
        btn.helpKeyword := "DevRef.pdf;namingconstantsacxautomationdrg10";
```

```
    endif;
    btn.showHelp;
end;

myForm.helpKeyword := "formHelpfile.pdf;formKeyword";
```

Tip Although it is more efficient to use a single help file, specified in the **Help File** text box on the **Application** sheet of the Define Application dialog, this feature is intended for situations in which multiple help files are required for a single application.

- When handling automatic Help menu items, if a **helpContextId** or **helpKeyword** property is specified on the Help menu **Index** automatic menu item, the destination of the help is based on the value of the **helpContextId** or **helpKeyword** property. In addition, the **click** event method is not executed.

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, the Microsoft Windows Help Compiler, or any other Windows help compiler.

CMDColor Class

The **CMDColor** class enables access to the common Color dialog facility for the selection of colors.

Notes This class is not available on a server node.

In JADE thin client mode, the common Color dialog executes on the presentation client and returns information relative to the presentation client.

For details about the properties and method defined in the **CMDColor** class, see "CMDColor Properties" and "CMDColor Method", in the following subsections.

Inherits From: CMDialog

Inherited By: (None)

CMDColor Properties

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

Property	Description
color	Contains the selected color for the Color dialog
customColors	Contains an array of custom colors
fullOpen	Specifies whether the entire common Color dialog is displayed when the dialog is created
preventFullOpen	Disables the **Define Custom Colors** button on the common Color dialog

color

Type: Integer

Default Value: Black

The **color** property of the **CMDColor** class contains the selected color for the common Color dialog.

Set the **color** property to determine the color that is selected on the displayed color dialog. The selected color is returned in this property if the common Color dialog is not cancelled.

customColors

Type: Integer Array

The **customColors** property of the **CMDColor** class contains a reference to an array of custom colors for the common Color dialog.

The array contains entries that are initialized to the Windows default values. This array can be changed to add user-defined custom colors. Any custom colors set in the common dialog by users are returned.

Note Only the first 16 entries of the array are used.

Colors customized in the common dialog are retained during an application session. When you create a **CMDColor** object, custom colors are initialized from the internal custom color list of the application. When the color dialog closes, the application custom color list is updated from the returned custom colors list. Deleting the **CMDColor** object and then subsequently creating another therefore retains any custom colors that were set during the same application session.

fullOpen

Type: Boolean

Default Value: Initial value is false

Set the **fullOpen** property of the **CMDColor** class to **true** to cause the entire common Color dialog to be displayed when the dialog is created, including the portion that enables the user to create custom colors.

If the value is **false**, the user must click the **Define Custom Colors** button to display this portion of the dialog.

preventFullOpen

Type: Boolean

Default Value: True

Set the **preventFullOpen** property of the **CMDColor** class to **false** to enable the **Define Custom Colors** button on the common Color dialog.

This allows the user to define custom colors or to close that portion of the dialog, depending on the setting of the **fullOpen** property.

CMDColor Method

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

Method	Description
open	Initiates the Color dialog for the CMDColor class

open

Signature `open(): Integer updating;`

The **open** method of the **CMDColor** class initiates the common Color dialog for the **CMDColor** class and returns a value indicating the success of the user actions, as listed in the following table.

Value	Description
0	User clicked the **OK** button. Values of the selections made are returned.
1	User clicked the **Cancel** button.
Other	Error number indicating a Windows fault number associated with the execution of the dialog.

The position of the displayed common Color dialog cannot be determined. The values of the variable options for the class should be set before the **open** method is used.

The following example shows the use of the **open** method to open the common Color dialog.

```
vars
    colDlg : CMDColor;
begin
    create colDlg;
    colDlg.color := backColor;              // set initial color displayed
    if colDlg.open = 0 then                 // not cancelled and no error
        backColor := colDlg.color;          // use the returned value
    endif;
epilog
    delete colDlg;                          // tidy
end;
```

Notes Any values that are returned are retained if further calls are made on the class instance. The object should be deleted when it is no longer required.

An exception is raised if this method is invoked from a server method.

CMDFileOpen Class

The **CMDFileOpen** class enables access to the common File Open dialog facility for the selection of files, by specifying the names and locations of files that are read by your application. The following example, which uses the **CMDFileOpen** class:

1. Displays a File Open dialog
2. Obtains the file selected by the user
3. Uses the data read from the file to load the logo from the selected file

```
logo_click(picture: Picture);
vars
    fileLogo : CMDFileOpen;
begin
    create fileLogo;
    if fileLogo.open = 0 then
        logo.picture := app.loadPicture(fileLogo.fileName);
        self.formatPicture;
    endif;
end;
```

Notes This class is not available on a server node.

In JADE thin client mode, the common File Open dialog executes on the presentation client and returns information relative to the presentation client.

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

Inherits From: CMDialog

Inherited By: (None)

CMDFileOpen Properties

The properties defined in the **CMDFileOpen** class are summarized in the following table. All properties must be set before the **open** method that invokes the dialog is called.

Property	Description
allowMultiSelect	Specifies whether the **File Name** list box allows multiple selections
createPrompt	Specifies whether the user is prompted to create a file that does not currently exist
defaultExt	Contains the default file name extension for the common File Open dialog
extensionDifferent	Specifies that the extension of the returned file name differs from the extension
fileMustExist	Specifies that the user can enter names of existing files only
fileName	Contains the path and file name of a selected file
fileTitle	Contains the name without the path of the file to open
filter	Contains the filters that are displayed in the **Type** list box

Property	Description
filterIndex	Contains the default filter
hideReadOnly	Specifies whether the **Read Only** check box is visible
initDir	Contains the initial file directory
noReadOnlyReturn	Specifies whether the returned file can have the **Read Only** attribute set
pathMustExist	Specifies whether the user can enter only valid path names
readOnly	Indicates the state of the **Read Only** check box when the dialog is closed
resetCurrentPath	Specifies whether the current path is reset
shareAware	Specifies whether network sharing violation errors are ignored
validate	Specifies whether the common dialog allows invalid characters in the file name

allowMultiSelect

Type: Boolean

Default Value: False

The **allowMultiSelect** property of the **CMDFileOpen** class specifies whether the **File Name** list box of the common File Open dialog allows multiple selections.

The user can select more than one file at run time, by pressing the Shift key and using the arrow keys to select the required files or by pressing the Ctrl key and clicking files to selectively add them to the list.

The **fileName** property is returned as a string containing a reference to the names of all selected files, with the file names in the string delimited by spaces and preceded by the path name. (See the **getMultiSelectCount**, **getMultiSelectDirectory**, and **getMultiSelectFileTitle** methods, which enable you to access the **fileName** property string when file names contain embedded spaces.)

createPrompt

Type: Boolean

Default Value: True

Set the **createPrompt** property of the **CMDFileOpen** class to **true** to specify that the common File Open dialog should ask if the user wants to create a file that does not currently exist.

defaultExt

Type: String

Default Value: Null

The **defaultExt** property of the **CMDFileOpen** class contains the default file name extension for the common File Open dialog.

Use this property to specify a default file name extension; for example, .**txt** or .**doc**.

When you define a value for the **defaultExt** property, you must also define the **filter** property value. If the specified **defaultExt** property value does not exist in the filter string, it defaults to **All Files|*.***.

The following is an example of the **defaultExt** and **filter** properties.

```
vars
   trCMD : CMDFileOpen;
begin
   create trCMD transient;
   trCMD.filter := "Text (*.txt)|*.txt|Pictures(*.bmp;*.ico)|*.bmp;*.ico|All
                   Files|*.*";
   trCMD.defaultExt := ".txt";
   if trCMD.open = 0 then
       // do some processing here
   endif;
epilog
   delete trCMD;
end;
```

extensionDifferent

Type: Boolean

The **extensionDifferent** property of the **CMDFileOpen** class specifies whether the extension of the returned file name differs from that of the extension specified by the **defaultExt** property for the File common Open dialog.

This property has meaning only after successfully returning from the dialog initiated by the **open** method.

fileMustExist

Type: Boolean

Default Value: True

Set the **fileMustExist** property of the **CMDFileOpen** class to **true** to specify that the user can enter only names of existing files in the **File Name** text box of the common File Open dialog.

If the value of this property is **true** and the user enters an invalid file name, a warning is displayed.

fileName

Type: String

Default Value: Null

The **fileName** property of the **CMDFileOpen** class contains the path and file name of a selected file for the common File Open dialog.

When the control is created at run time, the **fileName** property is set to a null string (**""**), indicating that there is no currently selected file.

To set an initial file name, set the **fileName** property before calling the **open** method.

If the **allowMultiSelect** property is set to **true** and more than one file is selected, the returned format is as follows.

```
directory-name first-file-name second-file-name ...
```

For example:

```
c:\dir f1.typ f2.typ f3.typ
```

If the **allowMultiSelect** property is set to **true** and more than one file is selected, the **fileTitle** property is not set.

See the **getMultiSelectCount**, **getMultiSelectDirectory**, and **getMultiSelectFileTitle** methods, which enable you to access the **fileName** property string when file names contain embedded spaces and the **allowMultiSelect** property is set to **true**.

fileTitle

Type: String

The **fileTitle** property of the **CMDFileOpen** class contains the name without the path of the file to open from the common File Open dialog. When the user selects a file and clicks the **OK** button in the dialog, the **fileTitle** property contains a value that can then be used to open the selected file.

Note The **fileTitle** property does not contain a value if the **validate** property is set to **false**. If the **allowMultiSelect** property is set to **true** and more than one file is selected, the **fileTitle** property is not set.

filter

Type: String

Default Value: "All files|*.*"

The **filter** property of the **CMDFileOpen** class contains the filters that are displayed in the **Type** list box of the common File Open dialog. A filter specifies the type of files that are displayed in the **File** list box of the dialog; for example, select the ***.txt** filter to display all text files.

Use this property to provide the user with a list of filters that can be selected when the dialog is displayed. Each type consists of two parts, as follows.

- A description that is displayed to the user
- The actual file type; for example, ***.txt** for text files

Use the pipe (**|**) symbol to separate each of the description and filter arguments. Do not include spaces before or after the pipe symbol, as these spaces are then displayed with the description and filter values.

The following text shows an example of a filter that enables the user to choose text files or picture files that include bitmaps and icons.

```
Text (*.txt)|*.txt|Pictures(*.bmp;*.ico)|*.bmp;*.ico
```

Use the **filterIndex** property to determine which filter is displayed as the default when you specify more than one filter for a dialog.

filterIndex

Type: Integer

Default Value: 1

The **filterIndex** property of the **CMDFileOpen** class contains the filter listed in the **filter** property that is the default for the common File Open dialog.

hideReadOnly

Type: Boolean

Default Value: False

The **hideReadOnly** property of the **CMDFileOpen** class specifies whether the **Read Only** check box is visible in the common File Open dialog. Set this property to **true** if you want to hide the display of the **Read-Only** check box.

initDir

Type: String

Default Value: Current directory

The **initDir** property of the **CMDFileOpen** class contains the initial file directory for the common File Open dialog.

Note The **resetCurrentPath** property has no effect when you specify an initial file directory.

noReadOnlyReturn

Type: Boolean

Default Value: False

Set the **noReadOnlyReturn** property of the **CMDFileOpen** class to **true** to specify that the file returned from the common File Open dialog cannot have the **Read Only** attribute set and cannot be in a write-protected directory.

pathMustExist

Type: Boolean

Default Value: True

Set the **pathMustExist** property of the **CMDFileOpen** class to **false** to specify that the user can enter path names that do not exist (that is, which are currently invalid) in the common File Open dialogs.

When this property is set to the default value of **true** and the user enters an invalid path name, a warning message is displayed.

readOnly

Type: Boolean

Default Value: False

Set the **readOnly** property of the **CMDFileOpen** class to **true** to specify that the **Read Only** check box in the common File Open dialog is initially checked when the dialog is created.

This property also indicates the state of the **Read Only** check box when the dialog is closed.

resetCurrentPath

Type: Boolean

Default Value: False

Set the **resetCurrentPath** property of the **CMDFileOpen** class to **true** to specify that the current path in the common File Open dialog is reset back to its value when the dialog was initiated.

Setting the property to **false** indicates that the current path is set to the last path selected by the user of the common dialog.

Note The **resetCurrentPath** property has no effect when a value is specified for the **initDir** property.

shareAware

Type: Boolean

Default Value: False

Set the **shareAware** property of the **CMDFileOpen** class to **true** to specify that network sharing violation errors are ignored in the common File Open dialog.

validate

Type: Boolean

Default Value: True

Set the **validate** property of the **CMDFileOpen** class to **false** to specify that the common dialog allows invalid characters in the file name returned from the common File Open dialog.

CMDFileOpen Methods

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

Method	Description
getMultiSelectCount	Returns the number of files selected when the **allowMultiSelect** property is set to **true**
getMultiSelectDirectory	Returns the directory of the files selected when the **allowMultiSelect** property is set to **true**
getMultiSelectFileTitle	Returns the title of the specified file when the **allowMultiSelect** property is set to **true**
open	Initiates the common File Open dialog

getMultiSelectCount

Signature `getMultiSelectCount(): Integer;`

The **getMultiSelectCount** method of the **CMDFileOpen** class returns the number of files selected in the **File Name** list box when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false**, this method returns a **null** value.

getMultiSelectDirectory

Signature `getMultiSelectDirectory(): String;`

The **getMultiSelectDirectory** method of the **CMDFileOpen** class returns the name of the directory in which the files selected in the **File Name** list box are located when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false**, this method returns a **null** value.

getMultiSelectFileTitle

Signature `getMultiSelectFileTitle(indx: Integer): String;`

The **getMultiSelectFileTitle** method of the **CMDFileOpen** class returns the title of the file specified in the **indx** parameter (that is, the name of the file without the path) when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false** or the value specified in the **indx** parameter is invalid, this method returns a **null** value.

open

Signature `open(): Integer updating;`

The **open** method of the **CMDFileOpen** class initiates the common File Open dialog and returns a value indicating the success of the user actions, as listed in the following table.

Value	Description
0	User clicked the **OK** button. Values of the selections made are returned.
1	User clicked the **Cancel** button.
Other	Windows error number indicating a fault associated with the execution of the dialog.

The position of the common File Open dialog cannot be determined. The values of the variable options for the class should be set before the **open** method is used.

The following example shows the use of the **open** method to open a common File Open dialog.

```
vars
    fopen : CMDFileOpen;
begin
    create fopen;
    fopen.initDir := app.dbPath;        // set the initial directory
    if fopen.open = 0 then              // not cancelled and no error
        self.name := fopen.fileTitle;   // use the returned value
    endif;
epilog
    delete fopen;                       // tidy
end;
```

Notes Any values that are returned are retained if further calls are made on the class instance. The object should be deleted when it is no longer required.

An exception is raised if this method is invoked from a server method.

CMDFileSave Class

The **CMDFileSave** class enables access to the common File Save dialog facility for the selection of files, by specifying the names and locations of files that are saved by your application.

Notes This class is not available on a server node.

In JADE thin client mode, the common File Save dialog executes on the presentation client and returns information relative to the presentation client.

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

Inherits From: CMDialog

Inherited By: (None)

CMDFileSave Properties

The properties defined in the **CMDFileSave** class are summarized in the following table. These properties must be set before the **open** method that invokes the dialog is called.

Property	Description
allowMultiSelect	Specifies whether the **File Name** list box allows multiple selections
createPrompt	Specifies that the user is prompted to create a file that does not currently exist
defaultExt	Contains the default file name extension for the common File Save dialog
extensionDifferent	Specifies whether the file name extension is different
fileMustExist	Specifies whether the user can enter only names of existing files
fileName	Contains the path and file name of a selected file
fileTitle	Contains the name without the path of the file to save
filter	Contains the filters that are displayed in the **Type** list box
filterIndex	Contains the default filter
hideReadOnly	Specifies whether the **Read Only** check box is visible
initDir	Contains the initial file directory
noReadOnlyReturn	Specifies whether the returned file can have the **Read Only** attribute set
overwritePrompt	Specifies whether a message box is generated if the selected file already exists
pathMustExist	Specifies whether the user can enter only valid path names
readOnly	Indicates the state of the **Read Only** check box when the dialog is closed
resetCurrentPath	Specifies whether the current path is reset
shareAware	Specifies whether network sharing violation errors are ignored
validate	Specifies whether the common dialog allows invalid characters in the file name

allowMultiSelect

Type: Boolean

Default Value: False

The **allowMultiSelect** property of the **CMDFileSave** class specifies whether the **File Name** list box of the common File Save dialog allows multiple selections.

The user can select more than one file at run time, by pressing the Shift key and using the arrow keys to select the required files.

The **fileName** property is returned as a string containing the names of all selected files, with the file names in the string delimited by spaces, preceded by the directory name. (See the **getMultiSelectCount**, **getMultiSelectDirectory**, and **getMultiSelectFileTitle** methods, which enable you to access the **fileName** property string when file names contain embedded spaces.)

createPrompt

Type: Boolean

Default Value: True

Set the **createPrompt** property of the **CMDFileSave** class to **true** to specify that the common File Save dialog asks if the user wants to create a file that does not currently exist.

defaultExt

Type: String

Default Value: Null

The **defaultExt** property of the **CMDFileSave** class contains the default file name extension for the common File Save dialog.

Use this property to specify a default file name extension; for example, .**txt** or .**doc**.

When a file with no extension is saved, the extension specified by this variable is automatically appended to the file name.

When you define a value for the **defaultExt** property, you must also define the **filter** property value. If the specified **defaultExt** property value does not exist in the filter string, it defaults to **All Files|*.***.

extensionDifferent

Type: Boolean

Set the **extensionDifferent** property of the **CMDFileSave** class to **true** to specify that the extension of the returned file name differs from the extension specified by the **defaultExt** property for the common File Save dialog.

This property has meaning only after successfully returning from the dialog initiated by the **open** method.

fileMustExist

Type: Boolean

Default Value: True

Set the **fileMustExist** property of the **CMDFileSave** class to **true** to specify that the user can enter only names of existing files in the **File Name** text box of the common File Save dialog. If the value of this property is **true** and the user enters an invalid file name, a warning is displayed. As this method currently does not work because of a Microsoft problem, you should use the **CMDFileOpen** class **fileMustExist** property if you require this functionality.

fileName

Type: String

Default Value: Null

The **fileName** property of the **CMDFileSave** class contains the path and file name of a selected file for the common File Save dialog.

When the control is created at run time, the **fileName** property is set to a null string (**""**), meaning that there is no currently selected file.

To set an initial file name, set the **fileName** property before calling the **open** method.

If the **allowMultiSelect** property is set to **true** and more than one file is selected, the returned format is as follows.

```
directory-name first-file-name second-file-name ...
```

For example:

```
c:\dir f1.typ f2.typ f3.typ
```

If the **allowMultiSelect** property is set to **true** and more than one file is selected, the **fileTitle** property is not set.

See the **getMultiSelectCount**, **getMultiSelectDirectory**, and **getMultiSelectFileTitle** methods, which enable you to access the **fileName** property string when file names contain embedded spaces and the **allowMultiSelect** property is set to **true**.

fileTitle

Type: String

The **fileTitle** property of the **CMDFileSave** class contains the name without the path of the file to save in the common File Save dialog.

When the user selects a file and clicks the **OK** button in the dialog, the **fileTitle** property contains a value that can then be used save the selected file.

Note If the value of the **validate** property is set to **false**, the **fileTitle** property does not return a value. If the **allowMultiSelect** property is set to **true** and more than one file is selected, the **fileTitle** property is not set.

filter

Type: String

Default Value: "All files|*.*"

The **filter** property of the **CMDFileSave** class contains the filters that are displayed in the **Type** list box of the common File Save dialog. Use this property to provide the user with a list of filters that can be selected when the dialog is displayed.

A filter specifies the type of files that are displayed in the **File** list box of the dialog; for example, select the ***.txt** filter to display all text files. Each type consists of two parts, as follows.

- A description that is displayed to the user
- The actual file type; for example, ***.txt** for text files

Use the pipe (**|**) symbol to separate each of the description and filter arguments. Do not include spaces before or after the pipe symbol, as these spaces are then displayed with the description and filter values. The following text shows an example of a filter that enables the user to choose text files or picture files that include bitmaps and icons.

```
Text (*.txt)|*.txt|Pictures(*.bmp;*.ico)|*.bmp;*.ico
```

Use the **filterIndex** property to determine which filter is displayed as the default when you specify more than one filter for a dialog.

filterIndex

Type: Integer

Default Value: 1

The **filterIndex** property of the **CMDFileSave** class contains the filter listed in the **filter** property that is the default for the common File Save dialog.

hideReadOnly

Type: Boolean

Default Value: False

Set the **hideReadOnly** property of the **CMDFileSave** class to specify whether the **Read Only** check box is hidden in the common File Save dialog.

Set this property to **true** if you want to hide the display of the **Read-Only** check box.

initDir

Type: String

Default Value: Current directory

The **initDir** property of the **CMDFileSave** class contains the initial file directory for the common File Save dialog.

noReadOnlyReturn

Type: Boolean

Default Value: False

Set the **noReadOnlyReturn** property of the **CMDFileSave** class to **true** to specify that the file returned from the common File Save dialog cannot have the **Read Only** attribute set and cannot be in a write-protected directory.

overwritePrompt

Type: Boolean

Default Value: True

Set the **overwritePrompt** property of the **CMDFileSave** class to **true** to specify that the common File Save dialog generates a message box if the selected file already exists.

The user must confirm whether to overwrite the existing file.

pathMustExist

Type: Boolean

Default Value: True

Set the **pathMustExist** property of the **CMDFileSave** class to **false** to specify that the user can enter path names that do not exist (that is, which are currently invalid) in the common File Save dialog.

When this property is set to the default value of **true** and the user enters an invalid path name, a warning message is displayed.

readOnly

Type: Boolean

Default Value: False

Set the **readOnly** property of the **CMDFileSave** class to **true** to specify that the **Read Only** check box in the common File Save dialog is initially checked when the dialog box is created.

This property also indicates the state of the **Read Only** check box when the dialog box is closed.

resetCurrentPath

Type: Boolean

Default Value: False

Set the **resetCurrentPath** property of the **CMDFileSave** class to **true** to specify that the current path in the common File Save dialog is reset back to its value when the dialog was initiated.

Setting the property to **false** indicates that the current path is set to the last path selected by the user of the common dialog.

shareAware

Type: Boolean

Default Value: False

Set the **shareAware** property of the **CMDFileSave** class to **true** to specify that network sharing violation errors are ignored in the common File Save dialog.

validate

Type: Boolean

Default Value: True

Set the **validate** property of the **CMDFileSave** class to **false** to specify that the common dialog allows invalid characters in the file name returned from the common File Save dialog.

CMDFileSave Methods

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

Method	Description
getMultiSelectCount	Returns the number of files selected when the **allowMultiSelect** property is set to **true**
getMultiSelectDirectory	Returns the directory of the files selected when the **allowMultiSelect** property is set to **true**
getMultiSelectFileTitle	Returns the title of the specified file when the **allowMultiSelect** property is set to **true**
open	Initiates the common File Save dialog for the **CMDFileSave** class

getMultiSelectCount

Signature `getMultiSelectCount(): Integer;`

The **getMultiSelectCount** method of the **CMDFileSave** class returns the number of files selected in the **File Name** list box when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false**, this method returns a **null** value.

getMultiSelectDirectory

Signature `getMultiSelectDirectory(): String;`

The **getMultiSelectDirectory** method of the **CMDFileSave** class returns the name of the directory in which the files selected in the **File Name** list box are located when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false**, this method returns a **null** value.

getMultiSelectFileTitle

Signature `getMultiSelectFileTitle(indx: Integer): String;`

The **getMultiSelectFileTitle** method of the **CMDFileSave** class returns the title of the file specified in the **indx** parameter (that is, the name of the file without the path) when the **allowMultiSelect** property is set to **true**.

If the **allowMultiSelect** property is set to **false** or the value specified in the **indx** parameter is invalid, this method returns a **null** value.

open

Signature `open(): Integer updating;`

The **open** method of the **CMDFileSave** class initiates the common File Save dialog for the **CMDFileSave** class and returns a value indicating the success of the user actions, as listed in the following table.

Value	Description
0	User clicked the **OK** button. Values of the selections made are returned.
1	User clicked the **Cancel** button.
Other	Windows error number indicating a fault associated with the execution of the dialog.

The following example, which uses the **CMDFileSave** class:

1. Displays a common File Save dialog.
2. Obtains the file selected by the user.
3. Writes details of products to the file.

```
vars
    file  : File;
    myDlg : CMDFileSave;
    prod  : Product;
begin
    create myDlg;
    if myDlg.open = 0 then                      // user clicked OK
        create file;
        file.fileName    := myDlg.fileName;     // get selected name
        file.mode        := File.Mode_Append;
        file.allowCreate := true;               // create if not there
        file.open;                              // not mandatory
        foreach prod in company.allProducts do
            file.writeLine(prod.code & ":" & prod.description);
        endforeach;
        file.close;
        delete file;
    endif;
epilog
    delete myDlg;
end;
```

Notes Any values that are returned are retained if further calls are made on the class instance. The object should be deleted when it is no longer required.

An exception is raised if this method is invoked from a server method.

The position of the displayed File Save dialog cannot be determined. The values of the variable options for the class should be set before the **open** method is used.

CMDFont Class

The **CMDFont** class enables access to the common Font dialog facility for the selection of fonts.

Notes This class is not available on a server node.

In JADE thin client mode, the common Font dialog executes on the presentation client and returns information relative to the presentation client.

For details about the properties and method defined in the **CMDFont** class, see "CMDFont Properties" and "CMDFont Method", in the following subsections.

Inherits From: CMDialog

Inherited By: (None)

CMDFont Properties

The properties defined in the **CMDFont** class are summarized in the following table. These properties must be set before the **open** method that invokes the dialog is called.

Property	Description
ansiOnly	Specifies whether the dialog allows only fonts that use the Windows character set
fixedPitchOnly	Specifies whether the common Font dialog displays only fixed-pitch fonts
fontBold	Specifies whether the **bold** attribute is initially selected in the common Font dialog
fontItalic	Specifies whether the **italic** attribute is initially selected in the common Font dialog
fontName	Contains the font initially selected in the common Font dialog
fontSize	Contains the font size initially selected in the common Font dialog
fontStrikethru	Specifies whether the **Strikethrough** attribute is initially selected in the common Font dialog
fontUnderline	Specifies whether the **Underline** attribute is initially selected in the common Font dialog
forceFontExist	Specifies whether the dialog displays an error message box if the user selects a font or style that does not exist
maxSize	Contains the largest font size displayed in the common Font dialog **Size** list box
minSize	Contains the smallest font size displayed in the common Font dialog **Size** list box
noNameSelection	Specifies whether the **Font** combo box in the common Font dialog initially displays a font name
noSizeSelection	Specifies whether the **Size** combo box in the common Font dialog initially displays a font size
noStyleSelection	Specifies whether the **Font style** combo box in the common Font dialog initially displays a font style
printerDC	Contains the 32-bit Windows device context of the printer for use with the dialog
printerDC64	Contains the 64-bit Windows device context of the printer for use with the dialog
printerFonts	Specifies whether the dialog lists only the fonts supported by the printer

Property	Description
scalableOnly	Specifies whether the dialog lists only the fonts that can be scaled
screenFonts	Specifies whether the dialog lists only the screen fonts supported by the system
showEffects	Specifies whether the dialog enables strikethrough, underline, and color effects
simulations	Specifies whether the dialog allows Graphic Device Interface (GDI) font simulations
textColor	Contains the color of the text displayed in the dialog Effects group box
trueTypeOnly	Specifies whether the dialog displays only true-type fonts for user selection
vectorFonts	Specifies whether the dialog includes vector-type fonts in the fonts list
wysiwyg	Specifies whether only fonts that are available both for display and on the printer can be selected

ansiOnly

Type: Boolean

Default Value: False

The **ansiOnly** property of the **CMDFont** class specifies whether the common Font dialog allows selection only of the fonts that use the Windows character set.

If this property is set to **true**, the user cannot select a font that contains only symbols.

fixedPitchOnly

Type: Boolean

Default Value: False

The **fixedPitchOnly** property of the **CMDFont** class specifies whether the common Font dialog displays only fixed-pitch fonts.

fontBold

Type: Boolean

Default Value: False

The **fontBold** property of the **CMDFont** class specifies whether the **bold** attribute is initially selected in the common Font dialog.

When the user clicks the **OK** button, the selected value is returned.

fontItalic

Type: Boolean

Default Value: False

The **fontItalic** property of the **CMDFont** class specifies whether the **italic** attribute is initially selected in the common Font dialog.

When the user clicks the **OK** button, the selected value is returned.

fontName

Type: String[31]

Default Value: Null

The **fontName** property of the **CMDFont** class contains the font initially selected in the common Font dialog.

When the user clicks the **OK** button, the selected value is returned.

fontSize

Type: Real

Default Value: 0

The **fontSize** property of the **CMDFont** class contains the font size initially selected in the common Font dialog. When the user clicks the **OK** button, the selected value is returned.

fontStrikethru

Type: Boolean

Default Value: False

The **fontStrikethru** property of the **CMDFont** class specifies whether the **Strikethrough** attribute is initially selected in the common Font dialog. When the user clicks the **OK** button, the selected value is returned.

Note To make the **fontStrikethru** property visible in the dialog, the **showEffects** property must also be set to **true**.

fontUnderline

Type: Boolean

Default Value: False

The **fontUnderline** property of the **CMDFont** class specifies whether the **Underline** attribute is initially selected in the common Font dialog. When the user clicks the **OK** button, the selected value is returned.

Note To make the **fontUnderline** property visible in the dialog, the **showEffects** property must also be set to **true**.

forceFontExist

Type: Boolean

Default Value: True

The **forceFontExist** property of the **CMDFont** class specifies whether the common Font dialog displays an error message box if the user attempts to select a font or style that does not exist.

maxSize

Type: Integer

Default Value: 0 (no restriction)

The **maxSize** property of the **CMDFont** class contains the largest font size displayed in the common Font dialog **Size** list box.

minSize

Type: Integer

Default Value: 0 (no restriction)

The **minSize** property of the **CMDFont** class contains the smallest font size displayed in the common Font dialog **Size** list box.

noNameSelection

Type: Boolean

Default Value: False

The **noNameSelection** property of the **CMDFont** class specifies whether the **Font** combo box in the common Font dialog initially displays a font name. By default, a font name that applies to all selected text is displayed.

Set this property to **true** if you want to prevent the common Font dialog from displaying an initial selection (for example, when there is no single font name that applies to the text selection.)

When the user selects a font name in the **Font** combo box, the value of the **noNameSelection** property is set to **false**.

noSizeSelection

Type: Boolean

Default Value: False

The **noSizeSelection** property of the **CMDFont** class specifies whether the **Size** combo box in the common Font dialog initially displays a font size. By default, a font size that applies to all selected text is displayed.

Set this property to **true** if you want to prevent the common Font dialog from displaying an initial selection (for example, when there is no single font size that applies to the text selection.)

When the user selects a font size in the **Size** combo box, the value of the **noSizeSelection** property is set to **false**.

noStyleSelection

Type: Boolean

Default Value: False

The **noStyleSelection** property of the **CMDFont** class specifies whether the **Font style** combo box in the common Font dialog initially displays a font style (for example, the italics or bold attribute). By default, a font style that applies to all selected text is displayed.

Set this property to **true** if you want to prevent the common Font dialog from displaying an initial selection (for example, when there is no single font style that applies to the text selection.)

When the user selects a font style in the **Font style** combo box, the value of the **noStyleSelection** property is set to **false**.

printerDC

Type: Integer

Availability: Write only

The **printerDC** property of the **CMDFont** class contains the 32-bit Windows device context of the printer for use with the common Font dialog. When selecting printer fonts, the **printerDC** property must be set to the device context of the appropriate printer before the common Font dialog is initiated.

printerDC64

Type: Integer64

Availability: Write only

The **printerDC64** property of the **CMDFont** class contains the 64-bit Windows device context of the printer for use with the common Font dialog. When selecting printer fonts, the **printerDC64** property must be set to the device context of the appropriate printer before the common Font dialog is initiated.

If this property is set, the **CMDFont** class **printerDC** property is ignored.

Note A value set in the **CMDFont** class **printerDC** property is truncated to a 32-bit integer, and may not function as required in a 64-bit environment.

printerFonts

Type: Boolean

Default Value: False

The **printerFonts** property of the **CMDFont** class specifies whether the common Font dialog lists only the fonts supported by the printer specified by the **printerDC** property.

You should set either the **screenFonts** or the **printerFonts** property. If you set neither property, the **screenFonts** property is assumed.

scalableOnly

Type: Boolean

Default Value: False

The **scalableOnly** property of the **CMDFont** class specifies whether the common Font dialog lists only the fonts that can be scaled.

screenFonts

Type: Boolean

Default Value: True

The **screenFonts** property of the **CMDFont** class specifies whether the common Font dialog lists only the screen fonts supported by the system.

showEffects

Type: Boolean

Default Value: True

The **showEffects** property of the **CMDFont** class specifies whether the common Font dialog enables strikethrough, underline, and color effects.

simulations

Type: Boolean

Default Value: True

The **simulations** property of the **CMDFont** class specifies whether the common Font dialog allows Graphic Device Interface (GDI) font simulations.

textColor

Type: Integer

Default Value: Black

The **textColor** property of the **CMDFont** class contains the color of the text displayed in the common Fonts dialog Effects group box. When the user clicks the **OK** button, the selected color value is returned. For the Effects group box to be visible, the **showEffects** property must also be set.

trueTypeOnly

Type: Boolean

Default Value: False

The **trueTypeOnly** property of the **CMDFont** class specifies whether the common Font dialog displays only true-type fonts for the user to select.

vectorFonts

Type: Boolean

Default Value: True

The **vectorFonts** property of the **CMDFont** class specifies whether the common Font dialog includes vector-type fonts in the fonts list.

wysiwyg

Type: Boolean

Default Value: False

The **wysiwyg** ("what you see is what you get") property of the **CMDFont** class specifies whether the common Fonts dialog allows only the selection of fonts that are available both for display and on the printer.

If you set this property, the **printerFonts**, **screenFonts**, and **scalableOnly** properties apply.

CMDFont Method

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

Method	Description
open	Initiates the common Fonts dialog

open

Signature `open(): Integer updating;`

The **open** method of the **CMDFont** class initiates the common Fonts dialog and returns a value indicating the success of the user actions, as listed in the following table.

Value	Description
0	User clicked the **OK** button. Values of the selections made are returned.
1	User clicked the **Cancel** button.
Other	Windows error number indicating a fault associated with the execution of the dialog.

The position of the displayed Fonts dialog cannot be determined. The values of the variable options for the class should be set before the **open** method is used.

Notes Any values that are returned are retained if further calls are made on the class instance. The object should be deleted when it is no longer required.

An exception is raised if this method is invoked from a server method.

CMDPrint Class

The **CMDPrint** class enables access to the common Print dialog facilities for the selection of the print environment. Common print dialogs, controlled by the **printSetup** property, are as follows.

- The Print Setup dialog (the default) enables a user to select the required printer, number of copies, paper size, pagination, source, orientation, and so on.
- The Print dialog enables the user to specify the properties of a specific print task (that is, the number of copies and the page range).

When the user clicks the **OK** button in either dialog, the printer, copies, paper size, and page orientation properties are established for the application printer and apply until the application is terminated or the application printer properties are directly modified.

Other print dialog properties (for example, page range and print to file options) are the responsibility of the caller of the common dialog. No automatic actions are taken with these values.

Notes This class is not available on a server node.

In JADE thin client mode, the common Print dialog executes on the presentation client and returns information relative to the presentation client.

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

Inherits From: CMDialog

Inherited By: (None)

CMDPrint Class Constants

The constants defined in the **CMDPrint** class are summarized in the following table.

Constant	Integer Value
InitializeWith_DefaultPrinter	0
InitializeWith_MostRecentSetup	1

CMDPrint Properties

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

Property	Description
allPagesStatus	Specifies the status of the **All Pages** check box
collateStatus	Specifies the status of the common Print dialog **Collate** check box
copies	Contains the number of copies to be printed for the common Print dialog
disablePageNumbers	Specifies whether the page number selection options are disabled
disablePrintToFile	Specifies whether the **Print to file** check box is disabled
disableSelection	Specifies whether the **Selection** option is disabled

Property	Description
documentType	Contains the printer form type
duplex	Contains the duplex setting of the print output
fromPage	Contains the *from* page that is to be printed from the common Print dialog
hidePrintToFile	Specifies whether the **Print to file** check box is hidden
initializeWith	Specifies whether a common Print Setup dialog is initialized with the default printer settings or values set on a previous Print Setup dialog
maxPage	Contains the maximum range for the **fromPage** and **toPage** properties
minPage	Contains the minimum range for the **fromPage** and **toPage** properties
orientation	Contains the orientation of the print output
pageNumbersStatus	Specifies the status of the **Page Range** check box
paperSource	Contains the paper source, or tray, of the print output
printSetup	Specifies whether the Printer Setup dialog is displayed instead of the common Print dialog
printToFileStatus	Specifies the status of the **Print to file** check box
printerDC	Contains a 32-bit device context of the selected printer
printerDC64	Contains a 64-bit device context of the selected printer
printerName	Contains the name of the selected printer
returnDC	Contains a device context returned from the dialog for the printer
selectionStatus	Specifies the status of the **Selection** check box
toPage	Contains the *to* page that is to be printed from the common Print dialog
warnIfNoDefault	Specifies whether a warning message box is displayed if there is no default

allPagesStatus

Type: Boolean

Default Value: True

The **allPagesStatus** property of the **CMDPrint** class specifies whether the **All Pages** check box in the common Print dialog is set. When the common Print dialog is displayed, the **All Pages** check box is set if the **allPagesStatus** property is set to **true** and the **printSetup** property is set to **false**.

collateStatus

Type: Boolean

Default Value: False

The **collateStatus** property of the **CMDPrint** class specifies whether the **Collate** check box in the common Print dialog is set, so that multiple copies are printed in the proper binding order.

You cannot modify this property after printing has begun.

Note This property applies only when the **printSetup** property is set to **false**, the value of the **copies** property is greater than **1** (the default), and the printer device supports the collation of multiple copies.

copies

Type: Integer

Default Value: 1

The **copies** property of the **CMDPrint** class contains the number of copies to be printed for the common Print dialog. This property applies only when the **printSetup** property is set to **false**. Set this value to display the number of copies that are to be shown in the common Print dialog.

Note Multiple copies are produced only if the printer device driver supports the printing of multiple copies.

disablePageNumbers

Type: Boolean

Default Value: False

Set the **disablePageNumbers** property of the **CMDPrint** class to **true** to disable the page number selection options in the common Print dialog.

This property applies only if the value of the **printSetup** property is **false**.

disablePrintToFile

Type: Boolean

Default Value: False

Set the **disablePrintToFile** property of the **CMDPrint** class to **true** to disable the common Print dialog **Print to file** check box.

This property applies only when the **printSetup** property is set to **false**.

disableSelection

Type: Boolean

Default Value: False

Set the **disableSelection** property of the **CMDPrint** class to **true** to disable the common Print dialog.

This property applies only when the **printSetup** property is set to **false**.

documentType

Type: Integer

Default Value: Value returned by **app**.**printer**.**getDefaultDocumentType**

The **documentType** property of the **CMDPrint** class contains the printer form type; that is, the size of the paper or envelope that you want to print. This property applies only when the **printSetup** property is set to **false** and it cannot be modified after printing has begun.

The document (printer form) types provided by the **Printer** global constant category 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
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

Global Constant	Integer Value	Description
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
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 following example shows the use of the **documentType** property.

```
buttonPrint_click(btn: Button input) updating;
vars
    cmdPrint : CMDPrint;
begin
    create cmdPrint;
    cmdPrint.documentType := Print_Letter;
    write cmdPrint.documentType;            // Outputs 1
    ...
epilog
    delete cmdPrint;                        // Tidy
end;
```

duplex

Type: Integer

Default Value: 1

The **duplex** property of the **CMDPrint** class contains the duplex value; that is, the number of sides on which the paper is printed and the way in which double-sided printing is performed.

You cannot modify this property after printing has begun.

Note This property applies only when the **printSetup** property is set to **false** and the printer device supports duplex printing.

The values of the **duplex** property are listed in the following table.

Integer Value	Description
1	Simple; that is, print is output to one side of the paper only (the default)
2	Vertical; that is, prints on both sides of the paper to read by turning like a book (that is, the duplex **Long Side** setting)
3	Horizontal; that is, prints on both sides of the paper to read by flipping over like a notepad (that is, the duplex **Short Side** setting)

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

```
create cmdPrint;
cmdPrint.duplex := 2;
delete cmdPrint;                            // tidy
```

fromPage

Type: Integer

Default Value: 0

The **fromPage** property of the **CMDPrint** class contains the *from* page that is to be printed from the common Print dialog. This property has no effect if the **maxPage** property is not also set. Set the value to display the starting page number in the common Print dialog.

This property applies only when the **printSetup** property is set to **false**.

hidePrintToFile

Type: Boolean

Default Value: False

Set the **hidePrintToFile** property of the **CMDPrint** class to **true** to hide the display of the common Print dialog **Print to file** check box. This property applies only when the **printSetup** property is set to **false**.

initializeWith

Type: Integer

Default Value: 0

Set the **initializeWith** property of the **CMDPrint** class to the class constant **InitializeWith_DefaultPrinter** (0) value to initialize the common Print Setup dialog with the default printer settings.

Set the value to the class constant **InitializeWith_MostRecentSetup** (1) to initialize the dialog with the values that were set when the dialog was most-recently opened in the JADE application.

Notes Specifying the **InitializeWith_MostRecentSetup** value without previously opening the common Print Setup dialog is equivalent to specifying the **InitializeWith_DefaultPrinter** value.

Printer setup values are saved only for the Print Setup dialog (not for the Print dialog); that is, the value of the **printSetup** property must be set to **true**.

---

maxPage

Type: Integer

Default Value: 0 (none)

The **maxPage** property of the **CMDPrint** class contains the maximum range for the **fromPage** and **toPage** properties that are set from the common Print dialog. Set the **maxPage** property to restrict the user to a specific page range during print selections. This property applies only when the **printSetup** property is set to **false**.

minPage

Type: Integer

Default Value: 0 (all)

The **minPage** property of the **CMDPrint** class contains the minimum range for the **fromPage** and **toPage** properties that are set from the common Print dialog.

Set the **minPage** property to restrict the user to a certain page range during print selections. This property applies only when the **printSetup** property is set to **false**.

orientation

Type: Integer

Default Value: Print_Portrait

The **orientation** property of the **CMDPrint** class contains the orientation of your printed output. This property applies only when the **printSetup** property is set to **false**. 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.

```
buttonPrint_click(btn: Button input) updating;
vars
    cmdPrint : CMDPrint;
begin
    create cmdPrint;
    cmdPrint.orientation  := Print_Landscape;
    write cmdPrint.orientation;             // Outputs 2
epilog
    delete cmdPrint;                        // Tidy
end;
```

pageNumbersStatus

Type: Boolean

Default Value: False

The **pageNumbersStatus** property of the **CMDPrint** class specifies whether the **Page Range** check box in the common Print dialog is set. When the common Print dialog is displayed, the **Page Range** check box is set if the **pageNumbersStatus** property is set to **true**.

This property applies only when the **printSetup** property is set to **false**.

paperSource

Type: Integer

Default Value: 0 (all)

The **paperSource** property of the **CMDPrint** class contains the location in the printer of the paper that you want to use when printing from the common Print dialog. You cannot modify this property after printing has begun.

This property applies only when the **printSetup** property is set to **false**.

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. A value of zero (**0**) indicates that all paper sources are displayed in the common Print dialog.)

Use the **Printer** class **getAllPaperSources** method to access the valid paper sources of a printer.

printSetup

Type: Boolean

Default Value: True

Set the **printSetup** property of the **CMDPrint** class to **false** if you want to initiate the common Print dialog when the **open** method is called, to specify the properties of a specific print task (that is, the number of copies and the page range).

By default, the common Print Setup dialog is initiated, which enables you to select the required printer, number of copies, paper size, pagination, source, orientation, and so on. These values are then applied by the next print request unless your logic overrides the printer properties.

printToFileStatus

Type: Boolean

Default Value: False

The **printToFileStatus** property of the **CMDPrint** class specifies whether the **Print to file** check box in the common Print dialog is set.

This property applies only when the **printSetup** property is set to **false**.

printerDC

Type: Integer

Availability: Read-only

The **printerDC** property of the **CMDPrint** class contains a 32-bit device context for the printer selected in the common Print dialog when the **returnDC** property is set to **1** (return DC) or **2** (return information DC).

Note The calling method is responsible for deleting the device context.

printerDC64

Type: Integer64

Availability: Read-only

The **printerDC64** property of the **CMDPrint** class contains a 64-bit device context for the printer selected in the common Print dialog when the **returnDC** property is set to **1** (return DC) or **2** (return information DC).

Note The calling method is responsible for deleting the device context.

The **printerDC** property also contains the printer device context handle if it is not larger than a 32-bit integer; otherwise the **printerDC** property is set to zero (**0**).

printerName

Type: String

The **printerName** property of the **CMDPrint** class contains the name of the printer selected in the common Print dialog.

returnDC

Type: Integer

The **returnDC** property of the **CMDPrint** class contains a device context that is returned for the printer from the common Print dialog.

The setting of the **returnDC** property can be one of the values listed in the following table.

Setting	Description
1	Returns a device context for the printer selected in the dialog. The device context is returned in the **printerDC** or the **printerDC64** property.
2	Returns an information context for the printer selected in the dialog. An information context provides a fast way to get information about the device without creating a device context. The information context is returned in the **printerDC** or the **printerDC64** property.
Neither 1 nor 2	Return value undefined.

It is the responsibility of the caller to use and delete the returned device context.

selectionStatus

Type: Boolean

Default Value: False

The **selectionStatus** property of the **CMDPrint** class specifies the status of the **Selection** check box in the common Print dialog.

When the common Print dialog is displayed, the **Selection** check box is set if the **selectionStatus** property is set to **true**.

This property applies only when the **printSetup** property is set to **false**.

toPage

Type: Integer

Default Value: 0

The **toPage** property of the **CMDPrint** class contains the *to* page that is to be printed from the common Print dialog. This property has no effect if the **maxPage** property is not also set.

Set the value to display the ending page number in the common Print dialog.

This property applies only when the **printSetup** property is set to **false**.

warnIfNoDefault

Type: Boolean

Default Value: True

Set the **warnIfNoDefault** property of the **CMDPrint** class to **false** to specify that a warning message box is not displayed if there is no printer default for the system on the common Print dialog.

By default, a warning message box is displayed if there is no printer default for the system on the common Print dialog.

CMDPrint Method

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

Method	Description
open	Initiates the common Print dialog for the **CMDPrint** class

open

Signature `open(): Integer updating;`

The **open** method of the **CMDPrint** class initiates the common Print dialog for the **CMDPrint** class and returns a value indicating the success of the user actions, as listed in the following table.

Value	Description
0	User clicked the **OK** button. Values of the selections made are returned.
1	User clicked the **Cancel** button.
Other	Windows error number indicating a fault associated with the execution of the dialog.

The position of the displayed Print dialog cannot be determined. The values of the variable options for the class should be set before the **open** method is used.

The following example shows the use of the **open** method to open a common Print dialog.

```
vars
    cmdPrint : CMDPrint;
begin
    create cmdPrint;
    if cmdPrint.open = 0 then              // not cancelled and no error
        str := cmdPrint.printerName;       // use returned value
    endif;
epilog
    delete cmdPrint;                       // tidy
end;
```

Notes Any values that are returned are retained if further calls are made on the class instance. The object should be deleted when it is no longer required.

An exception is raised if this method is invoked from a server method.

When a printer is about to be opened and the previously used printer name is no longer valid, the following log message is written and the current default printer is used instead.

```
Printer name is no longer valid: printer-name - selecting default printer
```

Note If the default printer is used, JADE does not retain the printer name between print jobs. If you change the default printer, the next print job to the default printer is output to a different (the new default) printer. If you have changed the default printer and you want to output the job to the earlier printer, specify the actual printer name.

Collection Class

Collections are the basic structures used to store multiple object references or primitive type values. The **Collection** class is the abstract superclass of all collection classes that defines the common protocol for all of its subclasses. The **Collection** class provides the protocol to:

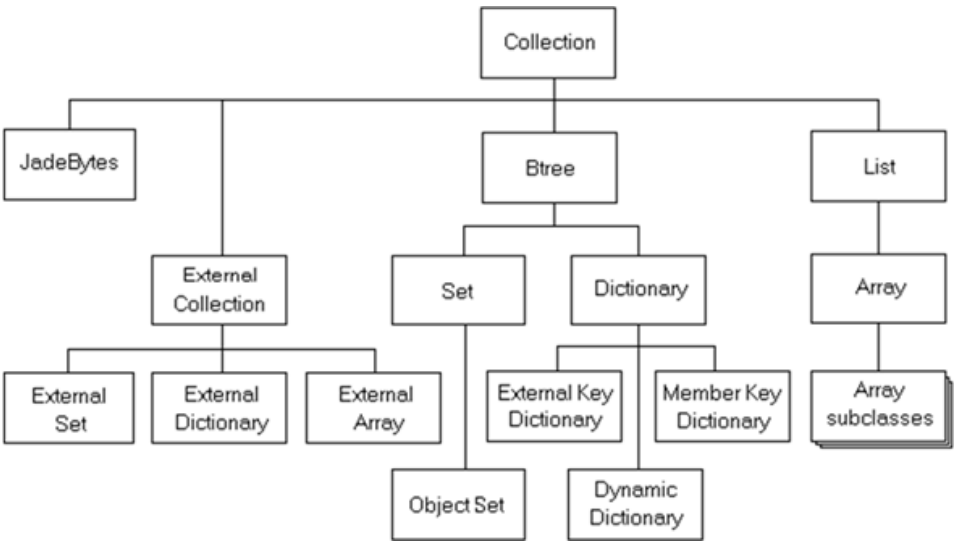
- Directly access or store a specific element in a collection
- Access all elements in a collection in a specific order

When membership of a collection is a class, instances of that class and all of its subclasses can be included in the collection. When a **Collection** object is cloned (by using the reimplemented **cloneSelf** method of the **Object** class), the entries in the collection are copied to the new collection instance.

Note The **add**, **remove**, and **includes** methods are defined at the **Collection** class level to provide closure and are inherited by all subclasses of collection. However, use of these **Collection** class methods with external key dictionaries is not recommended because none of the method signatures allow for the specification of external keys.

Transient objects, including exclusive collections, that are automatically created by JADE cannot be shared. However, all exclusive collections of a shared transient object are created as shared transient objects by JADE. (For details about specifying the creation of transient objects that can be shared across threads, see "*create* Instruction", in Chapter 1 of the *JADE Developer's Reference*.)

The structure of the **Collection** class hierarchy is shown in the following diagram.



The basic types of **Collection** classes are:

- Dictionaries
- Sets
- Arrays

Tip Use bracket symbols (**[]**) to access random entries in a dictionary or array.

An exclusive collection is created when it is updated for the first time. Whenever an exclusive collection is created, regardless of whether it has an inverse reference, the edition of the owner object is updated. An exclusive collection is also created if the instantiate method is called before the collection is updated the first time. For details about the behavior of and tuning collections, see Chapter 4 of the *JADE Developer's Reference*.

When the type of a property is a **Collection** subclass, the access mode setting applies to the reference to the collection rather than its members; that is, a setting of **read-only** does not prevent collection members being added, deleted, or updated.

Note For **Collection** classes that are internal pseudo arrays (that is, arrays of GUI-related information only in the JADE run time module), the only **Collection** class methods that are implemented are **size** and **size64**.

For details about specifying dictionary membership and keys at run time, see "DynaDictionary Class", later in this chapter. For details about the methods defined in the **Collection** class, see "Collection Methods", in the following subsection. (For details about the **lockReceiver** method option, which indicates that an exclusive lock of transaction duration is to be placed on the receiver collection before calling the method, see "lockReceiver Option", in Chapter 1 of the *JADE Developer's Reference*.)

Inherits From: Object

Inherited By: Btree, ExternalCollection, JadeBytes, List

Collection Methods

The methods summarized in the following table are implemented for (and inherited by) all **Collection** subclass instances.

Method	Description
add	Adds an object to a collection
clear	Removes all entries from a collection
copy	Copies entries from the receiver to a compatible collection
countOf	Returns the number of times the specified entry occurs in the collection
countOf64	Returns the number of times the specified entry occurs in the collection as an **Integer64** value
createIterator	Creates an iterator for the **Collection** class and subclasses
deleteIfEmpty	Deletes a shared or exclusive collection if it is empty
first	Returns the first entry in the collection
getOwner	Returns the object that is the owner of the collection
getStatistics	Analyzes the collection and returns structural statistics
includes	Returns **true** if the collection 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 as an **Integer64** value
indexOf	Returns the index of a specified entry if it exists in the collection
indexOf64	Returns the index of a specified entry if it exists in the collection as an **Integer64** value
inspect	Inspects a collection of objects

Method	Description
inspectModal	Opens a modal Inspector form for the receiver object
instantiate	In exclusive collections only, ensures that the object is created before it is used
isEmpty	Returns **true** if the collection has no entries
last	Returns the last entry in the collection
maxSize	Returns the maximum number of entries that the collection can contain
maxSize64	Returns the maximum number of entries that the collection can contain as an **Integer64** value
purge	Deletes all object references in a collection
rebuild	Removes invalid object references and fixes up dictionary keys in a collection
remove	Removes an item from a collection
setBlockSize	Specifies or changes the block size of the receiver
size	Returns the current number of entries in the collection
size64	Returns the current number of entries in the collection as an **Integer64** value

add

Signature `add(value: MemberType) updating, abstract;`

The **add** method of the **Collection** class adds the object specified in the **value** parameter to a collection.

clear

Signature `clear() updating, abstract;`

The **clear** method of the **Collection** class removes all entries (object references) from a collection.

The objects that are removed are not deleted; they are simply no longer in the collection.

Note If the collection is not instantiated, the **clear** method instantiates it. However, if the collection is frozen, exception 1106 (*Can not update a frozen object*) is raised.

copy

Signature `copy(toColl: Collection input);`

The **copy** method of the **Collection** 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. For example, the **copy** method can be used to copy entries from a dictionary of employees to an array of objects, as shown in the following examples.

```
create personArray transient;
dept.allEmployees.copy(personArray);

dept1.employees.copy(dept2.employees);
```

Note By default, entries copied from the receiver collection are *added* to entries that already exist in the collection to which you copy them.

countOf

Signature `countOf(value: MemberType): Integer;`

The **countOf** method of the **Collection** class returns the number of times the entry specified in the **value** parameter occurs in the collection.

Note Use the **countOf64** method instead of the **countOf** method, if the number of entries could exceed the maximum integer value of 2,147,483,647.

countOf64

Signature `countOf64(value: MemberType): Integer64;`

The **countOf64** method of the **Collection** class returns the number of times the entry specified in the **value** parameter occurs in the collection as an **Integer64** value.

createIterator

Signature `createIterator(): Iterator abstract;`

The **createIterator** method of the **Collection** class creates an iterator for the **Collection** class and its subclasses. Use an iterator associated with the collection to remember the current position in the collection. (For details about iterators, see the **Iterator** class.)

The following example shows the use of the iterator.

```
vars
    iter : Iterator;
begin
    iter := company.allEmployees.createIterator;
    while iter.next(emp) do
        ...
    endwhile;
epilog
    delete iter;
end;
```

deleteIfEmpty

Signature `deleteIfEmpty();`

The **deleteIfEmpty** method of the **Collection** class, which can be executed only in single user mode, deletes an empty shared or exclusive collection. If the method is called in multiuser mode, an exception 131 (*This method can only be executed in single user*) is raised.

If the collection is not empty, calling this method raises exception 1320 (*Operation invalid - collection not empty*). Call the **isEmpty** method to determine if a collection is empty.

first

Signature `first(): MemberType abstract;`

The **first** method of the **Collection** class returns a reference to the first entry in a collection.

getOwner

Signature `getOwner(): Object;`

The **getOwner** method of the **Collection** class returns a reference to the object that is the owner, or parent, of the collection. This method returns **null** if the object is not an exclusive collection.

getStatistics

Signature `getStatistics(statistics: JadeDynamicObject input) abstract;`

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 (oid (**6**) for **Set** classes and Integer (**4**) for **Array** classes)
entrySize	Size of each collection entry in bytes
size	Number of entries in the collection (that is, the size of the collection itself)
blockCount	Total number of blocks in the collection
height	Number of levels in the collection (always **1** for **Array** classes)
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 and tuning of collections, see Chapter 4 of the *JADE Developer's Reference*.

includes

Signature `includes(value: MemberType): Boolean abstract;`

The **includes** method of the **Collection** class returns **true** if the collection contains the object specified in the **value** parameter. This method returns **false** if a null reference is passed to the **value** parameter. For a dictionary the method returns **true** if the object is contained in the dictionary at its current key value.

indexNear

Signature `indexNear(value: MemberType): Integer;`

The **indexNear** method of the **Collection** 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 For the **Set** and **MemberKeyDictionary** subclasses, this method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf** method. For other subclasses of the **Collection** class, the **indexNear** method is the same as the **indexOf** method, and is included for compatibility.

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 **Collection** 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 For the **Set** and **MemberKeyDictionary** subclasses, this method calculates and returns an approximate index. This incurs less processing overhead than using the **indexOf64** method. For other subclasses of the **Collection** class, the **indexNear64** method is the same as the **indexOf64** method, and is included for compatibility.

indexOf

Signature `indexOf(value: MemberType): Integer abstract;`

The **indexOf** method of the **Collection** class returns the index of the entry specified in the **value** parameter if it exists in the collection or it returns zero (**0**) if it does not exist.

If the specified value occurs more than once in the collection, the index of the first occurrence is returned.

Note Use the **indexOf64** method instead of the **indexOf** method, if the number of entries in the collection could exceed the maximum integer value of 2,147,483,647.

indexOf64

Signature `indexOf64(value: MemberType): Integer64 abstract;`

The **indexOf64** method of the **Collection** class returns the index of 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.

If the specified value occurs more than once in the collection, the index of the first occurrence is returned.

inspect

Signature `inspect();`

The **inspect** method of the **Collection** class enables you to inspect a collection object. An Inspector form for the selected collection is then opened.

An exception is raised if this method is invoked from a server method.

inspectModal

Signature `inspectModal();`

The **inspectModal** method of the **Collection** class opens a modal Inspector form for the receiver object. The inspector enables you to view properties of a collection. An exception is raised if this method is invoked from a server method.

instantiate

Signature `instantiate();`

The **instantiate** method of the **Collection** class ensures that the object is created before it is used.

Note This method applies only to exclusive collections.

isEmpty

Signature `isEmpty(): Boolean;`

The **isEmpty** method of the **Collection** class returns **true** if the collection does not contain any entries.

last

Signature `last(): MemberType abstract;`

The **last** method of the **Collection** class returns a reference to the last entry in a collection.

maxSize

Signature `maxSize(): Integer;`

The **maxSize** method of the **Collection** class returns the maximum number of entries that a collection can contain.

Note Use the **maxSize64** method instead of the **maxSize** method as the number of entries in the collection exceeds the maximum integer value of 2,147,483,647.

maxSize64

Signature `maxSize64(): Integer64;`

The **maxSize64** method of the **Collection** class returns the maximum number of entries that a collection can contain as an **Integer64** value.

purge

Signature `purge() updating, abstract;`

The **purge** method of the **Collection** class deletes all objects in a collection and clears the collection; that is, **size = 0**.

Caution The objects that are removed are physically deleted.

The purge operation ignores *object not found* exceptions, which enables you to fix manually maintained collections that have references to objects that are now deleted.

The following example shows the use of the **purge** method.

```
buttonUnload_click(btn: Button input) updating;
begin
    // Deletes the data and unloads the form.
    statusLine1.caption := "Deleting data...";
```

```
        statusLine1.refreshNow;
        beginTransaction;
            if self.numbers <> null then
                NumberDict.instances.purge;
                Number.instances.purge;
            endif;
        commitTransaction;
        self.unloadForm;
    end;
```

rebuild

Signature `rebuild() updating, abstract;`

The **rebuild** method of the **Collection** class restores the structural integrity of a collection, removes invalid object references, and fixes up dictionary keys in the receiving collection.

The **rebuild** method records information in the **jommsg.log** file about entries that have been corrected and if the collection size changes as a result of the rebuild action.

This method iterates the collection and performs the following actions.

- Sets, arrays, and external key dictionaries
  - Restores structural integrity of the collection.
  - Removes references to non-existent objects or references that are not type-compatible with the membership of the collection.
- Member key dictionaries and dynamic dictionaries (**DynaDictionary** instances)
  - Restores structural integrity of the collection.
  - Removes references to non-existent objects or references that are not type-compatible with the membership, and checks that dictionary keys match the member keys. When they do not, the entry is removed and reinserted with the correct keys.
- Arrays of primitive types
  - Restores structural integrity of the collection.

The **rebuild** method must be enclosed in a transaction to repair persistent collections.

remove

Signature `remove(value: MemberType) updating, abstract;`

The **remove** method of the **Collection** class removes an item from a collection. If the collection does not contain the specified item, an exception is raised.

setBlockSize

Signature `setBlockSize(blockSize: Integer) updating;`

The **setBlockSize** method of the **Collection** class enables you to specify or change the block size of the receiver in terms of entries in each block. When you use this method to change the collection block size, all collection blocks for the receiver are created with the specified size. For details about the behavior of and tuning collections, see Chapter 4 of the *JADE Developer's Reference*.

A physical upper limit is enforced. (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).

If the value of entries per block multiplied by the collection entry size causes the actual block size to exceed the JADE limit, an exception is raised.

If this method is invoked on a populated collection and the block size differs from that already in use by the receiver, an automatic upgrade is triggered, which restructures the collection to use the new size.

Notes The time taken to reblock a collection increases with the collection size and could be quite lengthy for large collections. This reblock operation is similar to the type of upgrade that can occur during a reorganization, and the collection remains inaccessible until the process has completed.

To adjust the block size at the class level, use the **Entries Per Block** text box on the **Tuning** sheet of the Define Class dialog.

size

Signature `size(): Integer;`

The **size** method of the **Collection** class returns the number of entries in a collection.

Note Use the **size64** method instead of the **size** method, if the number of entries in the collection could exceed the maximum integer value of 2,147,483,647.

size64

Signature `size64(): Integer64;`

The **size64** method of the **Collection** class returns the number of entries in a collection as an **Integer64** value.

Connection Class

The **Connection** class is an abstract class that encapsulates the behavior required for communicating with external systems and external applications (either JADE or non-JADE systems).

In JADE thin client mode, all connections are made to the workstation that is running the JADE logic; that is, to the application server.

The **Connection** 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 successfully, 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 the same 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.

Performing a synchronous write operation stops any additional requests from being queued until the synchronous operation is completed.

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

Inherits From: Object

Inherited By: JadeSerialPort, NamedPipe, TcpIpConnection

Connection Class Constants

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

Constant	Integer Value	Constant	Integer Value
Connected	2	Connecting	1
Disconnected	0	Disconnecting	3

Connection Properties

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

Property	Description
fillReadBuffer	Determines when the receiver object is notified
name	Contains the generic name of the connection to which the object is to connect
state	Contains the state of the connection
timeout	Contains the timeout value in milliseconds for listen, read, and write operations

fillReadBuffer

Type: Boolean

Set the **fillReadBuffer** property of the **Connection** class to **true** to specify that the **readBinary** and **readBinaryAsynch** methods do not return or notify the receiver object until the requested length of the data has been received.

If the **fillReadBuffer** property is set to **false** (the default), the **readBinary** and **readBinaryAsynch** methods return or notify the receiver object as soon as any data is received for the connection. The **length** parameter of the **readBinary** or **readBinaryAsynch** method is therefore treated as a maximum buffer size.

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

```
multiReceive();
vars
    bin    : Binary;
    count  : Integer;
    len    : String;
    number : String;
    tcp    : TcpIpConnection;
begin
    // Loop around receiving multiple inputs
    count := 0;
    while true do
        tcp.fillReadBuffer := true;
        bin                := tcp.readBinary(10);
        len                := bin [ 1 : 5 ].String;
        number             := bin [ 6 : 5 ].String;
        bin                := tcp.readBinary(len.Integer);
        sl1.caption        := number & ' ' & bin.String;
        sl1.refreshNow;
        count              := count + 1;
        if number.Integer <= 1 then
            sl1.caption := count.String & $X_Received;
            break;
        endif;
    endwhile;
end;
```

name

Type: String[128]

The **name** property of the **Connection** class contains the generic name of the connection (remote device) to which the object is to connect. Each subclass of the **Connection** class can interpret the **name** property as required. If subclasses of the **Connection** class require additional information to establish a connection, you may need to define additional properties (for example, the TCP/IP connection may require a valid port number).

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

```
// Creates a TCP/IP connection, sets the name to the current computer
// name, and sets the listen port
create tcp;
tcp.name := app.computerName;
tcp.port := 7015;
```

state

Type: Integer

Availability: Read-only at any time

The **state** property of the **Connection** class contains the state of the connection. Methods can be called only in the appropriate state and they can cause the connection state to change.

The values of the **state** property are listed in the following table.

Connection Class Constant	Integer Value
Connected	2
Connecting (or listening)	1
Disconnected	0
Disconnecting	3

The **open**, **openAsynch**, **listen**, **listenAsynch**, **listenContinuous**, and **listenContinuousAsynch** methods can be called only when the **state** property is set to **Disconnected** (**0**). When these methods are called, the state is changed to **Connecting** (**1**). The connection state changes to **Connected** (**2**) when the connection is open or a **listen** method completes.

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 **getMaxMessageSize**, **readBinary**, **readBinaryAsynch**, **writeBinary**, and **writeBinaryAsynch** methods can be called only when the state of the connection is connected; that is, this property is set to **Connected** (**2**).

The **close** and **closeAsynch** methods can be called when the connection is in any state.

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

```
// 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.
tcp.listen();
if tcp.state = Connection.Connected then
    statusLine1.caption  := "Connected";
    textBoxLocalIP.text  := tcp.localIpAddress;
    textBoxRemoteIP.text := tcp.remoteIpAddress;
    textBoxName.text     := tcp.name;
endif;
```

timeout

Type: Integer

The **timeout** property of the **Connection** class contains the number of milliseconds after which a synchronous or asynchronous listen (including continuous), read, or write operation times out.

The timeout value remains active for these operations until you reset the value in your application for that transient instance of the connection object.

The default value of zero (**0**) indicates that the operation does not time out.

The functionality of the **timeout** property is not supported in the **NamedPipe** subclasses of the **Connection** class.

Connection Methods

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

Method	Description
close	Closes a connection to a remote application and then returns
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
getNextSessionId	Returns a string of the encrypted version of the Web session identifier
listen	Listens for an external application to connect to JADE and returns when a connection is established
listenAsynch	Listens for an external application to connect to JADE
listenContinuous	Waits for an external application to connect to its port and returns the new connection on a new instance of the **Connection** 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 established
openAsynch	Establishes a connection to a remote application and returns immediately
openPipeCallback	Initiates an asynchronous read of the opened pipe
readBinary	Reads binary data from the connection and returns when the data has been read or when a block of data is received
readBinaryAsynch	Reads binary data from the connection and returns immediately
readPipeCallback	Performs Web session evaluation processing
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
sendReply	Sends the formatted HyperText Markup Language (HTML) page to the opened pipe
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 **Connection** class closes a connection to a remote application or device 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 it has been left in connection state.

```
buttonUnload_click(btn: Button input) updating;
begin
    // If a connection is present, closes the connection
    if self.connection.state = Connection.Connected then
        self.connection.close;
    endif;
    self.unloadForm;
end;
```

closeAsynch

Signature `closeAsynch(receiver: Object; msg: String);`

The **closeAsynch** method of the **Connection** 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.

When the **closeAsynch** method completes, the user-written callback method specified in the **msg** parameter is called.

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.

The callback method must match the signature required by the calling **closeAsynch** method, as follows.

Signature `closeCallback(connection: Connection);`

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
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // 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 updateCloseCalls method.
    self.connection.closeAsynch(conlog, "updateCloseCalls");
```

```
    statusLine1.caption := "Disconnected";
end;
```

getMaxMessageSize

Signature `getMaxMessageSize(): Integer;`

The **getMaxMessageSize** method of the **Connection** class returns the maximum message size that can be sent or received at one time.

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.

The **getMaxMessageSize** method can be called only when the **state** is **Connected** (2).

getNextSessionId

Signature `getNextSessionId(sessionEncrypt: String output): Integer;`

The **getNextSessionId** method of the **Connection** class returns the next Web session identifier and returns a string of the encrypted version of that identifier in the **sessionEncrypt** parameter.

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 **Connection** class listens for an external application to connect to JADE and returns when a connection is established. The **listen** method can be called only when the **state** property is **Disconnected** (**0**).

Each subclass of the **Connection** class must provide any properties that are required to define the connection; for example, the **name**, **port**, and so on. The **state** property changes to **Connected** (2) when the **listen** method completes.

The code fragment in the following example sets the connection to listen to the current port, sets the status bar to read *connected*, and fills the text box with the name information if a connection is made.

```
self.connection.listen;
if self.connection.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox.text        := self.connection.name;
endif;
...
```

See also the **Connection** class **timeout** property.

listenAsynch

Signature
```
listenAsynch(receiver: Object;
             msg:      String);
```

The **listenAsynch** method of the **Connection** class listens for an external application to connect to JADE.

When a connection is established, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter.

The **listenAsynch** method can be called only when the **state** is **Disconnected** (0). When this method is called, the state is changed to **Connecting** (1), or listening.

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 shows the use of the **listenAsynch** method.

```
listenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, the object is created and its properties
    // are initialized.
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // Sets the connection to listen on the current port and returns
    // immediately.  If a connection is made, the ConnectionLog object
    // referenced by conlog is called, to run the updateListenCalls method.
    self.connection.listenAsynch(conlog, "updateListenCalls");
end;
```

See also the **Connection** class **timeout** property.

When a connection is established, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **listenAsynch** method, as follows.

Signature `listenCallback(connection: Connection);`

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(connection: Connection) updating;
begin
    beginTransaction;
    self.numberOfListenCalls := self.numberOfListenCalls + 1;
    commitTransaction;
end;
```

listenContinuous

Signature `listenContinuous(): Connection;`

The **listenContinuous** method of the **Connection** 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 **Connection** class while the original instance is still available for listening on subsequent calls.

The **Connection** class **state** property changes to **Connecting** (1) when listening is in progress. The newly created instance of the **Connection** class has its **state** property set to **Connected** (2) after the successful connection.

See also the **Connection** class **timeout** property.

The following example shows the use of the **listenContinuous** method.

```
listenContinuous_click(btn: Button input) updating;
begin
    // Sets the connection to listen on the current port.  When a connection
    // is made, a new instance of the Connection class is returned and
    // referenced.  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 box filled with the name information.
    self.connection := connection.listenContinuous;
    if self.connection.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox.text        := connection2.name;
    endif;
end;
```

listenContinuousAsynch

Signature
```
listenContinuousAsynch(receiver: Object;
                       msg:      String);
```

The **listenContinuousAsynch** method of the **Connection** 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 name of the callback method specified in the **msg** parameter.

The **listenContinuousAsynch** method can be called only when the **Connection** class **state** is **Disconnected** (0). When this method is called, the value of the **state** property is changed 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 shows the use of the **listenContinuousAsynch** method.

```
listenContAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, it is created and its properties initialized.
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
```

```
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // Sets the connection to listen on the current port.  A new instance
    // of Connection is created when a connection is made.  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 Connection instance is passed to this method as a parameter.
    self.connection.listenContinuousAsynch(conlog,
                                           "updateListenContinuousCalls");
    if self.connection.state = Connection.Connected then
        statusLine1.caption := "Connected";
        textBox.text        := self.connection2.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(listener:      Connection;
                         newConnection: Connection);
```

The following method is an example of a **ConnectionLog** class callback method for the **listenContinuousAsynch** method, which updates the number of method invocations recorded for this method.

```
updateListenContinuousCalls(connection:    Connection;
                            newConnection: Connection) updating;
begin
    beginTransaction;
    self.numberOfListenContinuousCalls := self.numberOfListenContinuousCalls
                + 1;
    commitTransaction;
    self.newConnection.readBinaryAsynch(1024, newConnection,
                                        "readCallback");
end;
```

The **listenContinuousAsynch** method continues accepting new connection requests until the listener **Connection** class instance is closed.

The **listenContinuousCallback** method is called for every successful connection request.

See also the **Connection** class **timeout** property.

open

Signature

```
open();
```

The **open** method of the **Connection** class establishes a connection to a remote application or device and returns when the connection is established. The **open** method can be called only when the **state** is **Disconnected** (0).

Each subclass of the **Connection** class must provide any properties that are required to define the connection; for example, **name**, **socket**, and so on. The value of the **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.connection.open;
elseif bListen.value = true then
    statusLine1.caption := "Listening";
    self.connection.listen;
else
    self.connection.close;
endif;
```

openAsynch

Signature `openAsynch(receiver: Object;`
`msg: String);`

The **openAsynch** method of the **Connection** 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 name of the callback method specified in the **msg** parameter.

The **openAsynch** method can be called only when the **state** property is **Disconnected** (0). When this method is called, the value of the **state** property is changed 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 shows the use of the **openAsynch** method.

```
buttonOpenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, it is created and its properties initialized.
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // 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.connection.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(connection: Connection);`

The following method is an example of a **ConnectionLog** class callback method for the **openAsynch** method, which updates the number of method invocations recorded for this method.

```
updateOpenCalls(connection: Connection) updating;
begin
    beginTransaction;
    self.numberOfOpenCalls := self.numberOfOpenCalls + 1;
    commitTransaction;
    self.connection.readBinaryAsynch(1024, connection, "readCallback");
end;
```

openPipeCallback

Signature `openPipeCallback(pipe: InternetPipe) updating;`

The **openPipeCallback** method of the **Connection** class is called when the **jadehttp** library file opens the Internet server end of the pipe or TCP/IP connection, to initiate an asynchronous read of the opened connection.

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.

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **Connection** 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 **fillReadBuffer** property. This method can be called only when the value of the **state** property is **Connected** (2).

Only one synchronous or asynchronous read operation can be performed at one time on a connection. See also the **Connection** class **timeout** property.

The following example shows the use of the **readBinary** method.

```
openButton_click(btn: Button input) updating;
vars
    pos : Integer;
    bin : Binary;
begin
    if openButton.caption = $X_Open then
        self.connection.name := connectionName.text;
        self.connection.open;
        openButton.caption   := $X_OK;
        listenButton.caption := $X_Close;
    else
        if sendIt.value then
            if loop.value then
                self.multiSend;
            else
```

```
                self.connection.writeBinary(input.text.Binary);
            endif;
        elseif receiveIt.value then
            if loop.value then
                self.multiReceive;
            else
                self.connection.fillReadBuffer := false;
                bin := self.connection.readBinary(200);
                sl1.caption := bin.String;
            endif;
        endif;
    endif;
end;
```

readBinaryAsynch

Signature `readBinaryAsynch(length: Integer; receiver: Object; msg: String);`

The **readBinaryAsynch** method of the **Connection** 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 **fillReadBuffer** property, the object specified in the **receiver** parameter is sent the name of the callback method 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 **state** property is **Connected** (2).

The following example shows the use of the **readBinaryAsynch** method.

```
receiveAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the variable conlog to reference a ConnectionLog object.  If
    // none exists, the object is created and its properties initialized.
    if self.connection.state = Connection.Connected then
        beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
        commitTransaction;
    // 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.connection.readBinaryAsynch(50, conlog, "updateBinaryReads");
    endif;
end;
```

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 callback method must match the signature required by the calling **readBinaryAsynch** method, as follows.

Signature
```
readBinaryCallback(connection: Connection;
                   buffer:     Binary);
```

The following is an example of a **ConnectionLog** class callback method for the **readBinaryAsynch** method, which updates the number of method invocations recorded for this method.

```
updateBinaryReads(connection: Connection; buffer: Binary) updating;
begin
    beginTransaction;
    self.numberOfBinaryReads := self.numberOfBinaryReads + 1;
    commitTransaction;
end;
```

See also the **Connection** class **timeout** property.

readPipeCallback

Signature
```
readPipeCallback(pipe: InternetPipe;
                 msg:  Binary) updating;
```

The **readPipeCallback** method of the **Connection** class is called to perform Web session evaluation processing when data is available on the pipe or TCP/IP connection.

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.

readUntil

Signature
```
readUntil(delimiter: Binary;
          maxLength: Integer): Binary abstract;
```

The **readUntil** method of the **Connection** 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 one 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 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) abstract;`

The **readUntilAsynch** method of the **Connection** 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 one 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 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(connection: Connection;`
`bin: Binary);`

sendReply

Signature `sendReply(html: Binary) updating;`

The **sendReply** method of the **Connection** class sends the formatted HyperText Markup Language (HTML) page back to the opened pipe or TCP/IP connection and starts the next read request.

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.

writeBinary

Signature `writeBinary(buffer: Binary);`

The **writeBinary** method of the **Connection** 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 **state** property is **Connected** (2).

The following example shows the use of the **writeBinary** method.

```
openButton_click(btn: Button input) updating;
vars
    pos : Integer;
    bin : Binary;
begin
    if openButton.caption = $X_Open then
        self.connection.name := connectionName.text;
        self.connection.open;
        openButton.caption   := $X_OK;
        listenButton.caption := $X_Close;
    else
        if sendIt.value then
            if loop.value then
                self.multiSend;
            else
                self.connection.writeBinary(input.text.Binary);
            endif;
        elseif receiveIt.value then
            if loop.value then
                self.multiReceive;
            else
                self.connection.fillReadBuffer := false;
                bin := self.connection.readBinary(200);
                sl1.caption := bin.String;
            endif;
        endif;
    endif;
end;
```

See also the **Connection** class **timeout** property.

writeBinaryAsynch

Signature `writeBinaryAsynch(buffer: Binary; receiver: Object; msg: String);`

The **writeBinaryAsynch** method of the **Connection** 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 **state** property is **Connected** (2).

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.

```
buttonSendAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
```

```
        // Sets the conlog variable to reference a ConnectionLog object.
        // If none exists, it is created and its properties initialized.
        if self.connection.state = Connection.Connected then
            beginTransaction;
                conlog := ConnectionLog.firstInstance;
                if conlog = null then
                    create conlog;
                    conlog.numberOfListenCalls  := 0;
                    conlog.numberOfOpenCalls    := 0;
                    conlog.numberOfCloseCalls   := 0;
                    conlog.numberOfBinaryReads  := 0;
                    conlog.numberOfBinaryWrites := 0;
                endif;
            commitTransaction;
            // 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.
            self.connection.writeBinaryAsynch(textBox1.text.Binary, conlog,
                                              "updateBinaryWrites");
        endif;
    end;
```

The callback method must match the signature required by the calling **writeBinaryAsynch** method, as follows.

Signature `writeBinaryCallback(connection: Connection);`

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(connection: Connection) updating;
    begin
        beginTransaction;
        self.numberOfBinaryWrites := self.numberOfBinaryWrites + 1;
        commitTransaction;
    end;
```

See also the **Connection** class **timeout** property.

ConnectionException Class

The **ConnectionException** class is the transient class that defines behavior for exceptions that occur as a result of connecting to external systems.

A connection exception is not raised on outstanding asynchronous operations if the connection is closed from within the application.

If a connection exception occurs because a callback method could not be located on the connection receiver, the callback method name is provided by the **errorItem** property of the **Connection** superclass.

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

Inherits From: NormalException

Inherited By: (None)

ConnectionException Properties

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

Property	Contains …
connection	A reference to the connection object that caused the exception
dataBuffer	The data that the user was trying to send when the exception was raised

connection

Type: Connection

The **connection** property of the **ConnectionException** class contains a reference to the connection object on which the exception was raised provided the connection object is an instance of a **Connection** subclass.

If the connection object is an instance of the **JadeMultiWorkerTcpConnection** class (which is not a subclass of the **Connection** class), the **errorObject** method returns a reference to the connection object and the **errorItem** property returns the name of method name in which the exception was raised; for example, **JadeMultiWorkerTcpConnection::writeBinary**.

dataBuffer

Type: Binary

The **dataBuffer** property of the **ConnectionException** class contains the data that the user was trying to send at the time the exception was raised.

This property is used only in the failure of the **Connection** class **writeBinaryAsynch** method.

ConstantNDict Class

The **ConstantNDict** class is used to hold references to instances of the **Constant** class (or its subclasses; for example, **TranslatableString**).

The key of the **ConstantNDict** class is the **name** property inherited from the **SchemaEntity** class.

Inherits From: MemberKeyDictionary

Inherited By: (None)

CurrencyFormat Class

The **CurrencyFormat** class is used to store Windows locale currency information.

You cannot modify system-created instances of the **CurrencyFormat** 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 **CurrencyFormat** for each application that you can read by using **app.currentLocaleInfo.currencyInfo**. This instance contains currency information for the current locale.

CurrencyFormat instances are also used to store user-defined currency 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 currency format, see the **userCurrencyFormat** method in the **Integer**, **Real**, or **Decimal** primitive type.

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

Inherits From: NumberFormat

Inherited By: (None)

CurrencyFormat Class Constants

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

Constant	Integer Value	Example
NegCurrLeadSignTrailSpSymbol	8	-10.25 $ (space before $)
NegCurrLeadSignTrailingSymbol	5	-10.25$
NegCurrLeadSymbolSpSign	12	$ -10.25 (space after $)
NegCurrLeadSymbolSpTrailSign	11	$ 10.25- (space after $)
NegCurrLeadSymbolTrailingSign	3	$10.25-
NegCurrLeadingSignSymbol	1	-$10.25
NegCurrLeadingSignSymbolSp	9	-$ 10.25 (space after $)
NegCurrLeadingSymbolBrackets	0	($10.25)
NegCurrLeadingSymbolSign	2	$-10.25
NegCurrLeadingSymbolSpBrackets	14	($ 10.25) (space after $)
NegCurrTrailSpSymbolBrackets	15	(10.25 $) (space before $)
NegCurrTrailingSignSpSymbol	13	10.25- $ (space before $)
NegCurrTrailingSignSymbol	6	10.25-$
NegCurrTrailingSpSymbolSign	10	10.25 $- (space before $)
NegCurrTrailingSymbolBrackets	4	(10.25$)
NegCurrTrailingSymbolSign	7	10.25$-
PosCurrLeadingSymbol	0	$10.25

Constant	Integer Value	Example
PosCurrLeadingSymbolSpace	2	$ 10.25 (space after $)
PosCurrTrailingSpaceSymbol	3	10.25 $ (space before $)
PosCurrTrailingSymbol	1	10.25$

CurrencyFormat Properties

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

Property	Description
intlCurrencySymbol	Contains the international currency symbol
intlDecimalPlaces	Contains the number of digits to the right of the decimal separator in the international monetary format
negSymbolPrecedesAmount	Specifies whether the currency symbol precedes the negative monetary amount
negSymbolSeparated	Specifies whether the symbol is separated from the negative monetary amount by a space
negativeSignPosition	Contains the position of the negative sign in a negative monetary amount
posSymbolPrecedesAmount	Specifies whether the currency symbol precedes the positive monetary amount
posSymbolSeparated	Specifies whether the symbol is separated from the positive monetary amount by a space
positiveFormat	Contains the Windows positive currency format index
symbol	Contains the currency symbol for the current locale

intlCurrencySymbol

Type: String[20]

The **intlCurrencySymbol** property of the **CurrencyFormat** class contains the international currency symbol.

The international currency symbol is the three characters of the international monetary symbol specified in ISO Standard 4217, "Code for the Representation of Currencies and Funds".

intlDecimalPlaces

Type: Integer

The **intlDecimalPlaces** property of the **CurrencyFormat** class contains the number of digits to the right of the decimal in the international monetary format.

negSymbolPrecedesAmount

Type: Boolean

The **negSymbolPrecedesAmount** property of the **CurrencyFormat** class specifies whether the currency symbol precedes the negative monetary amount in the currency format; for example, **$-1.22**.

negSymbolSeparated

Type: Boolean

The **negSymbolSeparated** property of the **CurrencyFormat** class specifies whether the currency symbol is separated by a space in the negative monetary amount in the currency format; for example, **($ -1.22)**.

negativeSignPosition

Type: Integer

The **negativeSignPosition** property of the **CurrencyFormat** class contains the position of the negative sign in the currency format.

The **CurrencyFormat** class constants listed in the following table specify the negative sign position.

CurrencyFormat Class Constant	Integer Value	Example
NegCurrLeadSignTrailSpSymbol	8	-10.25 $ (space before $)
NegCurrLeadSignTrailingSymbol	5	-10.25$
NegCurrLeadSymbolSpSign	12	$ -10.25 (space after $)
NegCurrLeadSymbolSpTrailSign	11	$ 10.25- (space after $)
NegCurrLeadSymbolTrailingSign	3	$10.25-
NegCurrLeadingSignSymbol	1	-$10.25
NegCurrLeadingSignSymbolSp	9	-$ 10.25 (space after $)
NegCurrLeadingSymbolBrackets	0	($10.25)
NegCurrLeadingSymbolSign	2	$-10.25
NegCurrLeadingSymbolSpBrackets	14	($ 10.25) (space after $)
NegCurrTrailSpSymbolBrackets	15	(10.25 $) (space before $)
NegCurrTrailingSignSpSymbol	13	10.25- $ (space before $)
NegCurrTrailingSignSymbol	6	10.25-$
NegCurrTrailingSpSymbolSign	10	10.25 $- (space before $)
NegCurrTrailingSymbolBrackets	4	(10.25$)
NegCurrTrailingSymbolSign	7	10.25$-

posSymbolPrecedesAmount

Type: Boolean

The **posSymbolPrecedesAmount** property of the **CurrencyFormat** class specifies whether the currency symbol precedes the positive monetary amount in the currency format; for example, **$1.22**.

posSymbolSeparated

Type: Boolean

The **posSymbolSeparated** property of the **CurrencyFormat** class specifies whether the currency symbol is separated by a space in the positive monetary amount in the currency format; for example, **$ 1.22**.

positiveFormat

Type: Integer

The **positiveFormat** property of the **CurrencyFormat** class contains the Windows positive currency format index.

The **CurrencyFormat** class constants, listed in the following table, specify the positive monetary amount sign position. (In these examples, the dollar symbol (**$**) represents any currency symbol defined by the **symbol** property.)

CurrencyFormat Class Constant	Value	Example	Comment
PosCurrLeadingSymbol	0	$1.1	
PosCurrTrailingSymbol	1	1.1$	
PosCurrLeadingSymbolSpace	2	$ 1.1	Space after $
PosCurrTrailingSpaceSymbol	3	1.1 $	Space before $

symbol

Type: String[20]

The **symbol** property of the **CurrencyFormat** class contains the monetary symbol for the currency format; for example, **$**.

CurrencyFormat Method

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

Property	Description
defineCurrencyFormat	Defines the characteristics of a currency format

defineCurrencyFormat

Signature

```
defineCurrencyFormat(numberOfDecimalPlaces: Integer;
                     decimalSep:            String;
                     thousandSep:           String;
                     posFormat:             Integer;
                     negFormat:             Integer;
                     showLeadingZero:       Boolean;
                     currencySymbol:        String) updating;
```

The **defineCurrencyFormat** method of the **CurrencyFormat** class enables you to dynamically define the characteristics of a currency format. (For details about returning a string containing the receiver in the supplied currency format, see the **userCurrencyFormat** method in the **Integer**, **Real**, or **Decimal** primitive type.)

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 on the database node is set to **false** or it is not defined, 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. To forward regional overrides set on the presentation client to the application server so that both use consistent locale settings for the application, set the **EnhancedLocaleSupport** parameter to **true**.

Set the **numberOfDecimalPlaces** parameter to the number of decimal places that you want displayed. You must specify a number in the range **0** through **9**. A value of zero (**0**) is assumed if you specify a value less than zero (**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 **CurrencyFormat** class constants listed in the following table in the **posFormat** parameter, **CurrencyFormat.PosCurrLeadingSymbol** is assumed.

CurrencyFormat Class Constant	Integer Value	Example	Comment
PosCurrLeadingSymbol	0	$1.1	
PosCurrTrailingSymbol	1	1.1$	
PosCurrLeadingSymbolSpace	2	$ 1.1	Space after $
PosCurrTrailingSpaceSymbol	3	1.1 $	Space before $

If you do not specify one of the **CurrencyFormat** class constants listed in the following table in the **negFormat** parameter, **CurrencyFormat.NegCurrLeadingSymbolBrackets** is assumed.

CurrencyFormat Class Constant	Integer Value	Example	Comment
NegCurrLeadSignTrailSpSymbol	8	-10.25 $	(space before $)
NegCurrLeadSignTrailingSymbol	5	-10.25$	
NegCurrLeadSymbolSpSign	12	$ -10.25	(space after $)
NegCurrLeadSymbolSpTrailSign	11	$ 10.25-	(space after $)
NegCurrLeadSymbolTrailingSign	3	$10.25-	
NegCurrLeadingSignSymbol	1	-$10.25	
NegCurrLeadingSignSymbolSp	9	-$ 10.25	(space after $)
NegCurrLeadingSymbolBrackets	0	($10.25)	
NegCurrLeadingSymbolSign	2	$-10.25	
NegCurrLeadingSymbolSpBrackets	14	($ 10.25)	(space after $)
NegCurrTrailSpSymbolBrackets	15	(10.25 $)	(space before $)
NegCurrTrailingSignSpSymbol	13	10.25- $	(space before $)
NegCurrTrailingSignSymbol	6	10.25-$	

CurrencyFormat Class Constant	Integer Value	Example	Comment
NegCurrTrailingSpSymbolSign	10	10.25 $-	(space before $)
NegCurrTrailingSymbolBrackets	4	(10.25$)	
NegCurrTrailingSymbolSign	7	10.25$-	

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 this parameter to **false** if you do not want to display a leading zero (**0**).

Use the **currencySymbol** parameter to specify a string of up to five characters that is to be used as the currency symbol (for example, **"$"**). If the string contains any numeric characters, these numeric characters are removed. The string is truncated if it is longer than five characters.

Database Class

The **Database** class encapsulates the definition of a database for a schema, including the database files and the class mappings to those files, and the behavior required to access entries in the database.

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

Inherits From: SchemaEntity

Inherited By: (None)

Database Properties

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

Property	Description
dbFiles	Contains the names of the database files
path	Contains the database path
schema	Contains a reference to the database schema
serverName	Contains the name of the database server

dbFiles

Type: DbFileNDict

The read-only **dbFiles** property of the **Database** class contains a reference to a collection of **DbFile** objects. For details, see the **DbFile** class.

path

Type: String

The read-only **path** property of the **Database** class contains the full database path; for example, **"s:\jade\system"**.

schema

Type: Schema

The read-only **schema** property of the **Database** class contains a reference to the schema in which the database is defined.

serverName

Type: String[30]

The **serverName** property of the **Database** class contains the name of the database server; for example, **"JADE_Dev_2"**.

Database Methods

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

Method	Description
getFile	Returns the specified file
getName	Returns the name of the database

getFile

Signature `getFile(name: String): DbFile;`

The **getFile** method of the **Database** class returns a reference to the database file specified in the **name** parameter.

getName

Signature `getName(): String;`

The **getName** method of the **Database** class returns a string representing the name of the database.

DateArray Class

The **DateArray** class is an ordered collection of **Date** values in which the values are referenced by their position in the collection.

Date arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a **Date** array.

Inherits From: Array

Inherited By: (None)

DateFormat Class

The **DateFormat** class is used to store Windows locale date information. You cannot modify system-created instances of the **DateFormat** 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 **DateFormat** for each application that you can read by using **app.currentLocaleInfo.dateInfo**. This instance contains date information for the current locale. **DateFormat** instances are also used to store user-defined date 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.

Notes Although the Windows environment does not allow dates earlier than 1601 (**Date** primitive type format methods return **"*invalid*"**), JADE stores dates that are 1600 or earlier.

For details about returning a string containing the receiver in the supplied date format, see the **Date** primitive type **userFormat** method.

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

Inherits From: LocaleFormat

Inherited By: (None)

DateFormat Class Constants

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

Constant	Integer Value	Constant	Integer Value
None	0	Gregorian	1
EnglishGregorian	2	JapaneseEra	3
ChineseEra	4	KoreanEra	5
Hijri	6	Thai	7
MonthDayYear	0	DayMonthYear	1
YearMonthDay	2	WeekOfJan1	0
FirstWeekAfterJan1	1	FirstWeekWith4Days	2
MonthFullName	1	MonthShortName	2
MonthNumberLeadingZero	3	MonthNumber	4
FullDayOfWeek	1	ShortDayOfWeek	2
NoDayOfWeek	3		

DateFormat Properties

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

Property	Description
activeCalendarType	Contains the type of calendar that is currently used
dayHasLeadingZeros	Specifies if the day number of the short date format contains leading zeros
firstDayOfWeek	Contains the day that is considered to be the first day of the week
firstWeekOfYear	Contains the week that is considered to be the first week of the year
longDayNames	Contains an array of the long day names for the locale
longFormat	Contains the long date formatting string for the locale
longFormatOrder	Contains the order of the long date format for the locale
longMonthNames	Contains an array of the long month names for the locale
monthHasLeadingZeros	Specifies if the month number of the short date format contains leading zeros
optionalCalendarType	Contains the additional calendar types that are available
separator	Contains the separator used in the short date format for the locale
shortDayNames	Contains an array of the short format day names for the locale
shortFormat	Contains the short formatting string for the locale
shortFormatOrder	Contains the order of the short date format for the locale
shortMonthNames	Contains an array of the short month names for the locale
showFullCentury	Specifies if the short date format is to display the full four-digit year

activeCalendarType

Type: Integer

The **activeCalendarType** property of the **DateFormat** class contains a reference to the calendar type that is currently used.

The calendar type can be one of the values listed in the following table.

Value	Class Constant	Description
1	Gregorian	Gregorian (localized)
2	EnglishGregorian	Gregorian (always English strings)
3	JapaneseEra	Japanese era (Year of the Emperor)
4	ChineseEra	Year of the Republic of China
5	KoreanEra	Tangun era (Korea)
6	Hijri	Hijri
7	Thai	Thai

dayHasLeadingZeros

Type: Boolean

The **dayHasLeadingZeros** property of the **DateFormat** class is set to **true** if the short format day numbers have leading zeros.

firstDayOfWeek

Type: Integer

The **firstDayOfWeek** property of the **DateFormat** class contains the day that is considered the first day of the week in the locale. The **firstDayOfWeek** property values are listed in the following table.

Value	Description	Value	Description
1	Monday	5	Friday
2	Tuesday	6	Saturday
3	Wednesday	7	Sunday
4	Thursday		

firstWeekOfYear

Type: Integer

The **firstWeekOfYear** property of the **DateFormat** class contains the week that is considered the first week of the year in the locale. The **firstWeekOfYear** property values are listed in the following table.

Value	Class Constant	Description
0	WeekOfJan1	Week containing the first day of the first month
1	FirstWeekAfterJan1	First full week following the first day of the first month
2	FirstWeekWith4Days	First week containing at least four days

longDayNames

Type: StringArray

The **longDayNames** property of the **DateFormat** class contains a reference to an array of the long names of days for the locale; for example, **Wednesday**.

longFormat

Type: String[127]

The **longFormat** property of the **DateFormat** class contains the long date format string for the locale.

longFormatOrder

Type: Integer

The **longFormatOrder** property of the **DateFormat** class contains the order of the long date format for the locale. The **longFormatOrder** property values are listed in the following table.

Value	Class Constant	Description
0	MonthDayYear	Month-Day-Year
1	DayMonthYear	Day-Month-Year
2	YearMonthDay	Year-Month-Day

longMonthNames

Type: StringArray

The **longMonthNames** property of the **DateFormat** class contains a reference to an array of the long names of months for the locale; for example, **December**.

The string array contains 12 entries, unless the locale defines a short or long name for the thirteenth month.

monthHasLeadingZeros

Type: Boolean

The **monthHasLeadingZeros** property of the **DateFormat** class is set to **true** if the short format month numbers have leading zeros.

optionalCalendarType

Type: Integer

The **optionalCalendarType** property of the **DateFormat** class contains the optional calendar type that is available for the locale.

The calendar type can be one of the values listed in the following table.

Value	Class Constant	Description
0	None	No additional calendar types
1	Gregorian	Gregorian (localized)
2	EnglishGregorian	Gregorian (always English strings)
3	JapaneseEra	Japanese era (Year of the Emperor)
4	ChineseEra	Year of the Republic of China
5	KoreanEra	Tangun era (Korea)
6	Hijri	Hijri
7	Thai	Thai

separator

Type: String[20]

The **separator** property of the **DateFormat** class contains the character used for the date separator in the locale; for example, **"/"**.

shortDayNames

Type: StringArray

The **shortDayNames** property of the **DateFormat** class contains a reference to an array of the short names of days for the locale; for example, **Wed**.

shortFormat

Type: String[127]

The **shortFormat** property of the **DateFormat** class contains the short date format string for the locale.

shortFormatOrder

Type: Integer

The **shortFormatOrder** property of the **DateFormat** class contains the order of the short date format for the locale. The **shortFormatOrder** property values are listed in the following table.

Value	Class Constant	Description
0	MonthDayYear	Month-Day-Year
1	DayMonthYear	Day-Month-Year
2	YearMonthDay	Year-Month-Day

shortMonthNames

Type: StringArray

The **shortMonthNames** property of the **DateFormat** class contains a reference to an array of the short names of months for the locale; for example, **Dec**.

The string array contains 12 entries, unless the locale defines a short or long name for the thirteenth month.

showFullCentury

Type: Boolean

The **showFullCentury** property of the **DateFormat** class is set to **true** if the four-digit year is to be displayed in the short date format; for example, **1999**.

DateFormat Methods

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

Method	Description
defineLongDateFormat	Defines the characteristics of a long date format
defineShortDateFormat	Defines the characteristics of a short date format

defineLongDateFormat

Signature

```
defineLongDateFormat(showDayWithLeadingZero: Boolean;
                     monthFormat:            Integer;
                     formatOrder:            Integer;
                     separator1:             String;
                     separator2:             String;
                     separator3:             String;
                     showFullYear:           Boolean;
                     dayOfWeek:              Integer) updating;
```

The **defineLongDateFormat** method of the **DateFormat** class enables you to dynamically define the characteristics of a long date format. (For details about returning a string containing the receiver in the supplied date format, see the **Date** primitive type **userFormat** method.) 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.

Set the **showDayWithLeadingZero** parameter to **true** if you want the day of the month to be displayed with a leading zero if it is less than **10**. If you do not want to display a leading zero, set this parameter to **false**.

If you do not set the **monthFormat** parameter to one of the following valid **DateFormat** class constant values, **DateFormat.MonthFullName** is assumed.

- **DateFormat.MonthFullName** (for example, **March**)
- **DateFormat.MonthShortName** (for example, **Mar**)
- **DateFormat.MonthNumber** (for example, **3**)
- **DateFormat.MonthNumberLeadingZero** (for example, **03**)

If you do not set the **formatOrder** parameter to one of the following valid **DateFormat** class constant values, **DateFormat.DayMonthYear** is assumed.

- **DateFormat.DayMonthYear** (for example, **5 April 2000**)
- **DateFormat.MonthDayYear** (for example, **April 5 2000**)
- **DateFormat.YearMonthDay** (for example, **2000 April 5**)

When the **longFormatOrder** property is set to **DayMonthYear** (**1**), use the **separator1**, **separator2**, and **separator3** parameters to specify a string of up to five characters that is to be displayed between the day name and the day of the month, the day of the month and the month name, and the month name and the year, respectively. If the strings contain any of the **d**, **M**, **y**, **g**, **h**, **H**, **m**, **s**, or **t** characters, these characters are removed. If the strings are longer than five characters, they are truncated to five characters.

When the **longFormatOrder** property is set to **MonthDayYear** (**0**) or **YearMonthDay** (**2**), **separator1** goes between day name and the date part (or in front of date, if the format specified in the **longFormat** property contains no day name), **separator2** goes between first two parts of the date (that is, day and month, month and day, or year and month, depending on the value of the **longFormatOrder** property), and **separator3** goes between last two parts of the date (that is, month and year, day and year, or month and day, depending on the value of the **longFormatOrder** property).

Set the **showFullYear** parameter to **true** if you want to display a full four-digit year or set it to **false** if you want to display a two-digit year.

Use the **dayOfWeek** parameter to specify whether the full name of the day of the week, the short name, or no day of the week name is to be displayed. If you do not set the **dayOfWeek** parameter to one of the following valid **DateFormat** class constant values, **DateFormat.FullDayOfWeek** is assumed.

- **DateFormat.FullDayOfWeek** (for example, **Sunday**)
- **DateFormat.NoDayOfWeek** (suppresses the display of the day name)
- **DateFormat.ShortDayOfWeek** (for example, **Sun**)

Note Although the Windows environment does not allow dates earlier than 1601 (**Date** primitive type format methods return **"*invalid*"**), JADE stores dates that are 1600 or earlier.

defineShortDateFormat

Signature

```
defineShortDateFormat(showDayWithLeadingZero:   Boolean;
                      showMonthWithLeadingZero: Boolean;
                      formatOrder:              Integer;
                      dayMonthYearSeparator:    String;
                      showFullYear:             Boolean) updating;
```

The **defineShortDateFormat** method of the **DateFormat** class enables you to dynamically define the characteristics of a short date format. (For details about returning a string containing the receiver in the supplied date format, see the **Date** primitive type **userFormat** method.) 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.

Set the **showDayWithLeadingZero** and **showMonthWithLeadingZero** parameters to **true** if you want the day of the month and the month number, respectively, to be displayed with a leading zero if they are less than **10**. If you do not want a leading zero displayed before the day, set this parameter to **false**.

If you do not set the **formatOrder** parameter to one of the following valid **DateFormat** class constant values, **DateFormat.DayMonthYear** is assumed.

- **DateFormat.DayMonthYear** (for example, **04/25/00**)
- **DateFormat.MonthDayYear** (for example, **25/4/00**)
- **DateFormat.YearMonthDay** (for example, **00/04/25**)

The **dayMonthYearSeparator** parameter enables you to specify a string of up to five characters that is to be displayed between the day, month, and year. 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 five characters, it is truncated to five characters.

Set the **showFullYear** parameter to **true** if you want to display a full four-digit year (for example, **2001**). If you want to display a two-digit year (for example, **01**), set this parameter to **false**.

Note Although the Windows environment does not allow dates earlier than 1601 (**Date** primitive type format methods return **"*invalid*"**), JADE stores dates that are 1600 or earlier.

DbFile Class

The **DbFile** class encapsulates the definition of a database file and provides methods to perform file-level operations; for example, to partition database files and iterate partitions.

For details about the constants, properties, methods, and event notifications defined in the **DbFile** class, see "DbFile Class Constants", "DbFile Properties", "DbFile Methods", and "DbFile Class Event Notifications", in the following subsections. For details about using the JADE database administration framework in your own applications, see Chapter 7, "Using the Database Administration Framework", of the *JADE Developer's Reference*.

Inherits From: SchemaEntity

Inherited By: (None)

DbFile Class Constants

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

Constant	Description	Integer Value
BackupBytesDoneEvent	File backup byte progress event	1002
BackupErrorEvent	File backup error event	1004
BackupOperationEvent	File backup operation event	1001
BackupOutputEvent	File backup output event	1003
BackupProgressEvent	File backup progress event	1000
CryptStatus_Encrypted	Encrypted	4
CryptStatus_Not	Not encrypted	0
CryptStatus_Pending	Pending encryption or decryption	1
CryptStatus_ReencryptPending	Pending re-encryption	5
EnableAudit_NoCompress	Auditing is enabled	1
GetTotLen_Base	Total length of base file	1
GetTotLen_Partitions	Total length of partitions	2
GetTotLen_SharedFileUDRs	Total length of shared file UDRs	4
GetTotLen_SingleFileUDRs	Total length of shared file UDRs	8
GetTotLen_Everything	Total length of all subfiles	255
Kind_Control	Control files (**_control** and **_reorg** database files)	1
Kind_Environmental	Environmental files (**_locks**, **_environ**, and **_stats** database files)	2
Kind_System	System files (**_system**, **_sysxrf**, **_sysgui**, **_sysint**, **_sysdev**, **_systools**, **_jadeapp**, **_jadedef**, and **_sysdef** database files)	4
Kind_Unknown	Unknown (when detected, raises an exception)	0

Constant	Description	Integer Value
Kind_User_Data	User data files (**_sindexes.dat**, **_sindexdefs.dat**, and **_rootdef** database files; and additional database files defined in user schemas)	32
Kind_User_Schema	User schema files (**_userscm**, **_userxrf**, **_usergui**, **_userint**, and **_userdev** database files)	8
Kind_Utility	Utility files (**_monitor** and **rpstrans** database files)	16
Mode_ReadOnly	Read-only database file access mode	1
Mode_Update	Update database file access mode	0
Status_DelEncrypted	Encrypted file deleted from the control file	9
Status_Deleted	File deleted from the control file	6
Status_InvalidPath	Invalid database file path in the control file	7
Status_Missing	File defined in the schema but does not exist in the database	3
Status_NotAssigned	File not defined in the control file	1
Status_NotCreated	File deleted or not yet created	2
Status_Offline	File is offline	8
Status_Resident	File is resident on disk	4
Status_Unmapped	File in RPS database that is not part of the RPS mapping	5

DbFile Properties

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

Property	Description
database	Contains the name of the database
excludeFromBackup	Specifies whether the file is excluded from database backup
kind	Contains the kind, or category, of database file
partitionable	Specifies whether the file is partitionable
path	Contains the database file path

database

Type: Database

The read-only **database** property of the **DbFile** class contains a reference to the database in which the database file is defined.

excludeFromBackup

Type: Boolean

The read-only **excludeFromBackup** property of the **DbFile** class is initialized by internal schema maintenance routines when a **DbFile** instance is created.

User backup methods, which enumerate database files using the **JadeDatabaseAdmin** class **getDbFiles** method, should test this property before calling the **DbFile** class **backupFile** method. If this property is set to **true**, the file should be excluded from backups. This property is set to **true** on:

- System files; that is, when the **kind** property contains **Kind_System** (indicating that the database category contains **_system**, **_sysxrf**, **_sysgui**, **_sysint**, **_sysdev**, **_systools**, **_ jadeapp**, **_ jadedef**, and **_sysdef** files)
- Certain control and environmental files, including the **_control**, **_reorg**, and **_environ** files

kind

Type: Integer

The read-only **kind** property of the **DbFile** class contains the kind, or category, of the database. The categories, represented by **DbFile** class constants, are listed in the following table.

Constant	Description	Integer Value
Kind_Control	Control files (**_control** and **_reorg** database files)	1
Kind_Environmental	Environmental files (**_locks**, **_environ**, and **_stats** database files)	2
Kind_System	System files (**_system**, **_sysxrf**, **_sysgui**, **_sysint**, **_sysdev**, **_ systools**, **_ jadeapp**, **_ jadedef**, and **_sysdef** database files)	4
Kind_Unknown	Unknown (when detected, raises an exception)	0
Kind_User_Data	User data files (**_sindexes.dat**, **_sindexdefs.dat**, and **_rootdef** database files, and additional database files defined in user schemas)	32
Kind_User_Schema	User schema files (**_userscm**, **_userxrf**, **_usergui**, **_userint**, and **_userdev** database files)	8
Kind_Utility	Utility files (**_monitor** and **_rpstrans** database files)	16

partitionable

Type: Boolean

The read-only **partitionable** property of the **DbFile** class specifies whether partitioning rules are to be enforced for the **DbFile** instance. This property can be set if at most one class is mapped to the file.

Tip You may want to allow instances of a class (for example, the **Sale** class, which is mapped to **sale.dat**) to be partitioned when the system is deployed. (The decision is to be left to the administrator of the deployed system.) To prevent another class being mapped inadvertently to **sale.dat** if it has not been physically partitioned in the development environment, set the **partitionable** property to **true**.

When the **partitionable** property is set to **true**, the compiler and the JADE development environment enforce file partitioning rules for the schema-defined **DbFile** instance irrespective of whether the file is physically partitioned.

Setting the **partitionable** property to **false** does not prevent a file from being partitioned. To enable partitioning on an empty database file, execute the **setPartitioned** method of the **DbFile** class passing **true** as the parameter. When partitioning is enabled on a database file, the **partitionable** property is also set if it was not already set.

You can change the value of the **partitionable** property:

- From the JADE development environment, by checking the **Partitionable** check box on the File dialog. This check box is disabled if it is not valid for the selected file. If you check the check box for a new file, JADE ensures that the file is not used elsewhere.
- During a batch schema load.
- By calling the **DbFile** class **setPartitioned** method.

When the value of the **partitionable** property is set to **true**, there can be one partitionable file only with a specific name and number in the database, and collections cannot be mapped to the file.

path

Type: String

The read-only **path** property of the **DbFile** class contains the full database file path if the file is located in a directory other than the default system location. If the file is in the default system location, the value of the **path** property is an empty string.

DbFile Methods

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

Method	Description
backupFile	Backs up a physical database file
beginPartitionedFileBackup	Begins the backup of selected database partitions, which can also be backed up in parallel
certifyFile	Initiates the certification of a database file
changeAccessMode	Changes the access mode of a file to read-only or updateable
compactFile	Initiates the compaction of a database file
createPartition	Creates a new empty partition and returns the partition identifier
disableAuditing	Disables auditing associated with object operations performed against the file
drop	Removes the file and marks it as deleted
enableAuditing	Re-enables the auditing associated with object operations
endPartitionedFileBackup	Ends the backup of selected database partitions
freeze	Converts a partition to read-only mode, after which no object update, delete, or create are permitted
getBackupTimestamp	Returns a timestamp containing the date and time the database file was last backed up
getCreationTimestamp	Returns a timestamp containing the date and time the database file was created
getCryptStatus	Returns the encryption status of a physical database file

Method	Description
getFileLength	Returns the size of a physical database file
getFileStatus	Returns the status of a physical database file during the backup process
getFreeSpace	Evaluates the available free space in a database file
getFullBackupTimestamp	Returns a timestamp containing the date and time the database file was last backed up
getModifiedTimestamp	Returns a timestamp containing the date and time the database file was last updated
getName	Returns the name of the database file
getOpenPartitions	Populates the input **partitionList** array with references to **JadeDbFilePartition** instances; one for each open partition of the associated database file
getPartition	Returns the **JadeDbFilePartition** instance associated with the indexed partition
getPartitionCount	Returns the number of non-removed partitions assigned to the file
getPartitionModulus	Returns the number of partitions in which new instances are stored
getPartitions	Populates the input **partitionList** array with **JadeDbFilePartition** instances; one for each partition of the associated database file
getPatchVersion	Returns the patch version numbers for the system files
getStatistics	Returns statistics on reads database activity
getTotalFileLength64	Returns the total bytes occupied by subfiles of a database map file
getUserPatchVersion	Returns the unformatted version number of user data map files
isAuditing	Returns **true** if auditing associated with object operations is enabled
isFrozen	Returns **true** if the database file is frozen
isOpen	Returns **true** if the database file is currently open
isPartitioned	Returns **true** if the database file is partitioned
setPartitionModulus	Specifies the modulus; that is, the number of partitions in which new instances are stored
setPartitioned	Changes the partitioned attribute of an empty (non-instantiated) database file
thaw	Restores the database partition to its default active state

backupFile

Signature

```
backupFile(backupDir:       String;
           verifyChecksums: Boolean;
           compress:        Boolean;
           overwriteDest:   Boolean);
```

Note The database must be in backup state (that is, the online file backup operation must be bracketed by a **beginBackup** and **commitBackup** transaction pair).

The **backupFile** method of the **DbFile** class initiates a back up of the physical database file to the directory specified in the **backupDir** parameter. (The backup directory must be a valid directory on the database server.) This method executes on the database server node, and is implemented and executed by the database engine. The backup process performs various consistency checks similar to a database certify, to ensure the integrity of the backup.

Set the **verifyChecksums** parameter to **true** if you want checksums verified in the backed up file. Checksum verification is performed in a separate pass of the backed up file immediately after the copy phase. A checksum analysis of your backed up database files verifies that the files have not been corrupted by a hardware or environmental problem during the backup process. You should perform a separate checksum analysis of any backup that has been moved across media, especially if transferred across a network.

Set the **compress** parameter to **true** if you want to compress backed up data. You can compress data in a checked or an unchecked backup.

Set the **overwriteDest** parameter to **true** if you want to allow file backups to overwrite existing files in the destination backup directory. When this parameter is **false**, an exception is raised if an existing file is detected.

The code fragment in the following example shows the use of the **backupFile** method.

```
if not dbFile.excludeFromBackup then
    dbFile.backupFile(null,      // use default directory
                      true,      // verify checksums during backup
                      true,      // request data compression
                      false);    // disallow overwrite of existing files
endif;
```

Separate JADE processes can initiate concurrent file backups. This allows multiple files to be copied concurrently, which you can use to reduce elapsed backup time when the source and destination volumes are on different physical devices.

Caution Because of increased disk contention and disk head movement, concurrent backup operations run slower if the backup is sent to a single disk drive.

You can use the **JadeDatabaseAdmin** class **enableProgressEvents** method to optionally notify operation and progress notifications for file backups. You must both enable and subscribe to this event if you want file backup operation and progress notification.

See also "DbFile Class Event Notifications", later in this section.

beginPartitionedFileBackup

Signature

```
beginPartitionedFileBackup(backupDir:      String;
                           verifyChecksums: Boolean;
                           compress:        Boolean;
                           overwriteDest:   Boolean) serverExecution;
```

The **beginPartitionedFileBackup** method of the **DbFile** class enables you to prepare to back up a subset of the partitions associated with a database file or to back up multiple partitions in parallel. Execution of this method backs up the partition control file and the partition index file.

This method must be called before the first partition is backed up by using the **backupFilePartition** method of the **JadeDbFilePartition** class.

If a parallel or selective backup of partitions is not required, use the **backupFile** method.

A **beginPartitionedFileBackup** method call must later be closed with an **endPartitionedFileBackup** method call.

An exception is raised if the database file is not partitioned.

certifyFile

Signature `certifyFile(): Integer;`

The **certifyFile** method of the **DbFile** class initiates an online certification of the physical database file; that is, it checks the database integrity.

The file must be stable when the certify operation is performed. In addition, the file access mode must be **read-only**, or the database mode must be **exclusive** or **archive**. When the database mode is **archive**, all files are effectively read only, as the database is in a quiescent state. When the database mode is **exclusive**, you must take responsibility to coordinate transaction activity with the certify operation.

The **certifyFile** method returns the number of errors that were detected when certifying the database file.

This method executes on a persistent server node, and is implemented and executed by the physical database engine. For details, see "Using the Certify Files Command", in Chapter 1 of the *JADE Database Administration Guide*.

changeAccessMode

Signature `changeAccessMode(mode: Integer);`

The **changeAccessMode** method of the **DbFile** class changes the access mode of the file from read-only (**Mode_ReadOnly**) to update (**Mode_Update**), or the reverse. Use this method to allow applications to compact database files without shutting down the database. This method executes on a persistent server node, and is implemented and executed by the physical database engine.

When a file is changed to read-only mode, any threads that attempt to start a new transaction are first blocked, waiting for pending transactions to complete. All dirty buffers for objects resident in the file are flushed and the file is locked against further updates. At this point, threads blocked by the **beginTransaction** instruction are allowed to continue.

Any database operations that attempt to update a file in read-only mode receive an exception.

compactFile

Signature `compactFile(): Integer;`

The **compactFile** method of the **DbFile** class initiates a compaction of the physical database file. This method returns the number of errors that were detected when compacting the database file.

The **compactFile** method executes on a persistent server node, and is implemented and executed by the physical database engine.

A database file can be compacted while permitting concurrent transactions to update the file. Moreover, the compaction operation can be aborted without losing updates made by committed transactions.

The following example shows the use of the **compactFile** method.

```
onlineCompact();
vars
    dba : JadeDatabaseAdmin;
    dbFile : DbFile;
    dbfiles : DbFileArray;
begin
```

```
        create dba;
        create dbfiles transient;
        dba.getDbFiles(DbFile.Kind_User_Data, dbfiles);
        foreach dbFile in dbfiles do
            dbFile.compactFile;
        endforeach;
    epilog
        delete dba;
        delete dbfiles;
    end;
```

For details, see "Compacting Files", in Chapter 3 of the *JADE Database Administration Guide*.

createPartition

Signature `createPartition(): Integer64;`

The **createPartition** method of the **DbFile** class creates a new empty database file partition and returns the partition identifier.

You can make a number of related **createPartition** operations atomic, by containing them in the same database transaction, as shown in the following example.

```
// execute just before midnight at end of current period
beginTransaction;
    Order.getDbFile.createPartition;
    OrderItem.getDbFile.createPartition;
// execute just after midnight at start of next period
commitTransaction;
```

When the **createPartition** operation is invoked on a partitioned database file, any transactions that attempt to create new instances are blocked until the transaction that invoked the **createPartition** operation commits or aborts.

An exception is raised if the database file is locked for reorganization or if the file is not partitioned.

The following restrictions apply to the use of the **createPartition** method.

- Partitions can be created only within a transaction
- No other partition creation operation can be in progress
- Persistent objects cannot be created or updated in the transaction that creates a partition
- Persistent objects cannot be created in a partitioned file by any user while a new partition for that file is being created

Note For a production application, you should implement a synchronization mechanism to prevent the creation of objects stored in a partitioned file while a new partition is created.

disableAuditing

Signature `disableAuditing(maxWaitForQuietpoint: Integer);`

The **disableAuditing** method of the **DbFile** class disables the auditing associated with object operations performed against the file.

Auditing is disabled within a quiet point after a checkpoint has successfully completed to move the database recovery point. If the **maxWaitForQuietpoint** parameter has a non-zero value, it specifies the maximum time in seconds that the operation will wait for there to be no transaction activity, and overrides the configured or default database value specified by the **MaxWaitForQuietPoint** parameter in the [PersistentDb] section of the JADE initialization file.

If a quiet point cannot be established, exception 3077 (*Maximum time to wait for quiet point was exceeded*) is raised.

drop

Signature `drop();`

The **drop** method of the **DbFile** class removes the file and marks it as deleted.

enableAuditing

Signature `enableAuditing(options: Integer);`

The **enableAuditing** method of the **DbFile** class re-enables the auditing associated with object operations performed against the file.

File operations are blocked and the file is made stable. A copy of the file is then inserted into the audit stream.

By default, the file is compressed before being written into the journal. If you know that the data in the file compresses poorly, you can disable compression by specifying the value **EnableAudit_NoCompress** (a **DbFile** class constant) for the **options** parameter.

File compression and decompression operations use the directory specified by the **ReorgWorkDirectory** parameter in the [JadeReorg] section of the JADE initialization file.

endPartitionedFileBackup

Signature `endPartitionedFileBackup() serverExecution;`

The **endPartitionedFileBackup** method of the **DbFile** class signals the end of a backup of selected database partitions. Call this method after the application has backed up the required partitions.

An exception is raised if the database file is not partitioned.

freeze

Signature `freeze() updating;`

The **freeze** method of the **DbFile** class converts a database file to read-only mode after which any object update, delete, or create operations are not be permitted. (See also the **thaw** method.)

Note All objects in a frozen database file are automatically frozen, overriding individual volatility state.

getBackupTimestamp

Signature `getBackupTimestamp(): TimeStamp;`

The **getBackupTimestamp** method of the **DbFile** class returns a timestamp containing the date and time the database file was last backed up.

getCreationTimestamp

Signature `getCreationTimestamp(): TimeStamp;`

The **getCreationTimestamp** method of the **DbFile** class returns a timestamp containing the date and time the database file was created.

getCryptStatus

Signature `getCryptStatus(): Integer;`

The **getCryptStatus** method of the **DbFile** class returns an **Integer** value representing the encryption status of the receiving individual database map file. Partitions share the same crypt state as their parent map file.

The **DbFile** class constants listed in the following table represent the file encryption status.

Class Constant	Integer Value	The database file of the receiver is…
CryptStatus_Encrypted	4	Encrypted
CryptStatus_Not	0	Not encrypted
CryptStatus_Pending	1	Pending encryption or decryption
CryptStatus_ReencryptPending	5	Pending re-encryption

getFileLength

Signature `getFileLength(): Decimal;`

The **getFileLength** method of the **DbFile** class returns the size of the physical database file in bytes as a decimal value.

Note The maximum size of a database file is 2^64-1 (or approximately 16 Exabytes), which requires 19 digits of precision for storage. To handle up to the maximum possible database file size, you should use a **Decimal[19]** primitive type to receive the return value.

This method executes on a persistent server node, and is implemented and executed by the physical database engine.

getFileStatus

Signature `getFileStatus(): Integer;`

The **getFileStatus** method of the **DbFile** class returns the status of dropped physical database file during the backup process. The status of the database file is represented by **DbFile** class constants listed in the following table.

Constant	Description	Integer Value
Status_Deleted	File deleted from control file	6
Status_InvalidPath	Invalid database file path in control file	7
Status_Missing	File defined in the schema but does not exist in the database	3
Status_NotAssigned	File not defined in control file	1

Constant	Description	Integer Value
Status_NotCreated	File deleted or not yet created	2
Status_Offline	File is offline	8
Status_Resident	File is resident on disk	4
Status_Unmapped	File in RPS database that is not part of the RPS mapping	5

You can use this method in backup applications to determine the status of database files prior to commencing the backup or to determine the status of database files returned in the **droppedFiles** parameter array passed to **JadeDatabaseAdmin** class **backupAllDbFiles** and **backupDbFiles** methods.

This method executes on a persistent server node, and is implemented and executed by the physical database engine.

getFreeSpace

Signature `getFreeSpace(freeSpace: Decimal output): Integer;`

The **getFreeSpace** method of the **DbFile** class evaluates the total amount of free space in the physical database file and returns the amount as a **Decimal** value.

The **getFreeSpace** method returns the number of errors encountered, if any, while performing the evaluation operation.

You can execute the **getFreeSpace** method when one of the following conditions is true.

- The file is read-only
- The database is open with **update** usage; however, if the database mode is not **exclusive** and is not **archive**, the file access mode must be **read-only**
- The database is in archive mode (**quiesced** mode)

The following example shows the use of the **getFreeSpace** method.

```
vars
    dbFile : DbFile;
    free : Decimal[20,0];
begin
    foreach dbFile in DbFile.instances do
        if dbFile.name = "customer" then
            dbFile.changeAccessMode(DbFile.Mode_ReadOnly);
            dbFile.getFreeSpace(free);
            dbFile.changeAccessMode(DbFile.Mode_Update);
        endif;
    endforeach;
    write free;
end;
```

This method executes on a persistent server node, and is implemented and executed by the physical database engine. For details, see "Evaluating Free Space", in Chapter 3 of the *JADE Database Administration Guide*.

getFullBackupTimestamp

Signature `getFullBackupTimestamp(): TimeStamp;`

The **getFullBackupTimestamp** method of the **DbFile** class returns a timestamp containing the date and time the full database file was *stable* in a backup where *stable* means the database file was not updated during the backup and therefore does not require recovery.

getModifiedTimestamp

Signature `getModifiedTimestamp(): TimeStamp;`

The **getModifiedTimestamp** method of the **DbFile** class returns a timestamp containing the date and time the database file was last updated.

getName

Signature `getName(): String;`

The **getName** method of the **DbFile** class returns the name of the database file, which is the file name without the **.dat** extension.

getOpenPartitions

Signature
```
getOpenPartitions(partitionList: JadeDbFilePartitionArray input;
                  maxEntries:    Integer);
```

The **getOpenPartitions** method of the **DbFile** class populates the **partitionList** parameter with references to **JadeDbFilePartition** instances; one for each open partition of the associated database file. An exception is raised if the database file is not partitioned.

If the value of the **maxInstances** parameter is non-zero, the array contains an entry for the number of open partitions specified by the **maxInstances** parameter; otherwise it contains an entry for all open partitions.

Tip Remember to remove the transient objects created by using the **getOpenPartitions** method, as shown in the following code fragment.

```
epilog
    partitionList.purge;
    delete partitionList;
end;
```

---

getPartition

Signature `getPartition(partID: Integer64): JadeDbFilePartition;`

The **getPartition** method of the **DbFile** class returns the **JadeDbFilePartition** instance associated with the partition identifier specified in the **partID** parameter.

An exception is raised if the database file is not partitioned.

getPartitionCount

Signature `getPartitionCount(): Integer64;`

The **getPartitionsCount** method of the **DbFile** class returns the number of non-removed partitions assigned to the database file.

An exception is raised if the database file is not partitioned.

getPartitionModulus

Signature `getPartitionModulus(): Integer;`

The **getPartitionModulus** method of the **DbFile** class returns the number of partitions of a database file in which new instances are stored.

An exception is raised if the database file is not partitioned.

getPartitions

Signature
```
getPartitions(partitionList: JadeDbFilePartitionArray input;
              maxEntries:    Integer);
```

The **getPartitions** method of the **DbFile** class populates the **partitionList** parameter with **JadeDbFilePartition** instances; one for each partition of the associated database file.

If the value of the **maxInstances** parameter is non-zero, the array contains an entry for the number of latest partitions specified by the **maxInstances** parameter; otherwise it contains an entry for all partitions that have not been removed.

An exception is raised if the database file is not partitioned.

Tip Remember to remove the transient objects created by using the **getPartitions** method, as shown in the following code fragment.

```
epilog
    partitionList.purge;
    delete partitionList;
end;
```

---

getPatchVersion

Signature
```
getPatchVersion(major: Integer output;
                minor: Integer output;
                build: Integer output;
                patch: Integer output);
```

The **getPatchVersion** method of the **DbFile** class returns the patch version numbers for the system files **_ jadeapp.bin**, **_ jadedef.bin**, **_sysdef.bin**, **_sysdev.bin**, **_sysgui.bin**, **_sysint.bin**, **_system.bin**, **_systools.bin**, and **_sysxrf.bin**.

The patch version numbers are assigned when a system file is built by Jade Software Corporation to address an issue with the JADE product. The patches are available for download by customers who have a support contract with Jade Software Corporation. Use this method to determine whether a patch has been applied.

Although the method can be executed against user database files (with a **.dat** extension), the values of the **major**, **minor**, **build**, and **patch** parameters that are output are always zero (**0**) for these non-system files.

The following example shows the use of the **getPatchVersion** method.

```
vars
    db          : JadeDatabaseAdmin;
    systemFiles : DbFileArray;
    file        : DbFile;
    major       : Integer;
    minor       : Integer;
    build       : Integer;
    patch       : Integer;
begin
    create db transient;
    create systemFiles transient;
    db.getDbFiles(DbFile.Kind_System, systemFiles);
    foreach file in systemFiles do
        file.getPatchVersion(major, minor, build, patch);
        write file.getName() & ": " & major.String & "." & minor.String
                             & "."  & build.String & "." & patch.String;
    endforeach;
epilog
    delete systemFiles;
    delete db;
end;
```

The following output results from executing the example method.

```
_system: 7.1.03.031
_sysxrf: 7.1.03.031
_sysgui: 7.1.03.031
_sysint: 7.1.03.031
_sysdev: 7.1.03.031
_jadeapp: 7.1.03.031
_jadedef: 7.1.03.031
_systools: 7.1.03.031
_sysdef: 7.1.03.031
```

getStatistics

Signature `getStatistics(jdo: JadeDynamicObject input);`

The **getStatistics** method of the **DbFile** class returns statistics relating to read and write operations on the persistent database file represented by the **DbFile** instance used as 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 properties returned in the **JadeDynamicObject** are listed in the following table.

Property	Description
logicalReads	The total number of read requests
logicalWrites	The total number of write requests

Property	Description
logicalReadBytes	The total accumulated size for all read requests
logicalWriteBytes	The total accumulated size for all write requests
physicalReads	The actual number of file read operations
physicalWrites	The actual number of file write operations
physicalReadBytes	The actual accumulated size for all file read operations
physicalWriteBytes	The actual accumulated size for all file write operations

The logical counts record the number and size of requests that can be serviced in cache, whereas the physical counts record actual disk activity.

The returned values include cumulative counters, which are not reset during the lifetime of the database server node. You need to compare values from one execution of the **getStatistics** method with the 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 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 following example shows the use of the **getStatistics** method.

```
showAllDbFileStats();
//display DB file statistics for all user files
vars
   dbf : DbFile;
   jdo : JadeDynamicObject;
   dba : JadeDatabaseAdmin;
   dbfiles : DbFileArray;
begin
   create dba transient;
   create dbfiles transient;
   create jdo transient;
   dba.getDbFiles(DbFile.Kind_User_Data, dbfiles);
   foreach dbf in dbfiles do
      dbf.getStatistics(jdo);
      write dbf.name & ":" & jdo.display;
   endforeach;
epilog
   delete dbfiles;
   delete dba;
   delete jdo;
end;
```

The output from the **getStatistics** method shown in the previous example is as follows.

```
MCustomers:---DatabaseFileStatistics(108)---
logicalReads = 1
logicalWrites = 1
logicalReadBytes = 119
logicalWriteBytes = 119
physicalReads = 1
physicalWrites = 1
physicalReadBytes = 4096
physicalWriteBytes = 8192
MVendors:---DatabaseFileStatistics(108)---
logicalReads = 1
logicalWrites = 1
logicalReadBytes = 119
logicalWriteBytes = 119
physicalReads = 1
physicalWrites = 1
physicalReadBytes = 4096
physicalWriteBytes = 8192
MProducts:---DatabaseFileStatistics(108)---
logicalReads = 1
logicalWrites = 1
logicalReadBytes = 119
logicalWriteBytes = 119
physicalReads = 1
physicalWrites = 1
physicalReadBytes = 4096
physicalWriteBytes = 8192
```

getTotalFileLength64

Signature `getTotalFileLength64(selector: Integer): Integer64;`

The **getTotalFileLength64** method of the **DbFile** class returns the total bytes occupied by a database map file, including partitions and Unstructured Data Resource (UDR) files.

The value of the **selector** parameter is a bitmask that defines which subfile types to include in the bytes total. One or more of the first four of the following **DbFile** class constants values, separated by the plus symbol (**+**), can be added together to give a subtotal.

DbFile Class Constant	Unpartitioned Map	Partitioned Map
GetTotLen_Base (1)	*X*.dat	*X*.dat + *X*_ndx.dat
GetTotLen_Partitions (2)	0	Sum(*X*_part*NNNNNNNNNN*.dat)
GetTotLen_SharedFileUDRs (4)	*X*_udr.dat	Sum(*X*_part*NNNNNNNNNN*_udr.dat)
GetTotLen_SingleFileUDRs (8)	Sum(*X*_udr[<oid>].dat)	Sum(*X*_part*NNNNNNNNNN*_udr[<oid>].dat)
GetTotLen_Everything (255)	Sum of all subfiles	Sum of all subfiles

Except for the **GetTotLen_Everything (255)** value, in this table the integer values in parentheses after the first four class constant names are bit positions. The *X* value represents the map file name and the *NNNNNNNNNN* value represents a partition number.

The following code fragments are examples of the **getTotalFileLength64** method.

```
write myDbFile.getTotalFileLength64(DbFile.GetTotLen_Base);
// 1 - simple case, just the .dat, or if partitioned, the
// .dat + the global index _ndx.dat

write myDbFile.getTotalFileLength64(DbFile.GetTotLen_Base +
              DbFile.GetTotLen_SharedFileUDRs);
// 1 + 4 - add in the _udr file length if the file exists
```

The following code fragment is an example of the **getTotalFileLength64** method when you have **Integer** options in your method.

```
options:= DbFile.GetTotLen_Base + DbFile.GetTotLen_SharedFileUDRs;
write dbFile.getTotalFileLength64(options);
```

getUserPatchVersion

Signature `getUserPatchVersion():Integer64;`

The **getUserPatchVersion** method of the **DbFile** class returns the unformatted current value of the version number of user data map files. You can use this method to determine whether a patch has been applied.

You can set and retrieve a version number from user schema files (for example, **_userdev.dat**) represented by the **DbFile** class **Kind_User_Schema** constant and user data files (for example, **_rootdef.dat**, **locktest.dat**, **_sindexes.dat**,and so on) represented by the **DbFile** class **Kind_User_Data** constant. This enables system administrators to optionally assign a version number to user map files.

If a user data version number is not set, this method returns zero (**0**).

isAuditing

Signature `isAuditing(): Boolean;`

The **isAuditing** method of the **DbFile** class returns **true** if auditing associated with object operations performed against the file is enabled; otherwise it returns **false**.

isFrozen

Signature `isFrozen(): Boolean;`

The **isFrozen** method of the **DbFile** class returns **true** if the associated database file has been frozen; otherwise it returns **false**. (See also the **freeze** and **thaw** methods.)

isOpen

Signature `isOpen(): Boolean;`

The **isOpen** method of the **DbFile** class returns **true** if the database file is currently open; otherwise it returns **false**.

isPartitioned

Signature `isPartitioned(): Boolean;`

The **isPartitioned** method of the **DbFile** class returns **true** if the associated database file is partitioned; otherwise it returns **false**. (See also the **freeze** and **thaw** methods.)

setPartitionModulus

Signature `setPartitionModulus(modulus: Integer);`

The **setPartitionModulus** method of the **DbFile** class specifies the number of partitions in which new instances are stored through the value the **modulus** parameter, which is in the range **1** through **1,024**. This sliding window is referred to as the creation window, as it defines the subset or window of partitions in which new objects are created. Expanding the creation window to include frozen, or offline, partitions is not allowed.

An exception is raised if the database file is locked for reorganization, the file is not partitioned, the specified partition modulus is a value outside the range of **1** through **1024**, or the new modulus value would include frozen or offline partitions in the creation window.

For more details, see "Partition Index, Modulus, and Creation Window", in Chapter 20 of the *JADE Developer's Reference*.

setPartitioned

Signature `setPartitioned(onOff: Boolean) updating;`

The **setPartitioned** method of the **DbFile** class changes the partitioned attribute of an empty database file (that is, a non-instantiated database file), as shown in the following example.

```
begin
    beginTransaction;
    Customer.getDbFile.setPartitioned(true);
    commitTransaction;
end;
```

Note Converting a non-partitioned file that contains objects to a partitioned format is accomplished using a file-based reorganization operation initiated by the JADE Database Administration utility (**jdbadmin**) **MakePartitioned** action.

An exception is raised if the database file is locked for reorganization, more than one class is mapped to the file, or the file contains objects.

thaw

Signature `thaw();`

The **thaw** method of the **DbFile** class restores the database file to its default active state. This brings the volatility of individual objects back into effect, allowing non-frozen objects to be updated or deleted. (See also the **freeze** method.)

DbFile Class Event Notifications

You can use the **JadeDatabaseAdmin** class **enableProgressEvents** or **enableByteProgressEvents** method to optionally notify operation and progress notifications for file backups. You must both enable and subscribe to this event if you want file backup operation and progress notification.

Note Enabling operation and progress notification is likely to have some impact on the elapsed time of file backups.

Subscribing to Backup Progress Events

The following examples show the **Object** class **beginNotification** method used to subscribe to backup progress events.

```
beginNotification(dbFile,
                  DbFile.BackupProgressEvent,  // progress as a percentage
                  Response_Continuous,
                  0);

beginNotification(dbFile,
                  DbFile.BackupBytesDoneEvent, // progress as num of bytes
                  Response_Continuous,
                  0);
```

Notification Event Methods

The **BackupProgressEvent** event is caused by a **DbFile** instance whenever a specified percentage increment of the file has been copied. The **userInfo** parameter of your notification callback contains the percentage amount of the file that has been copied so far. You could use a user notification method signature of the following specific form for objects that are interested only in this notification.

```
user-notification-method(eventType:   Integer;
                         obj:         DbFile;
                         eventTag:    Integer;
                         percentDone: Integer) updating;
```

In this specific form, the **obj** parameter that caused the notification is of type **DbFile** and the **userinfo** parameter is named **percentDone**, of type **Integer**.

Objects that need to handle several notification types need to use the more generic user notification method signature, as follows.

```
user-notification-method(eventType: Integer;
                         obj:       Object;
                         eventTag:  Integer;
                         userInfo:  Any) updating;
```

In this generic form, the **obj** parameter that caused the notification is of type **Object** and the **userinfo** parameter is of type **Any**. The callback method must dynamically interpret the parameter types, depending on the type of event. In the notation in these method signatures, *user-notification-method* is **userNotification** for non-form objects or **userNotify** for notifications registered by form objects.

The **BackupOperationEvent** event is caused during a backup on the **DbFile** instance being backed up to notify backup applications what operation is being performed on the file. The **RootSchema** backup and **JadeMonitorSchema** create RPS database applications that subscribe to this event and use it to update the operation text in the progress dialog.

The **userInfo** parameter passed to the notification callback is a string containing an English language description of the backup operation being performed; that is, a string containing an **operation** and **filename** pair that has the following format.

```
"operation=operation-text;fileName=filename-text"
```

You can subscribe to one of the following **DbFile** class events, represented by a class constant, during the backup of a file.

- The **BackupErrorEvent** event **userInfo** parameter passed to the notification callback is an integer error value. This provides for inclusion of error information in output streams of backup applications.
- The **BackupOutputEvent** event **userInfo** parameter passed to the notification callback is a string containing information pertinent to the backup operation for inclusion in output streams of backup applications. For example, if a partitioned file an offline partition is encountered when backing up, the **"Skipped offline file *partition-fileName-text*"** message will be received.
- The **BackupBytesDoneEvent** event **userInfo** parameter passed to the notification callback is a string containing the operation information and the progress made thus far in terms of bytes. The string has the following format, in which the *byte-count-value-text* values are formed from 64-bit file offset values.

  ```
  "operation=operation-text;fileName=filename-text; bytesDone=byte-count-value-text;bytesToDo=byte-count-value-text"
  ```

  When partitions are backed up, *filename-text* is the external file name of the partition.

  This notification is generated only if the backup application has enabled byte progress event notifications. Use the **JadeDatabaseAdmin** class **enableByteProgressEvents** and **disableByteProgressEvents** methods for this purpose. The **enableByteProgressEvents** method takes an integer increment parameter that specifies the byte value increment to be used for progress reporting (a zero value defaults to 128K bytes).

See also "JadeDatabaseAdmin Class Event Notifications", later in this chapter.

DbFileArray Class

The **DbFileArray** class is the persistent class that encapsulates behavior required to access database files in an array.

The database files are referenced by their position in the collection.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a database file array.

Inherits From: ObjectArray

Inherited By: (None)

DeadlockException Class

The **DeadlockException** class is the transient class that defines behavior for exceptions that occur because of deadlocks. The properties defined in this class are loaded from the lock operation that caused the deadlock and are equivalent to those properties of the **LockException** class.

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

For details about the **getPersistentDeadlockPriority**, **getTransientDeadlockPriority**, **setPersistentDeadlockPriority**, and **setTransientDeadlockPriority** methods that enable you to choose which process should be given a deadlock exception, see the **Process** class. See also the **DoubleDeadlockException** parameter in the [JadeServer] section of the JADE initialization file, in the *JADE Initialization File Reference*.

Inherits From: SystemException

Inherited By: (None)

DeadlockException Properties

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

Property	Contains the…
lockDuration	Duration of the lock
lockTimeout	Timeout period of the lock
lockType	Type of lock
targetLockedBy	Process that locked the object

lockDuration

Type: Integer

The read-only **lockDuration** property of the **DeadlockException** class contains the duration of the lock that was encountered in a multiuser environment.

The lock durations, provided by global constants in the **LockDurations** category, are listed in the following table.

Global Constant	Integer
Persistent_Duration (reserved for future use; that is, not yet implemented)	2
Session_Duration	1
Transaction_Duration	0

lockTimeout

Type: Integer

The read-only **lockTimeout** property of the **DeadlockException** class contains the timeout period of the lock that was encountered in a multiuser environment.

The timeout periods, provided by global constants in the **LockTimeouts** category, 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)

lockType

Type: Integer

The read-only **lockType** property of the **DeadlockException** class contains the type of lock that was encountered in a multiuser environment.

The lock types, provided by global constants in the **Locks** category, are listed in the following table.

Global Constant	Integer	Description
Not applicable	0	Result of a **getObject** operation
Share_Lock	1	Shared lock
Reserve_Lock	2	Reserve lock
Exclusive_Lock	3	Exclusive lock
Update_Lock	4	Update lock

A lock type of **4** indicates an internal resource deadlock condition, and the lock exception properties will be left as **null** values.

targetLockedBy

Type: Process

The read-only **targetLockedBy** property of the **DeadlockException** class contains a reference to the process that locked the object in a multiuser environment.

DeadlockException Methods

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

Method	Returns a reference to…
lockTarget	The target object of the deadlock
obtainedLock	An object over which this process has obtained a lock

lockTarget

Signature `lockTarget(): Object;`

The **lockTarget** method of the **DeadlockException** class returns a reference to the object that is the target of the deadlock on which an exception is raised.

The following example shows the use of the **lockTarget** method.

```
deadlockException(le: DeadlockException): 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;
```

obtainedLock

Signature `obtainedLock(): Object;`

The **obtainedLock** method of the **DeadlockException** class returns a reference to an object over which this process has obtained a lock.

A lock is being requested on this object, either directly or indirectly, by the process referenced by the **targetLockedBy** property of the **DeadlockException** class.

DecimalArray Class

The **DecimalArray** class is an ordered collection of **Decimal** values in which the values are referenced by their position in the collection.

By default, the decimal size is **23**.

Caution The scale factor in decimal arrays is currently not enforced.

Decimal arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a Decimal array.

Inherits From: Array

Inherited By: (None

Dictionary Class

The **Dictionary** class and its associated subclasses encapsulate the behavior required to store and retrieve objects in a collection by a user-defined key or keys.

Note Mapped properties (that is, properties that have a mapping method) from a **RootSchema** class cannot be used as dictionary keys. In a dictionary that allows duplicate keys, entries are inserted in *<key><oid>* order. Duplicate key entries therefore occur in object creation order within instances of the same class. If you have subclasses included in your collection, the order within the dictionary is not necessarily in strict creation order but in creation order within the instances of each subclass.

If you are using entity names (for example, a schema, class, or property name) in dictionary keys, you must consider the size of your dictionary key, as entity names can have a length of up to 100 characters and the total size of the key cannot exceed 512 key units. A key unit is a byte for any non-character data type, or one character for any character data type; that is, key sizes are string-encoding agnostic. Key sizes also must allow for a null character to terminate any strings; characters are not null-terminated.

For details about subscript operators in dictionaries, string keys in dictionary methods, and the methods defined in the **Dictionary** class, see "Using Subscripts in Dictionaries", "Using String Keys in Dictionary Methods", and "Dictionary Methods", in the following subsections.

Inherits From: Btree

Inherited By: DynaDictionary, ExtKeyDictionary, MemberKeyDictionary

Using String Keys in Dictionary Methods

Trailing 'white space' characters are trimmed from string keys when they are added to a dictionary or when string keys are passed as parameters to dictionary methods. White space characters are the space (32), horizontal tab (9), line feed (10), vertical tab (11), form feed (12), and carriage return (13) characters.

Using Subscripts in Dictionaries

The bracket (**[]**) substring operators enable you to assign values to and receive values from a dictionary. The code fragments in the following examples show the syntax of bracket subscript operators in **Dictionary** methods.

```
productDict[prodName] := prod;

prod := productDict[prodName];

customerDict["Sid Who", "12 Any Avenue", date1] := cust;

cust := customerDict["Sid Who", "12 Any Avenue", date1];
```

Dictionary Methods

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

Method	Description
createIterator	Creates an iterator for the dictionary
getAtKey	Returns an object in the receiver collection at the specified key

Method	Description
getAtKeyGeq	Returns an object in the receiver collection with a key greater than or equal to the specified key
getAtKeyGtr	Returns an object in the receiver collection with a key greater than the specified key
getAtKeyLeq	Returns an object in the receiver collection with a key equal to or less than the specified key
getAtKeyLss	Returns an object in the receiver collection with a key less than the specified key
getIteratorKeys	Retrieves the keys of a dictionary while iterating through the dictionary
includesKey	Returns **true** if the receiver contains an entry at the specified key
removeKey	Removes an item with a specified key from a dictionary
removeKeyEntry	Removes duplicated key entries from dictionaries
startKeyGeq	Sets a start position within a collection for an **Iterator** object at the object equal to or after the specified key
startKeyGtr	Sets a start position within a collection for an **Iterator** object at the next object after the specified key
startKeyLeq	Sets a start position within a collection for an **Iterator** object at the object equal to or before the specified key
startKeyLss	Sets a start position within a collection for an **Iterator** object at the object before the specified key
stringKeyCompareGeq	Returns **true** if the first string parameter is greater than or equal to the second string parameter
stringKeyCompareGtr	Returns **true** if the first string parameter is greater than to the second string parameter
stringKeyCompareLeq	Returns **true** if the first string parameter is less than or equal to the second string parameter
stringKeyCompareLss	Returns **true** if the first string parameter is less than or equal to the second string parameter

Tip Use the **startKey** methods to start or restart at a selected position in the dictionary, or to synchronize a list box or any list style view that has an associated dictionary of objects.

createIterator

Signature `createIterator(): Iterator;`

The **createIterator** method creates an iterator for the **Dictionary** class. Use an iterator associated with the dictionary to remember the current position in the dictionary. (For details about iterators, see the **Iterator** class.)

The following examples show the use of the iterator.

```
load() updating;
begin
    centreWindow;
    iter := app.myCompany.allCustomers.createIterator;
    iter.reset;
```

```
    iter.next(cust);
    if cust <> null then
       textBoxName.text    := cust.name;
       textBoxAddress.text := cust.address;
       textBoxContact.text := cust.contact;
       listBoxCustomers.listCollection(app.myCompany.allCustomers, true, 0);
       listBoxCustomers.listIndex := 1;
    else
       textBoxName.text := "No Customer Instances";
    endif;
end;

buttonFillRight_click(btn: Button input) updating;
vars
    count     : Integer;
    startTime : Time;
begin
    app.mousePointer:= self.MousePointer_HourGlass;
    startTime := app.clock.Time;
    listBoxRight.clear;
    textBoxRightStart.text := null;
    iter := app.myCompany.allProducts.createIterator;
    count := 1;
    while count <= listBoxRight.lines do
        iter.next(myProduct);
        listBoxRight.addItem(myProduct.name);
        count := count + 1;
    endwhile;
    if theArray = null then
        create self.theArray transient;
    else
        self.theArray.clear;
    endif;
    app.myCompany.allProducts.copy(self.theArray);
    listBoxScrollBar.min := 1;
    listBoxScrollBar.value := 1;
    listBoxScrollBar.max := self.theArray.size - listBoxRight.lines - 1;
    listBoxScrollBar.largeChange := (self.theArray.size/20).Integer;
epilog
    labelRight.caption :="Time Taken := " & ((app.clock.Time -
                         startTime).Integer/1000).String & " Seconds";
    app.mousePointer:= self.MousePointer_Arrow;
end;
```

getAtKey

Signature `getAtKey(keys: KeyType): MemberType;`

The **getAtKey** method of the **Dictionary** class returns a reference to an object in the receiver collection at the specified key value.

If an entry with the key specified in the **keys** parameter is not found, this method returns a **null** value.

The following examples show the use of the **getAtKey** method.

```
cust := custNameDict.getAtKey("Jones", "11 Any Road", customer);

delete() updating;
vars
    fault : Fault;
begin
    foreach fault in app.myCompany.allFaults do
        if fault.myCustomer = self and f.closedDate = null then
            fault.myCustomer := app.myCompany.allCustomers.getAtKey("Deleted
                                Customers").Customer;
        endif;
    endforeach;
end;
```

The code fragments in the following examples show the use of the bracket (**[]**) subscript operators to assign values to a dictionary.

```
cust := custNameDict["Jones", "11 Any Road", customer];

cust := custNameDict[custName];
```

getAtKeyGeq

Signature `getAtKeyGeq(keys: KeyType): MemberType;`

The **getAtKeyGeq** method of the **Dictionary** class returns a reference to an object in the receiver collection with a key equal to the value specified in the **keys** parameter.

If the specified key is not found, the next object in the dictionary after the specified key is returned. If no entry equal to or greater than the key is found, this method returns a **null** value.

Note When dealing with a descending key dictionary, the terms greater than, less than, and so on, indicate the order of the dictionary keys. For example, **C** is less than **B** for a descending alpha key and **10** is less than **5** for a descending numeric key.

The following example shows the use of the **getAtKeyGeq** method.

```
positionCollectionByKey(startKey: String;
                        collIter: Iterator io): Object;
begin
    app.myMarket.allInvestors.startKeyGeq(startKey, collIter);
    return(app.myMarket.allInvestors.getAtKeyGeq(startKey));
end;
```

getAtKeyGtr

Signature `getAtKeyGtr(keys: KeyType): MemberType;`

The **getAtKeyGtr** method of the **Dictionary** class returns a reference to an object in the receiver collection with a key greater than the value specified in the **keys** parameter. The next object in the dictionary after the specified key is returned. If no entry greater than the key is found, this method returns a **null** value.

Note When dealing with a descending key dictionary, the terms greater than, less than, and so on, indicate the order of the dictionary keys. For example, **C** is less than **B** for a descending alpha key and **10** is less than **5** for a descending numeric key.

getAtKeyLeq

Signature `getAtKeyLeq(keys: KeyType): MemberType;`

The **getAtKeyLeq** method of the **Dictionary** class returns a reference to an object in the receiver collection with a key equal to or less than the value specified in the **keys** parameter. If the specified key is not found, the object in the dictionary before the specified key is returned. If no entry less than or equal to the key is found, this method returns a **null** value.

Note When dealing with a descending key dictionary, the terms greater than, less than, and so on, indicate the order of the dictionary keys. For example, **C** is less than **B** for a descending alpha key and **10** is less than **5** for a descending numeric key.

getAtKeyLss

Signature `getAtKeyLss(keys: KeyType): MemberType;`

The **getAtKeyLss** method of the **Dictionary** class returns a reference to an object in the receiver collection with a key less than the value specified in the **keys** parameter.

The object in the dictionary before the specified key is returned. If no entry less than the key is found, this method returns a **null** value.

Note When dealing with a descending key dictionary, the terms greater than, less than, and so on, indicate the order of the dictionary keys. For example, **C** is less than **B** for a descending alpha key and **10** is less than **5** for a descending numeric key.

getIteratorKeys

Signature
```
getIteratorKeys(keys: KeyType output;
                iter: Iterator);
```

The **getIteratorKeys** method of the **Dictionary** class retrieves the keys from a dictionary while iterating through the dictionary. You can use this method 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 **getIteratorKeys** method returns values of the keys at the current position of the iterator in the associated dictionary. The **iter** parameter defines the position 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.

The method shown in the following example uses a dictionary of employees, which has two keys (**firstName** and **lastName**) that are string properties of the **Employee** class. The method assumes that the **iter** parameter is currently positioned at the required position in the dictionary (perhaps from where it left off from a previous call to this method). The **count** parameter specifies the number of entries to display. The **employeeListBox** property is a property of the receiver of type **ListBox**.

```
showEmployees(selectedEmp: Employee;
              employees:   EmpsByFirstNameLastNameDict;
```

```
                count:       Integer;
                iter:        Iterator);
    vars
        firstName : String;
        lastName  : String;
        number    : Integer;
        emp       : Employee;
    begin
        beginLoad;
        // add next count employees to list
        while(iter.next(emp)) and number < count do
            employees.getIteratorKeys(firstName, lastName, iter);
            employeeListBox.addItem(firstName & " " & lastName);
            number := number + 1;
        endwhile;
        endLoad;
    end;
```

includesKey

Signature `includesKey(keys: KeyType): Boolean;`

The **includesKey** method of the **Dictionary** class returns **true** if the receiver contains an entry at the key value specified in the **keys** parameter.

removeKey

Signature `removeKey(keys: KeyType) updating;`

The **removeKey** method of the **Dictionary** class removes an item with a specified key from a dictionary. If no entry with the value specified in the **keys** parameter key is found, an exception is raised.

The following is an example of the use of the **removeKey** method.

```
custNameDict.removeKey("Jones");
```

Note Use the **removeKeyEntry** method to remove specific duplicated key entries from a collection.

removeKeyEntry

Signature
```
removeKeyEntry(keys:  KeyType;
               value: MemberType) updating;
```

The **removeKeyEntry** method of the **Dictionary** class removes duplicated key entries from dictionaries.

Use the **value** parameter to specify the dictionary entry that is to be deleted when the key specified in the **keys** parameter is duplicated. If the specified entry for the key is not found, an exception is raised.

The following is an example of the use of the **removeKeyEntry** method.

```
wordIndex.removeKeyEntry(word, wordUsage);
```

startKeyGeq

Signature `startKeyGeq(keys: KeyType; iter: Iterator);`

The **startKeyGeq** method of the **Dictionary** class sets a start position within a collection for an **Iterator** object at the object equal to or after the key specified in the **keys** parameter. This method is used in conjunction with the **Iterator** class **next** method.

The following example shows the use of the **startKeyGeq** method.

```
listBoxScrollBar_scrolled(scroll:    ScrollBar input;
                          scrollBar: Integer) updating;
vars
    count : Integer;
begin
    listBoxRight.clear;
    textBoxRightStart.text :=
                        self.theArray[listBoxScrollBar.value].Product.name;
    app.myCompany.allProducts.startKeyGeq(textBoxRightStart.text, iter);
    while count < listBoxRight.lines and iter.next(myProduct) do
        count := count + 1;
        listBoxRight.addItem(myProduct.name);
    endwhile;
epilog
    textBoxRightStart.refresh;
end;
```

In the code fragment in the following example, the iterator remains positioned at the next employee so that the list is continued from this point when the user scrolls through the list.

```
// iter is associated with the collection allEmployees
iter := company.allEmployees.createIterator;
// start iteration from current selection
company.allEmployees.startKeyGeq(selectedEmp, "Smith",
                                 "11 Other Road", iter);
while (iter.next(emp)) and count < 10 do
// add next 10 employees to the list
    addToList(emp);
    count := count + 1;
endwhile;
```

startKeyGtr

Signature `startKeyGtr(keys: KeyType; iter: Iterator);`

The **startKeyGtr** method of the **Dictionary** class sets a start position within a collection for an **Iterator** object at the next object after the key specified in the **keys** parameter. This method is used in conjunction with the **Iterator** class **next** method.

startKeyLeq

Signature `startKeyLeq(keys: KeyType; iter: Iterator);`

The **startKeyLeq** method of the **Dictionary** class sets a start position within a collection for an **Iterator** object at the object equal to or before the key specified in the **keys** parameter. This method is used in conjunction with the **Iterator** class **back** method.

startKeyLss

Signature `startKeyLss(keys: KeyType; iter: Iterator);`

The **startKeyLss** method of the **Dictionary** class sets a start position within a collection for an **Iterator** object at the object before the key specified in the **keys** parameter. This method is used in conjunction with the **Iterator** class **back** method.

stringKeyCompareGeq

Signature
```
stringKeyCompareGeq(ord: Integer;
                    k1:  String;
                    k2:  String): Boolean;
```

The **stringKeyCompareGeq** method of the **Dictionary** class returns **true** if the first string specified in the **k1** parameter is greater than or equal to the second string specified in the **k2** parameter. Use this method for string comparisons of key values where the comparison takes into consideration the defined locale and case-sensitivity of the key (that is, when the **Latin1** locale option is selected in the **Sort Order** combo box on the **Keys** sheet of the Define Class dialog when defining the sort order).

The **stringKeyCompareGeq** method is not affected by whether the key sequence is ascending or descending.

Notes Call this method instead of directly comparing the key with string values (by using the **>=** relational binary operator) when the locale and case-sensitivity values of the key are to be taken into consideration.

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. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The parameters are listed in the following table.

Parameter	Description
ord	The ordinal value of the key (that is, **1** if it is the first or only key, **2** if it is the second key, and so on)
k1	The value of the key as a string
k2	The string value with which to compare the key value specified in the **k1** parameter

The ordinal value specified in the **ord** parameter determines the locale and case-sensitivity values to be used for the comparison. The string values specified in the **k1** and **k2** parameters specify the strings to be compared.

stringKeyCompareGtr

Signature
```
stringKeyCompareGtr(ord: Integer;
                    k1:  String;
                    k2:  String): Boolean;
```

The **stringKeyCompareGtr** method of the **Dictionary** class returns **true** if the first string specified in the **k1** parameter is greater than the second string specified in the **k2** parameter.

Use this method for string comparisons of key values where the comparison takes into consideration the defined locale and case-sensitivity of the key (that is, when the **Latin1** locale option is selected in the **Sort Order** combo box on the **Keys** sheet of the Define Class dialog when defining the sort order).

The **stringKeyCompareGtr** method is not affected by whether the key sequence is ascending or descending.

Notes Call this method instead of directly comparing the key with string values (by using the **>** relational binary operator) when the locale and case-sensitivity values of the key are to be taken into consideration.

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. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The parameters are listed in the following table.

Parameter	Description
ord	The ordinal value of the key (that is, **1** if it is the first or only key, **2** if it is the second key, and so on)
k1	The value of the key as a string
k2	The string value with which to compare the key value specified in the **k1** parameter

The ordinal value specified in the **ord** parameter determines the locale and case-sensitivity values to be used for the comparison. The string values specified in the **k1** and **k2** parameters specify the strings to be compared.

stringKeyCompareLeq

Signature
```
stringKeyCompareLeq(ord: Integer;
                    k1:  String;
                    k2:  String): Boolean;
```

The **stringKeyCompareLeq** method of the **Dictionary** class returns **true** if the first string specified in the **k1** parameter is less than or equal to the second string specified in the **k2** parameter. Use this method for string comparisons of key values where the comparison takes into consideration the defined locale and case-sensitivity of the key (that is, when the **Latin1** locale option is selected in the **Sort Order** combo box on the **Keys** sheet of the Define Class dialog when defining the sort order).

The **stringKeyCompareLeq** method is not affected by whether the key sequence is ascending or descending.

Notes Call this method instead of directly comparing the key with string values (by using the **<=** relational binary operator) when the locale and case-sensitivity values of the key are to be taken into consideration.

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. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The parameters are listed in the following table.

Parameter	Description
ord	The ordinal value of the key (that is, **1** if it is the first or only key, **2** if it is the second key, and so on)
k1	The value of the key as a string
k2	The string value with which to compare the key value specified in the **k1** parameter

The ordinal value specified in the **ord** parameter determines the locale and case-sensitivity values to be used for the comparison. The string values specified in the **k1** and **k2** parameters specify the strings to be compared.

stringKeyCompareLss

Signature
```
stringKeyCompareLss(ord: Integer;
                    k1:  String;
                    k2:  String): Boolean;
```

The **stringKeyCompareLss** method of the **Dictionary** class returns **true** if the first string specified in the **k1** parameter is less than the second string specified in the **k2** parameter.

Use this method for string comparisons of key values where the comparison takes into consideration the defined locale and case-sensitivity of the key (that is, when the **Latin1** locale option is selected in the **Sort Order** combo box on the **Keys** sheet of the Define Class dialog when defining the sort order).

The **stringKeyCompareLss** method is not affected by whether the key sequence is ascending or descending.

Notes Call this method instead of directly comparing the key with string values (by using the **<** relational binary operator) when the locale and case-sensitivity values of the key are to be taken into consideration.

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. Formatting of locale data is done on the application server, based on the locale of the corresponding presentation client.

The parameters are listed in the following table.

Parameter	Description
ord	The ordinal value of the key (that is, **1** if it is the first or only key, **2** if it is the second key, and so on)
k1	The value of the key as a string
k2	The string value with which to compare the key value specified in the **k1** parameter

The ordinal value specified in the **ord** parameter determines the locale and case-sensitivity values to be used for the comparison. The string values specified in the **k1** and **k2** parameters specify the strings to be compared.

DynaDictionary Class

The transient **DynaDictionary** class 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. In addition, the **DynaDictionary** class enables you to defer the specification of the membership and keys until run time. Dynamic dictionaries are useful in applications with requirements for:

- Ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries
- Intensive collation or collection-based sorting

  The sorting provided by a dynamic dictionary is sometimes referred to as an *insertion* sort, in which each entry is inserted in the correct place in the structure as opposed to moving the entries around to obtain the required order.

As with any collection, the size of a dynamic dictionary is limited by the maximum entries for a collection 2^32-1 (4,294,967,295) or the available disk space provided for the transient database.

Notes Dynamic dictionaries do not offer a facility to sort objects not entirely based on a comparison of embedded attribute values; for example, the ability for you to provide your own sort compare routine is not supported.

If the membership type of a **DynaDictionary** is removed by deleting the class or removing the schema, any dynamic dictionaries that have been populated with that membership class are no longer valid and attempting to use it will raise exception 1046 *(Invalid class number)*.

The **DynaDictionary** class fully supports the methods summarized in the following table that are inherited from superclasses.

Object::cloneSelf	Object::cloneSelfAs	Collection::copy	Object:: copySelf
Object::copySelfAs	Dictionary::createIterator	Collection::getOwner	Collection::indexOf
Collection::inspect	Collection::inspectModal	Collection::instantiate	Collection::isEmpty
Collection::maxSize	Collection::size		

The **DynaDictionary** class reimplements the methods summarized in the following table.

Collection::add	Collection::clear	Collection::first	Dictionary::getAtKey
Dictionary::getAtKeyGeq	Dictionary::getAtKeyGtr	Dictionary::getAtKeyLeq	Dictionary::getAtKeyLss
Collection::includes	Dictionary::includesKey	Collection::indexOf	Collection::indexOf64
Collection::last	Collection::maxSize	Collection::maxSize64	Collection::purge
ExtKeyDictionary::putAtKey	Collection::remove	Dictionary::removeKey	Dictionary::removeKeyEntry
Dictionary::startKeyGeq	Dictionary::startKeyGtr	Dictionary::startKeyLeq	Dictionary::startKeyLss

The reimplemented **Collection** class **add**, **includes**, and **remove** methods can be used only with member keys. The reimplemented **ExtKeyDictionary** class **putAtKey** method can be used only with external keys.

For details about the methods defined in the **DynaDictionary** class and usage of this class, see "DynaDictionary Methods" and "Using Dynamic Dictionaries", respectively, in the following subsections. For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Dictionary

Inherited By: (None)

DynaDictionary Methods

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

Method	Description
addExternalKey	Adds an external key definition
addExternalKeyWithSortOrder	Adds an external key definition including the sort order
addMemberKey	Adds a member key definition
addMemberKeyWithSortOrder	Adds a member key definition including the sort order
clearKeys	Clears existing key definitions
endKeys	Indicates the end of a single or multiple key definition
isValid	Returns **true** when the dynamic dictionary is fully defined
setMembership	Sets or changes the membership of a dynamic dictionary

For examples of the use of **DynaDictionary** class methods, see "Using Dynamic Dictionaries", later in this chapter.

addExternalKey

Signature

```
addExternalKey(keyType:         PrimType;
               keyLength:       Integer;
               descending:      Boolean;
               caseInsensitive: Boolean) updating;
```

The **addExternalKey** method of the **DynaDictionary** class adds an external key specification to a dynamic dictionary.

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**, and **Binary** keys, you must specify the **keyLength** parameter. This parameter is ignored for keys of other primitive types. 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.

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.

addExternalKeyWithSortOrder

Signature
```
addExternalKeyWithSortOrder(keyType:         PrimType;
                            keyLength:       Integer;
                            descending:      Boolean;
                            caseInsensitive: Boolean;
                            sortOrder:       Integer) updating;
```

The **addExternalKeyWithSortOrder** method of the **DynaDictionary** class adds an external key specification to a dynamic dictionary.

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**, and **Binary** keys, you must specify the **keyLength** parameter. This parameter is ignored for keys of other primitive types. 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 **addExternalKeyWithSortOrder** 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) updating;
```

The **addMemberKey** method of the **DynaDictionary** class adds a member key specification to a dynamic dictionary.

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.

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 Dynamic Dictionaries", later in this chapter.

addMemberKeyWithSortOrder

Signature

```
addMemberKeyWithSortOrder(propertyName:    String;
                          descending:      Boolean;
                          caseInsensitive: Boolean;
                          sortOrder:       Integer) updating;
```

The **addMemberKeyWithSortOrder** method of the **DynaDictionary** class adds a member key specification to a dynamic dictionary.

If you require multiple keys, call the **addMemberKeyWithSortOrder** 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 compare entries to determine the order of 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 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.

clearKeys

Signature

```
clearKeys() updating;
```

The **clearKeys** method of the **DynaDictionary** class clears existing key definitions so that the dictionary 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 **DynaDictionary** 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 Dynamic Dictionaries", later in this chapter.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **DynaDictionary** class returns **true** when the dynamic dictionary is fully defined; that is, after the **endKeys** method is called. When a dynamic dictionary is only partially defined, the method returns **false**.

setMembership

Signature `setMembership(type: Class) updating;`

The **setMembership** method of the **DynaDictionary** class sets the membership (that is, the base type for members) of the dynamic dictionary.

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.

Note Dynamic dictionaries can have object members only; that is, these dictionaries cannot have primitive type membership.

For an example of the use of this method, see "Using Dynamic Dictionaries", later in this chapter.

Using Dynamic Dictionaries

In the following example, a dynamic dictionary performs an object sort based on member attributes. The **publications** property is a collection of **Publications**. This example shows the use of the reimplemented **Dictionary** class **startKeyGeq** method to start the iteration at a specific point and demonstrates passing a variable list of key parameters where the keys are not known at compile-time.

```
vars
    dynaDict : DynaDictionary;
    pub      : Publication;
    iter     : Iterator;
begin
    create dynaDict transient;
    // set the membership of our dynamic dictionary
    dynaDict.setMembership(Publication);
    // specify the ytdSales, royalty, and descending pubDate dictionary keys
    dynaDict.addMemberKey("ytdSales", false, false);
    dynaDict.addMemberKey("royalty", false, false);
    // specify descending key so that most recent titles appear first
```

```
    dynaDict.addMemberKey("pubdate", true, false);
    // complete key definition
    dynaDict.endKeys(false);
    // copy publication instances into the dynamic dictionary
    publications.copy(dynaDict);
    // display all publications with more than 1000
    // sales (ytd) in sorted order
    iter := dynaDict.createIterator;
    // start the iteration where ytdSales >= 1000
    // since the dynadict has 3 keys we must pass
    // keys to the startKeyGeq method
    dynaDict.startKeyGeq(1000, null, null, iter);
    while iter.next(pub) do
        write pub.name & " " & pub.ytdSales.String &
                    pub.royalty.String & " " & pub.pubdate.String;
    endwhile;
epilog
    // ensure we delete transients
    delete iter;
    delete dynaDict;
end;
```

The following example shows the use of a dynamic dictionary that orders employees by age and length of service. These key values are returned by the **getAge** and **getLengthOfService** methods in the **Employee** class. As the key values are not properties, the dynamic dictionary cannot be defined using member keys and can only be defined using external keys.

```
vars
    dyna : DynaDictionary;
    emp : Employee;
    root : Root;
    iter : Iterator;
begin
    create dyna transient;
    dyna.setMembership(Employee);
    dyna.addExternalKey(TimeStampInterval,8,false,false); // age
    dyna.addExternalKey(TimeStampInterval,8,false,false); // length service
    dyna.endKeys(false);                                  // no duplicates
    root := Root.firstInstance;
    foreach emp in root.allEmployeesByName do
        dyna.putAtKey(emp.getAge, emp.getLengthOfService, emp);
    endforeach;
    iter := dyna.createIterator();
    while iter.next(emp) do
        write emp.name & Tab & emp.dob.shortFormat;
    endwhile;
epilog
    delete dyna;
    delete iter;
end;
```

The following examples demonstrate the use of a dynamic dictionary in an application-specific query where a suitable dictionary type is not available in the object model. These examples show the use of a key path specification and the bracket (**[]**) substring operators to perform a dictionary lookup and use the model **Department**, **Employee**, and **DepartmentSet** (a set of **Department**) classes, with the following partial definitions.

```
Employee
(
referenceDefinitions
    department:     Department explicitInverse, readOnly;
)
Department
(
referenceDefinitions
    manager:        Employee explicitInverse, readOnly;
)
```

The following **AQController** class is a query controller, which has a singleton transient instance at run time. This class defines the **managers** exclusive property of type **DynaDictionary**.

```
AQController
(
referenceDefinitions
    managers : DynaDictionary implicitMemberInverse, protected;
jadeMethodDefinitions
    // public interface operations
    getManagerByDeptName(deptName: String): Manager;
    // implementation method
    loadManagers(departments: DepartmentSet) protected;
)
```

The following method is called when an **AQController** instance is initialized and sets up and then populates the **managers** dynamic dictionary. You could use a notification mechanism to ensure that the **managers** dictionary is kept current when departments are added or deleted or managers are changed.

```
AQController::loadManagers
loadManagers(departments: DepartmentSet) protected;
vars
    dept : Department;
begin
    // set the membership of our dynamic dictionary
    managers.setMembership(Employee);
    // the single key is the keypath: department.name
    managers.addMemberKey("department.name", false, false);
    // end key specification, and do not allow duplicates as the
    // model does not allow two departments with the same name
    managers.endKeys(false);
    foreach dept in departments do
        managers.add(dept.manager);
    endforeach;
end;
```

The following method looks up the **managers** dynamic dictionary to find the manager by department name.

```
AQController::getManagerByDeptName
getManagerByDeptName(deptName: String): Manager;
begin
```

```
    return managers[deptName].Manager;
end;
```

Exception Class

The **Exception** class defines the protocol for raising and responding to exception conditions. The specific kinds of exceptions that can occur are defined by the subclasses of exception, as follows.

- Fatal errors
- Normal exceptions
  - JADE (system) exceptions (integrity violations and lock exceptions)
  - File and connectivity exceptions
  - User interface exceptions
  - Notification exceptions

In JADE thin client mode, exception dialogs are always displayed on the presentation client. The following is an example of the definition of an exception handler method.

```
handleException(exObj: Exception): Integer;
constants
    StringTooLong = 1035;
begin
    if exObj.errorCode = StringTooLong then
        app.msgBox("The picture is too large:", "Error", 0);
        return Ex_Abort_Action;
    else
        return Ex_Pass_Back;    // default dialog
    endif;
end;
```

For details about the returned values from exceptions and the properties and methods defined in the **Exception** class, see "Exception Class Return Values", "Exception Properties", and "Exception Methods", in the following subsections. For a list of the global constants that you can use in your exception handlers, if required, see the **JadeErrorCodesDatabase**, **JadeErrorCodesSDS**, and **JadeErrorCodesWebService** global constants categories in Appendix A of the *JADE Encyclopaedia of Primitive Types*.

Inherits From: Object

Inherited By: FatalError, NormalException

Exception Class Return Values

The following table lists the **Exception** class return values that indicate the action the system takes.

Return Value	Global Constant	Action
0	Ex_Continue	Resumes execution from the next expression after the expression that caused the exception. Use this return mode only in circumstances when you are certain that continuing the code execution will still be correct after ignoring the exception.

Return Value	Global Constant	Action
		For lock exceptions, use this return mode *only* if the lock has been successfully retried. If the exception occurred while updating, ensure that the transaction has not been aborted by the exception handler. Continuable exceptions can be ignored (that is, the **Ignore** button is enabled) when production mode is disabled. The **Ignore** button is always disabled when production mode is enabled for the database. (For details, see "Running JADE Production Mode Databases", in Chapter 1 of the *JADE Runtime Application Guide*.)
1	Ex_Abort_Action	Causes the currently executing methods to be aborted. The execution stack is stripped back and the application reverts to an idle state in which it is waiting for user input or some other Windows event, in most cases. If there is a transaction in progress, this is not aborted. An **abortTransaction** instruction must be explicitly coded within the exception handler if the database transaction in progress is also to be aborted.
2	Ex_Resume_Next	Passes control back to the method that armed the exception handler. Resumes from the next statement after the evaluation of the method call or expression in which the exception occurred. You cannot resume from global exception handlers. Using this value for a global exception handler is equivalent to returning the **Ex_Abort_Action** value. **Note** If an exception occurs during evaluation of a return expression, when the exception handler returns, the executing method is terminated (in accordance with executing a return instruction) and the default value of the return type is returned.
-1	Ex_Pass_Back	Passes control back to the prior local exception handler for this type of exception or if a local handler is not found, a global exception handler for this type of exception.

When you have created an exception handler, you must arm it or tell the system to invoke that method in case of an exception. The following syntax sets the current exception handler to *method-call-expression*.

```
on exception-class do method-call-expression
```

The *exception-class* identifier is the **Exception** class or one of its subclasses.

Note If a global exception handler is armed on a **serverExecution** method and returns **Ex_Abort_Action** (or **Ex_Resume_Next**) when an exception occurs, exception 1242 (that is, a method executing in another node was aborted) is raised on the client node.

The following is an example of a method that arms an exception handler.

```
work(currentObject: Object);
vars
    fileSave : CMDFileSave;
    file     : File;
```

```
begin
    on FileException do fileExceptionHandler(exception);
    if fileSave.open = 0 then
        create file;
        file.mode        := File.Mode_Output;
        file.allowCreate := false;
        file.fileName    := fileSave.fileName;
        currentObject.saveToFile(file);
        delete file;
    endif;
end;
```

Exception Properties

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

Property	Description
category	Contains the category of exception within an **Exception** subclass
continuable	Specifies if execution can be continued after the exception has been handled
currentMethodDesc	Contains a reference to the current **MethodCallDesc** object
errorCode	Identifies an exception within a class of exceptions
errorItem	Contains additional information about the exception
extendedErrorText	Contains diagnostic text of an error or warning message
helpBook	Windows help file that contains the explanation of the exception
kind	Contains the kind of exception that is raised
level	Level number of the exception
remoteErrorCode	Identifies an exception that occurred while executing a method on another node
reportingMethodDesc	Contains a reference to the method that reported or raised the exception
resumable	Specifies if execution can be resumed after the exception has been handled

category

Type: Integer

The **category** property of the **Exception** class contains the category of exception within an **Exception** subclass.

continuable

Type: Boolean

The **continuable** property of the **Exception** class specifies whether execution can be continued after the exception has been handled. This property is set to **false** by default for both system and user exceptions.

When set, this property is displayed in the default exception dialog as **continuable: Yes**.

By default, no system exception can be continued (unless it is a lock exception). For your user exceptions, it is your responsibility to set this property to perform the appropriate action when you create your object.

Trying to continue a non-continuable exception causes a further exception (that is, *1238 - exception handler invalid return code*) to be raised. If this exception is caught by an exception handler that then tries to continue the exception, exception 1239 (*nested exceptions limit exceeded*) is eventually raised, due to repeated 1238 exceptions. Your exception handlers should therefore test to see if an exception is continuable or not before attempting to return **Ex_Continue**. Your exception handler should also include checks to see if it is in a nested exception situation. It may also be beneficial to specifically check for the 1238 exception.

If an exception handler that is handling an exception with this property set to **false** returns an **Ex_Continue** value, a dialog is displayed that advises you that the exception cannot be continued, and JADE forces the action to abort.

When production mode is disabled, continuable exceptions can be ignored (that is, the **Ignore** button is enabled). The **Ignore** button is always disabled when production mode is enabled for the database. For details about production mode, see "Running JADE Production Mode Databases", in Chapter 1 of the *JADE Runtime Application Guide*.

currentMethodDesc

Type: MethodCallDesc

The read-only **currentMethodDesc** property of the **Exception** class contains a reference to the current **MethodCallDesc** object. Every time a method calls another method, a **MethodCallDesc** object is created in a stack. Each new **MethodCallDesc** object has a reference to the previous **MethodCallDesc** object in the stack.

Use the **currentStack** method of the **Process** class to obtain the call stack for the current process.

errorCode

Type: Integer

The **errorCode** property of the **Exception** class contains a number that uniquely identifies an exception within a class of exceptions.

Tip As JADE itself uses exception codes with lower numbers (that is, numbers less than 63,999), you should define error codes for your user-defined exceptions in the range 64,000 through **Max_Integer** (#7FFFFFFF, which equates to 2,147,483,647).

The following examples show the use of the **errorCode** property.

```
invalidObjectExceptionHandler(exObj: Exception; referencedObj: Object):
                              Integer updating, protected;
// ------------------------------------------------------------------------
// Exception handler to reset list box and collection for object not found
// and object deleted exceptions.  exObj is the exception object and
// referencedObj is the object to check against the error object.
// ------------------------------------------------------------------------
constants
    ObjectNotFound = 4;
    ObjectDeleted  = 1072;
begin
    if (exObj.errorCode = ObjectNotFound or exObj.errorCode = ObjectDeleted)
                and
        exObj.errorObject = referencedObj
    then
    // If exception is for the object we're checking for, reset ourselves
        abortTransaction;
        setCollection(null, false);
```

```
        return Ex_Abort_Action;
    endif;
    // Otherwise pass control to the next exception handler
    return Ex_Pass_Back;
end;

handleException(exObj: Exception): Integer;
constants
    StringTooLong = 1035;
begin
    if exObj.errorCode = StringTooLong then
        app.msgBox("The picture is too large:", "Error", 0);
        return Ex_Abort_Action;
    else
        return Ex_Pass_Back;                // default dialog
    endif;
end;
```

This property is displayed in the default exception dialog as **error code**. For details, see "Error Messages and System Messages", in the **JADEMsgs.pdf** file.

errorItem

Type: String[539]

The **errorItem** property of the **Exception** class is optionally set by the module that raised the exception and normally contains additional information about the exception; for example, an invalid property exception would contain the name of the property in error. This property is displayed in the default exception dialog as **error item**.

The maximum length of the **errorItem** property is **539** characters. If you exceed this maximum, an exception is raised.

extendedErrorText

Type: String

The **extendedErrorText** property of the **Exception** class contains an extended error description for exception instances recorded by any services that raise exceptions; for example, ODBC.

The default exception handler dialog appends this extended information to the error text in the error description text box. If the exception is an ODBC exception, the diagnostic message must explain if the source of an error or warning is an ODBC component itself. The text of messages therefore has two formats. For errors and warnings that do not occur in a data source, the diagnostic message uses the following format.

```
[vendor-id][ODBC-component-id]component-supplied-text
```

For errors and warnings that occur in a data source, the diagnostic message uses the following format.

```
[vendor-id][ODBC-component-id][data-source-id]data-source supplied-text
```

The following example shows a diagnostic message for an error that occurred in an ODBC SQL server.

```
[Microsoft][ODBC SQL Server Driver]Connection is busy with
                                   results for another hstmt
```

The **extendedErrorText** property can also contain the value of a decimal before it is truncated. (For details about specifying whether an exception is raised when a decimal overflow occurs and an example of the use of the **extendedErrorText** property in an exception handler, see the **Process** class **truncateOnDecimalOverflow** method.)

helpBook

Type: String[12]

The **helpBook** property of the **Exception** class contains the Windows help file that contains the explanation of the exception.

This file is opened when you select the **Help** button in the default exception dialog.

kind

Type: Character[1]

The **kind** property of the **Exception** class contains the kind of exception that is raised, as listed in the following table.

Character	Exception raised by ...	Description
0	Precondition violation	Exception of the calling method; for example, incorrect parameters or a condition not met
1	Internal exception	Exception of the receiver
2	Post-condition violation	Not yet implemented; reserved for future use

level

Type: Integer

The read-only **level** property of the **Exception** class contains the level number of the exception. The level number is automatically incremented for each nested exception.

remoteErrorCode

Type: Integer

The read-only **remoteErrorCode** property of the **Exception** class contains a number that uniquely identifies an exception that occurred while executing a method in another node, typically the server node.

This property is available only when the **errorCode** property value is **1242** (*A method executing in another node was aborted*).

reportingMethodDesc

Type: MethodCallDesc

The **reportingMethodDesc** property of the **Exception** class contains a reference to a **MethodCallDesc** instance that describes the method that reported or raised the exception.

resumable

Type: Boolean

The **resumable** property of the **Exception** class specifies that execution can be resumed after the exception has been handled when the value is set to **true**. For system exceptions, this property is set to **true** by default. For fatal errors, this property is set to **false**.

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

```
vars
    ex : UserException;
begin
    // Creates an object of the UserException Class and defines the
    // properties for this object.  The exception is then raised.
    create ex;
    ex.errorCode   := 64000;
    ex.continuable := true;
    ex.resumable   := true;
    raise ex;
end;
```

Note You cannot resume from a global exception handler. Using the **Ex_Resume_Next** value for a global exception handler is equivalent to returning the **Ex_Abort_Action** value.

Your exception handling code could check for this situation before it tries to resume an exception; for example:

```
...  // exception handler
if exception.resumable then
    return Ex_Resume_Next;
endif;
...
```

For more information about exception handling, see "Handling Exceptions", in Chapter 3 of the *JADE Developer's Reference*.

Exception Methods

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

Method	Description
createSOAPMessage	Returns a string representing a SOAP fault message
debug	Displays current process stack information, and enables you to inspect variables
defaultHandler	Calls the **showDialog** method to display the default exception dialog
errorObject	Returns the object reference in error
initializationHandler	Placeholder for a user-defined exception handler for use during the JADE initialization process
logExceptionHistory	Logs the exception stack history of each nested exception
logProcessHistory	Logs the call stack history
logSelf	Appends a description of the exception object to a file

Method	Description
setErrorObject	Saves a reference to the error object in the exception instance
showDialog	Displays the default exception dialog
text	Returns the error text associated with the **errorCode** property exception

createSOAPMessage

Signature `createSOAPMessage(): String;`

The **createSOAPMessage** method of the **Exception** class returns a string representing a SOAP fault message. The receiver is the exception that was raised.

For more details, see "Using Communications Protocols Other than HTTP in your Web Service", in Chapter 11 of the *JADE Developer's Reference*.

debug

Signature `debug();`

The **debug** method of the **Exception** 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.

For details, see "Debugging the Method Call Stack", in Chapter 3 of the *JADE Developer's Reference*.

defaultHandler

Signature `defaultHandler(): Integer;`

The **defaultHandler** method of the **Exception** class is the "handler of the last resort" that is automatically invoked on the exception instance if no other exception handler consumes the exception. The exception, including the method call stack history and the exception stack history, is written to the log file of the current application (for example, **MyApp.log**).

In client-side GUI applications, the **defaultHandler** method calls the **Exception** class **showDialog** method on **self**.

If the **showDialog** method returns **true**, the **defaultHandler** method ignores the exception by returning the result of **Ex_Continue**. (Note that it is not valid to ignore non-continuable exceptions.) However, if the **showDialog** method returns **false**, the **defaultHandler** method first aborts any current persistent or transient transaction and then terminates the current action by returning an **Ex_Abort_Action** result.

When the **defaultHandler** method is invoked for an exception raised in non-GUI-capable methods (including server methods, server application methods, and any non-GUI application methods), the **defaultHandler** method does not call the **showDialog** method but instead aborts any current persistent or transient transaction, logs the exception to the exception log file of the current application, and when invoked from a server method it returns **Ex_Pass_Back**; otherwise, it returns **Ex_Abort_Action**.

Note You can reimplement this method in your subclasses of the **Exception** class, if you want different default behavior. Your exception handlers that do not handle a specific exception should return **Ex_Pass_Back** rather than directly calling the **defaultHandler** method.

For details about the default exception handler and the values that are returned by JADE, see "Handling Exceptions" and "Creating an Exception Handler", respectively, in Chapter 3 of the *JADE Developer's Reference*.

errorObject

Signature `errorObject(): Object;`

The **errorObject** method of the **Exception** class returns a reference to the object in error if this is relevant to the exception, otherwise it returns **null**. For example, the *Object not found* and *Object deleted* system exceptions return a reference to the object that is not found or is deleted, respectively.

initializationHandler

Signature `initializationHandler(): Integer;`

The **initializationHandler** method of the **Exception** class is the placeholder for your user-defined default handler for exceptions raised during the initialization of your client node.

You can specify your own initialization handler and specify the name of its library file in the **InitializationHandlerLibrary** parameter in the [JadeClient] section of the JADE initialization file. If you do not specify your own initialization handler, the default handler is called, aborting the action.

logExceptionHistory

Signature `logExceptionHistory(logFileName: String);`

The **logExceptionHistory** method of the **Exception** class enables you to log the exception stack history in an exception handler.

The exception stack contains an entry for each nested exception. The output is appended to the file specified in the **logFileName** parameter.

logProcessHistory

Signature `logProcessHistory(logFileName: String);`

The **logProcessHistory** method of the **Exception** class enables you to log the call stack history in an exception handler.

The output is appended to the file specified in the **logFileName** parameter.

logSelf

Signature `logSelf(logFileName: String);`

The **logSelf** method of the **Exception** class appends a description of the exception object to the file specified in the **logFileName** parameter.

The output is similar to the exception information logged by the JADE default exception handler.

setErrorObject

Signature `setErrorObject(obj: Object);`

The **setErrorObject** method of the **Exception** class saves a reference to the error object in the exception instance. Use this method to report the object in error when a user raises an exception.

showDialog

Signature `showDialog(): Boolean;`

The **showDialog** method of the **Exception** class displays the default exception dialog, which provides details of the current exception and buttons that enable you to:

- Abort, which aborts the action, effectively terminating the code that was executing at the time of the exception.
- Debug, which displays a Call Stack Browser that enables you to browse methods in the call stack and to inspect parameters and local variables.
- Ignore (continue), which ignores the exception and continues execution from the next expression after the expression that caused the exception. (The **Ignore** button is enabled only if the exception is continuable and the system is not in production mode.)

The exception dialog can be closed by clicking the **Abort** button or the **Ignore** button (if the exception is continuable). If the exception can be ignored and the user clicks the **Ignore** button, the **showDialog** method returns **true**. If the user clicks the **Abort** button, the **showDialog** method returns **false**.

For more details, see "Default Exception Handler", in Chapter 3 of the *JADE Developer's Reference*.

Note This method can be called from a user-defined exception handler and it can be called by the **defaultHandler** method. (For more details, see the **Exception** class **defaultHandler** method, earlier in this section.) See also "Handling Exceptions", in Chapter 3 of the *JADE Developer's Reference*.

text

Signature `text(): String;`

The **text** method of the **Exception** class returns the error text in English corresponding to the **errorCode** of the exception. This error text is displayed in the default exception dialog as the **description**. The text is obtained from the **jadmsgs.eng** file in the JADE **bin** directory by finding a line that begins with the **errorCode** number and then returning the text that follows it.

You can use a file other than **jadmsgs.eng** by setting the **Language** parameter in the [JadeClient] section of the JADE initialization file. For more details, see the *JADE Initialization File Reference*.

The code fragment in the following example shows the use of the **text** method.

```
if exc.errorCode = 1310 then   // Key already in dictionary
    abortTransaction;
    app.msgBox(exc.text, self.name, MsgBox_OK_Only +
         MsgBox_Exclamation_Mark_Icon);
    return Ex_Abort_Action;
endif;
```

ExceptionHandlerDesc Class

The **ExceptionHandlerDesc** class represents exception handlers that have been armed globally or locally by the current process in the current node.

You can populate an array of transient instances of the **ExceptionHandlerDesc** class by executing the **getExceptionHandlerStack** method of the **Process** class. The array of **ExceptionHandlerDesc** objects represents the exception handler stack for the process on the current node. For more details, see "Viewing the Exception Handler Stack", in Chapter 3 of the *JADE Developer's Reference*.

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

Inherits From: Object

Inherited By: (None)

ExceptionHandlerDesc Properties

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

Property	Description
armingApplication	Application in effect when the exception handler was armed.
armingMethod	Method in which the exception handler was armed.
exceptionClass	Exception class or subclass for which the exception handler is armed.
exceptionHandlerMethod	Exception handler method that is armed.
exceptionHandlerReceiver	Receiving object upon which the exception handler method will be called.
invocationCount	Current number of active invocations of the exception handler. Valid only when the exception handler is in the current call stack.
isGlobal	**true** if the exception handler is armed globally or **false** if armed locally.

armingApplication

Type: Application

The **armingApplication** property of the **ExceptionHandlerDesc** class contains a reference to the application that was in effect when the exception handler was armed. This is usually the application that is currently executing, although if a method from an imported package or peer schema is executing, it is the application object associated with that package or peer schema.

armingMethod

Type: Method

The **armingMethod** property of the **ExceptionHandlerDesc** class contains a reference to the method in which the local exception handler was armed.

If an exception handler is globally armed, the **armingMethod** property contains a **null** value.

exceptionClass

Type: Class

The **exceptionClass** property of the **ExceptionHandlerDesc** class contains a reference to the specific **Exception** class or subclass for which the exception handler is armed. For example, if an exception handler is armed by the following instruction, the **exceptionClass** property contains a reference to the **FileException** class.

```
on FileException do agent.handleDuplicates(exception);
```

exceptionHandlerMethod

Type: Method

The **exceptionHandlerMethod** property of the **ExceptionHandlerDesc** class contains a reference to the exception handler method that is armed. For example, if an exception handler is armed by the following instruction, the **exceptionHandlerMethod** property contains a reference to the **handleDuplicates** method.

```
on Exception do agent.handleDuplicates(exception);
```

exceptionHandlerReceiver

Type: Object

The **exceptionHandlerReceiver** property of the **ExceptionHandlerDesc** class contains a reference to the object that is to receive the call to execute the exception handler method in the event of an exception. For example, if an exception handler is armed by the following instruction, the value of the **exceptionHandlerReceiver** property is the **agent** object.

```
on Exception do agent.handleDuplicates(exception);
```

invocationCount

Type: Integer

The **invocationCount** property the **ExceptionHandlerDesc** class contains the number of times the exception handler is currently invoked. This is usually zero (**0**) or **1**, unless the **getExceptionHandlerStack** method is called when nested exceptions have occurred, in which case the exception handler method could have been invoked more than once.

isGlobal

Type: Boolean

The **isGlobal** property of the **ExceptionHandlerDesc** class has a value of **true** if the exception handler is armed globally.

It has a value of **false** if the exception handler is armed locally.

ExternalArray Class

The **ExternalArray** class encapsulates the behavior of an ordered virtual collection that represents the rows in a result set generated from an SQL query containing a sort specification; that is, the **ORDER BY** clause. Instances of this class occur in the order determined by the **ORDER BY** clause.

For details about external array subscripts, see "Using Subscripts in External Arrays", in the following subsection.

Inherits From: ExternalCollection

Inherited By: (None)

Using Subscripts in External Arrays

The bracket (**[]**) substring operators enable you to access rows at a specified position in a result set.

The following example of the **ExternalArray** class shows the access of the eighth transaction in an external array called transactions.

```
transaction := transactions[8];
```

The bracket notation (**[]**) is a syntax shortcut for the **at** method of the **ExternalCollection** class.

ExternalCollection Class

The **ExternalCollection** class provides the common protocol for external collection classes, which represent the result set of a selection from an external data source.

External collections are virtual, in the sense that member instances do not exist until they are first referenced. When a member is first referenced by a direct key access or by an iteration, an external proxy instance is created, which represents the corresponding row in the result set. (Proxy classes act as the mediators between JADE and the external relational database, and are derived from the **ExternalObject** class.)

External collections provide operations for direct and relative key access, and may be used in collaboration with external iterators to access rows in a result set. External collections are read-only; that is, operations such as add, remove, clear, and purge are not supported. The JADE compiler prevents the use of updating methods for external collections, in the same way that they are prevented for automatic collections participating in an inverse relationship.

There is a corresponding external collection for each of the standard JADE collection types listed in the following table.

Collection Type	Description
ExternalCollection	Abstract superclass of all external collections
ExternalArray	Ordered collection that requires an **ORDER BY** clause sort specification and an optional **WHERE** clause filtering specification
ExternalDictionary	Ordered collection that requires keys, an **ORDER BY** clause sort specification, and an optional **WHERE** clause filtering specification
ExternalSet	Unordered collection that can have an optional **WHERE** clause filtering expression

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

Inherits From: Collection

Inherited By: ExternalArray, ExternalDictionary, ExternalSet

ExternalCollection Properties

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

Property	Description
database	Contains a reference to the external database instance
filterExpression	Contains the filter that enables you to select specific rows
sortExpression	Contains the expression that controls how instances in the collection are ordered

database

Type: ExternalDatabase

The **database** property of the **ExternalCollection** class contains a reference to the external database instance.

filterExpression

Type: String

The **filterExpression** property of the **ExternalCollection** class contains the string expression used to override the default filtering or the **WHERE** predicate defined for an external collection. Use this property to select a subset of records at run time.

The filtering expression should not contain the **WHERE** keyword, and it must be defined in terms of external column names and not the attribute names to which they are mapped. If the resultant SQL statement is not valid, an ODBC exception is raised.

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

```
displayExtDBCustomers();
vars
    custs : CustomersByCityDict;
    cust  : Customers;
begin
    create custs;
    custs.filterExpression := "Customers.city = 'London' ";
    foreach cust in custs do
        write cust.contactName & " " & cust.city;
    endforeach;
epilog
    delete custs;
end;
```

sortExpression

Type: String

The **sortExpression** property of the **ExternalCollection** class contains the string expression used to override the default sort or SQL **ORDER BY** specification defined for an external collection. The sort expression should not contain the **ORDER BY** SQL keywords. The filtering expression (**filterExpression**) must be defined in terms of external column names and not the attribute names to which they are mapped. If the resultant SQL statement is not valid, an ODBC exception is raised.

The following example shows the use of the **sortExpression** and **filterExpression** properties to create a shared external collection instance, set the filter and sort expressions, and then use a **foreach** instruction to fetch the instances. (Alternatively, you could use an iterator to fetch the instances.)

```
findRichCustomers();
    accounts : CustomerAccountDict;
    account  : Account;
begin
    create accounts;
    accounts.filterExpression := "account.balance > 100000";
    accounts.sortExpression   := "account.balance";
    foreach account in accounts do
        display(account.number, account.name);
    endforeach;
epilog
    delete accounts;
end;
```

ExternalCollection Methods

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

Method	Description
at	Returns the entry at a specified index in the collection
canCreate	Returns **true** if member type instances can be created
createIterator	Creates the external iterator for an external collection
createObject	Creates a new instance of the external object
first	Returns the first entry in the collection
getSQL	Returns the SQL statement of the receiver
includes	Returns **true** if the collection contains the specified object
last	Returns the last entry in the collection
maxSize	Returns the maximum number of entries that the external collection can contain
maxSize64	Returns the maximum number of entries that the external collection can contain as an **Integer64** value
size	Returns the current number of entries in the external collection
size64	Returns the current number of entries in the external collection as an **Integer64** value

at

Signature `at(index: integer): MemberType;`

The **at** method of the **ExternalCollection** class returns a reference to the entry in the collection specified by the **index** parameter. This position corresponds to accessing a specified row in the result set.

If there is no row at the specified index in the result set, an exception is raised.

canCreate

Signature `canCreate(): Boolean;`

The **canCreate** method of the **ExternalCollection** class returns **true** if instances of the member-type of the external collection can be created.

This method returns **false** if the data-source is read-only or the class of the external collection members is read-only. An external class is read-only if it is based on a relational view or a **join** query defined by the External Schema Wizard.

createIterator

Signature `createIterator(): Iterator;`

The **createIterator** method of the **ExternalCollection** class creates a reference to an external iterator for use with external collections.

Use an iterator associated with an array to remember the current position in the external array. (For details about iterators, see the **Iterator** class.)

createObject

Signature `createObject(): MemberType;`

The **createObject** method of the **ExternalCollection** class creates a new instance of the external object. Use this method to create a row in an external database table, if required.

This method enables you to perform a *cursor-based*, or positioned, creation update of the current proxy object in the external collection.

first

Signature `first(): MemberType;`

The **first** method of the **ExternalCollection** class returns a proxy reference that represents the first entry in the virtual collection.

For ordered external collections such as dictionaries and arrays, the proxy represents the first row selected, determined by the **ORDER BY** clause.

getSQL

Signature `getSQL(): String;`

The **getSQL** method of the **ExternalCollection** class returns a string containing the SQL statement of the receiver.

includes

Signature `includes(value: MemberType): Boolean;`

The **includes** method of the **ExternalCollection** class returns **true** if the virtual collection or result set contains the object specified in the **value** parameter. This method results in a key-equal query, based on the attributes that comprise the primary keys in the proxy.

last

Signature `last(): MemberType;`

The **last** method of the **ExternalCollection** class returns a proxy reference that represents the last entry in the virtual collection.

For ordered external collections such as dictionaries and arrays, the proxy represents the last row selected, determined by the **ORDER BY** clause.

maxSize

Signature `maxSize(): Integer;`

The **maxSize** method of the **ExternalCollection** class returns the maximum number of entries that an external collection can contain.

Note Use the **maxSize64** method instead of the **maxSize** method as the number of entries in the collection exceeds the maximum integer value of 2,147,483,647.

---

maxSize64

Signature `maxSize64(): Integer64;`

The **maxSize64** method of the **ExternalCollection** class returns the maximum number of entries that an external collection can contain as an **Integer64** value.

size

Signature `size(): Integer;`

The **size** method of the **ExternalCollection** class returns the number of entries in the virtual collection.

Caution As this method results in an SQL query that counts the rows in the selected tables mapped to the proxy class, you should use this method with caution if you expect a large number of rows in the result set.

Use the **size64** method instead of the **size** method, if the number of entries in the collection could exceed the maximum integer value of 2,147,483,647.

size64

Signature `size64(): Integer64;`

The **size64** method of the **ExternalCollection** class returns the number of entries in an external collection as an **Integer64** value.

ExternalDatabase Class

The **ExternalDatabase** class encapsulates the behavior required to access entries in an external database. This class represents a connection to an external database and provides methods that operate on the data source.

External databases cannot be passed across nodes. The **open** and **close** method calls or any access of an **ExternalObject** instance in the external database must occur on the same node.

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

Inherits From: Object

Inherited By: (None)

ExternalDatabase Properties

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

Property	Description
connectionString	Contains parameters required to connect to a data source
name	Contains the name of the external database
password	Contains the password required by the data source
serverName	Contains the name of the server defined for the data source
userName	Contains a user id used to establish a connection

connectionString

Type: String

The **connectionString** property of the **ExternalDatabase** class contains any parameters required for connecting to a data source. These parameters are generally specific to the driver or they are data source-specific.

A default connection string is generated automatically when connecting to a data source using the External Schema Wizard browse facility.

Use this property to override the default connection string at run time on a user or connection basis, if required. The connection string should not include the user id (UID) and password (PWD) parameters.

name

Type: String[100]

The **name** property of the **ExternalDatabase** class contains the name of the external database.

password

Type: String[128]

The **password** property of the **ExternalDatabase** class contains a password, if it is required by the data source. This property is used for authentication in conjunction with **userName** property at the data source, if required.

A default password can be stored on the database object if the schema translator has allowed this. You can change the default password at run time for each user, before opening a database connection.

serverName

Type: String[128]

The **serverName** property of the **ExternalDatabase** class contains the name of the database server if this is applicable for the data source.

userName

Type: String[128]

The **userName** property of the **ExternalDatabase** class contains the name of a valid user id, which is used for authentication at the data source. A default user id is established at design time, by using the External Schema Wizard. You can change the default user name at run time for each user before opening a database connection.

ExternalDatabase Methods

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

Method	Description
abortExternalTransaction	Rolls back the changes made during the current transaction
beginExternalTransaction	Starts a database transaction
canTransact	Returns **true** if the external database supports transactions
close	Closes the connection to an external database
commitExternalTransaction	Commits a transaction
executeSQL	Directly executes an SQL statement
getFileDSN	Returns the file data source name
getLastError	Returns the last ODBC exception when the **isSQLValid** method returns **false**
getMachineDSN	Returns the machine data source name
importStoredProcedures	For internal use only
isOpen	Returns **true** if the external database is currently open
isSQLValid	Checks the syntax of an SQL statement
isUpdatable	Returns **true** if the external database can be updated
loadProcedure	Reserved for future use
open	Opens a connection to an external database
setFileDSN	Programmatically sets the file data source name
setMachineDSN	Programmatically sets the machine data source name

abortExternalTransaction

Signature `abortExternalTransaction() updating;`

The **abortExternalTransaction** method of the **ExternalDatabase** class rolls back the changes made during the current transaction. If the external database supports transactions, use this method to undo the effects of a transaction, if required. All updating operations (creates, deletes, or updates) made since the last **beginExternalTransaction** call are reversed to the state that existed at the time of that call.

If the external database does not support transactions (use the **canTransact** method to determine this), calling the **abortExternalTransaction** method has no effect.

beginExternalTransaction

Signature `beginExternalTransaction() updating;`

The **beginExternalTransaction** method of the **ExternalDatabase** class starts an external database transaction.

If the external database supports transactions, call this method at the start of a series of updating operations (creates, deletes, or updates) that must be applied atomically to the target database to ensure consistency and the ability to recover.

By default, updates are committed immediately; calling this method delays the commitment of updates until the **commitExternalTransaction** method is called. If the external database does not support transactions (use the **canTransact** method to determine this), calling the **beginExternalTransaction** method has no effect.

canTransact

Signature `canTransact(): Boolean;`

The **canTransact** method of the **ExternalDatabase** class returns **true** if the connected database supports transactions.

close

Signature `close() updating;`

The **close** method of the **ExternalDatabase** class closes the connection to an external database.

commitExternalTransaction

Signature `commitExternalTransaction() updating;`

The **commitExternalTransaction** method of the **ExternalDatabase** class commits the transaction.

If the external database supports transactions, call this method at the end of a series of updating operations (creates, deletes, or updates) to commit or apply the changes to the external database.

If the external database does not support transactions (use the **canTransact** method to determine this), calling the **commitExternalTransaction** method has no effect.

executeSQL

Signature `executeSQL(sql: String);`

The **executeSQL** method of the **ExternalDatabase** class directly executes the SQL statement specified in the **sql** parameter. As this method does not return any data, it is not suitable for data retrieval operations.

Use this method to perform searched updates or call stored procedures that do not return a result set. If the SQL statement is invalid, an ODBC exception containing driver or data-source diagnostics is raised.

getFileDSN

Signature `getFileDSN(): String updating;`

The **getFileDSN** method of the **ExternalDatabase** class returns the file data source name (DSN) expression from the external database definition.

This method returns a value only if the external database definition contains a file data source name expression. (See also the **ExternalDatabase** class **getMachineDSN** method.)

getLastError

Signature
```
getLastError(native: Integer output;
             state:  String output): String updating;
```

The **getLastError** method of the **ExternalDatabase** class can be called to return error information for the last ODBC exception when the **isSQLValid** method returns **false**. The error is returned as the text string associated with the **errorCode** property of the **Exception** class.

The **native** parameter contains the error code of the native data-source from the **ODBCException** class **nativeError** property, and the **state** parameter contains the ODBC-defined state variable from the **state** property of the **ODBCException** class.

getMachineDSN

Signature `getMachineDSN(): String updating;`

The **getMachineDSN** method of the **ExternalDatabase** class returns the machine data source name (DSN) expression from the external database definition.

This method returns a value only if the external database definition contains a machine data source name expression. (See also the **ExternalDatabase** class **getFileDSN** method.)

importStoredProcedures

Signature `importStoredProcedures() updating;`

The **importStoredProcedures** method of the **ExternalDatabase** class is for internal use only.

isOpen

Signature `isOpen(): Boolean;`

The **isOpen** method of the **ExternalDatabase** class returns **true** if the external database is currently open or it returns **false** if it is closed.

isSQLValid

Signature `isSQLValid(sql: String): Boolean;`

The **isSQLValid** method of the **ExternalDatabase** class checks the syntax of the SQL statement specified in the **sql** parameter. It returns **true** if the syntax is valid and it is supported by the driver and the data source.

If the SQL syntax is not valid and supported, this method returns **false**. No exception is raised if the syntax is not acceptable.

isUpdatable

Signature `isUpdatable(): Boolean;`

The **isUpdatable** method of the **ExternalDatabase** class determines whether the connected database allows updates. Not all drivers support updating; for example, the JADE ODBC driver.

loadProcedure

Signature
```
loadProcedure(name:  String;
              index: Integer);
```

The **loadProcedure** method of the **ExternalDatabase** class is not yet implemented. It is reserved for future use.

open

Signature `open() updating;`

The **open** method of the **ExternalDatabase** class opens a connection to an external database.

The connection must have been opened in the same node that accesses the database (and that node must, of course, have the correct ODBC connections defined and available externally). For example, you can use the **serverExecution** method option to open, access, and close the external database from a method. You cannot, however, open the external database on a client node, access the external database on the server node, and then close it on a client node (that is, all three actions must take place on the same node).

setFileDSN

Signature `setFileDSN(dsn: String) updating;`

The **setFileDSN** method of the **ExternalDatabase** class programmatically sets the file data source name (DSN) expression in the external database connection string.

Note Using this method to set the data source name expression in the connection string overwrites any value that was previously set by using the **setMachineDSN** method.

---

setMachineDSN

Signature `setMachineDSN(dsn: String) updating;`

The **setMachineDSN** method of the **ExternalDatabase** class programmatically sets the machine data source name (DSN) expression in the external database connection string.

Note Using this method to set the data source name expression in the connection string overwrites any value that was previously set by using the **setFileDSN** method.

ExternalDictionary Class

The **ExternalDictionary** class encapsulates the behavior of an ordered virtual collection containing keys that represents the rows in a result set generated from an SQL query containing a sort specification; that is, the **ORDER BY** clause. External dictionaries provide direct key access to an external object instance; that is, random access to a row or tuple in the relational database. For example:

```
department := E_Company.departments[name];
```

The **ORDER BY** specification is generated by the External Schema Wizard and represents the order specification defined for the member-key attribute values.

For details about accessing external dictionary keys and the methods defined in the **ExternalDictionary** class, see "Associating External Dictionary Key Access Using Subscript Notation" and "ExternalDictionary Methods", in the following subsections.

Inherits From: ExternalCollection

Inherited By: (None)

Associating External Dictionary Key Access Using Subscript Notation

The bracket (**[]**) substring notation provides you with a shortcut for the **ExternalDictionary** class **getAtKey** method that supports random access with a key equal search condition, as shown in the following example.

```
customer := customers[name];
```

ExternalDictionary Methods

Use the **startKey** methods to start or restart at a selected position in the external dictionary or to synchronize a list box or any list style view with an associated dictionary of objects.

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

Method	Description
getAtKey	Returns the object at the specified key
getAtKeyGeq	Returns the object with a key greater than or equal to the specified key
getAtKeyGtr	Returns the object with a key greater than the specified key
getAtKeyLeq	Returns the object with a key less than or equal to the specified key
getAtKeyLss	Returns the object with a key less than the specified key
includesKey	Returns **true** if the receiver contains an entry at the specified key
startKeyGeq	Sets a start position within a collection for an external iterator object
startKeyGtr	Sets a start position within a collection for an external iterator object at the next object after the specified key
startKeyLeq	Sets a start position within a collection for an external iterator object at the object equal to or before the specified key
startKeyLss	Sets a start position within a collection for an external iterator object at the object before the specified key

getAtKey

Signature `getAtKey(keys: KeyType): MemberType;`

The **getAtKey** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for an exact match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

If a row is selected, a proxy reference representing that row is returned; otherwise this method returns **null**.

The following example shows the use of the **getAtKey** method.

```
listBoxEmployees_dblClick(listbox: ListBox input) updating;
vars
    emp  : Employees;
    emps : EmployeesByLastNameDict;
    cm   : CustMaintForExternalDB;
begin
    create emps;
    emp := emps.getAtKey(listBoxEmployees.text);
    create cm;
    cm.textBoxName.enabled := false;
    cm.myEmployee          := emp;
    cm.textBoxName.text    := emp.firstName & " " & emp.lastName.toUpper;
    cm.textBoxCity.text    := emp.city;
    cm.textBoxAddress.text := emp.address;
    cm.show;
epilog
    delete emps;
end;
```

getAtKeyGeq

Signature `getAtKeyGeq(keys: KeyType): MemberType;`

The **getAtKeyGeq** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for a greater than or equal key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

If a row is selected, a proxy reference representing that row is returned; otherwise this method returns **null**.

getAtKeyGtr

Signature `getAtKeyGtr(keys: KeyType): MemberType;`

The **getAtKeyGtr** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for a greater than key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

If a row is selected, a proxy reference representing that row is returned; otherwise this method returns **null**.

getAtKeyLeq

Signature `getAtKeyLeq(keys: KeyType): MemberType;`

The **getAtKeyLeq** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for a less than or equal key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

If a row is selected, a proxy reference representing that row is returned; otherwise this method returns **null**.

getAtKeyLss

Signature `getAtKeyLss(keys: KeyType): MemberType;`

The **getAtKeyLss** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for a less than key match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

If a row is selected, a proxy reference representing that row is returned; otherwise this method returns **null**.

includesKey

Signature `includesKey(keys: KeyType): Boolean;`

The **includesKey** method of the **ExternalDictionary** class issues a singleton SQL select, which searches for an exact match between the member-key attribute values of virtual instances (rows) and the corresponding key parameters.

This method returns **true** if a row is selected; otherwise it returns **false**.

startKeyGeq

Signature
```
startKeyGeq(keys: KeyType;
            iter: Iterator);
```

The **startKeyGeq** method of the **ExternalDictionary** class sets a start position specified in the **keys** parameter within a collection for the external **Iterator** object specified in the **iter** parameter.

startKeyGtr

Signature
```
startKeyGtr(keys: KeyType;
            iter: Iterator);
```

The **startKeyGtr** method of the **ExternalDictionary** class sets a start position within a collection for the external **Iterator** object specified in the **iter** parameter at the next object after the key specified in the **keys** parameter.

startKeyLeq

Signature
```
startKeyLeq(keys: KeyType;
            iter: Iterator);
```

The **startKeyLeq** method of the **ExternalDictionary** class sets a start position within a collection for the external **Iterator** object specified in the **iter** parameter at the object equal to or before the key specified in the **keys** parameter.

startKeyLss

Signature

```
startKeyLss(keys: KeyType;
            iter: Iterator);
```

The **startKeyLss** method of the **ExternalDictionary** class sets a start position within a collection for the external **Iterator** object specified in the **iter** parameter at the object before the key specified in the **keys** parameter.

ExternalIterator Class

The **ExternalIterator** class encapsulates the behavior required to sequentially access elements of an external collection. An external iterator instance sequentially accesses the virtual instances of the collection, in a forward or a reverse direction.

External iterators provide the operations required to scroll an SQL cursor associated with the result set of the query, which was used to populate the external collection.

For details about the methods defined in the **ExternalIterator** class, see "ExternalIterator Methods", in the following subsection.

Inherits From: Iterator

Inherited By: (None)

ExternalIterator Methods

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

Method	Description
back	Accesses entries in reverse order in the collection to which the external iteration is attached
getCollection	Returns the external collection associated with the receiver
isValid	Returns **true** if the receiver is a valid external iterator
next	Accesses successive entries in the collection to which the external iterator is attached
reset	Initializes the external iterator
startAtIndex	Sets the starting position of the external iterator to a specified row in the result set

back

Signature `back(value: Any output): Boolean updating;`

The **back** method of the **ExternalIterator** class scrolls backwards through an SQL result set and returns a proxy reference in the **value** parameter representing each row as the cursor is scrolled.

This method returns **true** when it has accessed rows or it returns **false** if a row cannot be found or is invalid.

getCollection

Signature `getCollection(): ExternalCollection;`

The **getCollection** method of the **ExternalIterator** class returns the external collection currently associated with the receiver.

If no external collection is associated with the receiver, a **null** value is returned.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **ExternalIterator** class returns **true** if the receiver is a valid external iterator.

The **isValid** method returns **false** when the iterators snapshot of entries is out of date; that is, if entries have been added or deleted to the collection that is being iterated. (Note that the iterator detects changes to the collection only in the cache of the executing node.)

next

Signature `next(value: Any output): Boolean updating;`

The **next** method of the **ExternalIterator** class scrolls forwards through an SQL result set and returns a proxy reference in the **value** parameter representing each row as the cursor is scrolled.

This method returns **true** when it has accessed rows or it returns **false** if a row cannot be found or is invalid.

reset

Signature `reset() updating;`

The **reset** method of the **ExternalIterator** class resets the state of an external iterator.

After the external iterator has been reset, the first **next** or **back** method operation causes a reissue of the default SQL query associated with the attached external collection.

startAtIndex

Signature `startAtIndex(index: Integer64) updating;`

The **startAtIndex** method of the **ExternalIterator** class positions the cursor at the result set row specified in the **index** parameter.

ExternalObject Class

The **ExternalObject** class provides a superclass for all external class subclasses and defines the behavior specific to external proxy classes. The query engine uses the **ExternalObject** class at run time to populate virtual proxy instances. Each external class contains the SQL query required to populate a class extent or to do a **join** query for a single valued reference.

For details about the methods defined in the **ExternalObject** class, see "ExternalObject Methods", in the following subsection.

Inherits From: Object

Inherited By: External database-defined and user-defined external object subclasses

ExternalObject Methods

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

Method	Description
deleteSelf	Deletes the external object
isUpdatable	Returns **true** if instances can be updated (that is, create, delete, or update instances)
update	Completes a create or update operation and saves changes to the external database

deleteSelf

Signature `deleteSelf() updating;`

The **deleteSelf** method of the **ExternalObject** class deletes the external object. This method enables you to perform a *cursor-based*, or positioned, deletion update of the current proxy object. As this method can be used to delete a proxy object only when it has been selected, the object must have been read from the external database by using an external collection or an iterator "query" method.

To delete an external object that has just been created, you must first refetch that object using query methods of a collection. If you attempt to delete an object immediately after creating it, an ODBC exception is raised.

isUpdatable

Signature `isUpdatable(): Boolean;`

The **isUpdatable** method of the **ExternalObject** class determines whether instances of the external class can be updated. This method returns **false** if the data-source is read-only or the class is read-only. An external class is read-only if it is based on a relational view or **join** query defined by the External Schema Wizard.

update

Signature `update() updating;`

The **update** method of the **ExternalObject** class completes a create operation or an update operation and saves changes to the external database. This method enables you to perform a *cursor-based*, or positioned, update of the current proxy object.

ExternalSet Class

The **ExternalSet** class encapsulates the behavior of an unordered virtual collection that represents the rows in a result set generated from an SQL query that has no sort specification; that is, it has no **ORDER BY** clause.

The order in which instances are retrieved is dependent on your data-source.

For details about the method defined in the **ExternalSet** class, see "ExternalSet Method", in the following subsection.

Inherits From: ExternalCollection

Inherited By: External database-defined and user-defined external set subclasses

ExternalSet Method

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

Method	Description
includes	Returns **true** if the virtual external collection or result set contains the specified object

includes

Signature `includes(key: MemberType): Boolean;`

The **includes** method of the **ExternalSet** class returns **true** if the virtual external collection or result set contains the object specified in the **key** parameter.

This method results in a key-equal query, based on the attributes that comprise the primary keys of the proxy class.

ExtKeyDictionary Class

The **ExtKeyDictionary** class encapsulates the behavior required to access entries in external key dictionary subclasses.

External key dictionaries are dictionaries in which the keys are not derived from the properties in the member objects but are external values supplied as parameters to the access methods.

Note The **add**, **remove**, and **includes** methods are defined at the **Collection** class level to provide closure and are inherited by all subclasses of collection. However, use of these **Collection** class methods with external key dictionaries is not recommended because none of the method signatures allow for the specification of external keys.

For details about accessing dictionary keys and the methods defined in the **ExtKeyDictionary** class, see "Using Subscripts in Dictionaries" and "ExtKeyDictionary Methods", in the following subsections.

Inherits From: Dictionary

Inherited By: ObjectByObjectDict, ObjectLongNameDict

Using Subscripts in Dictionaries

The bracket (**[]**) substring operators enable you to assign values to and receive values from a dictionary. The code fragments in the following examples show the syntax of bracket subscript operators in **ExtKeyDictionary** methods.

```
productDict[prodName] := prod;

prod := productDict[prodName];

customerDict["Sid Who", "12 Any Avenue", date1] := cust;

cust := customerDict["Sid Who", "12 Any Avenue", date1];
```

ExtKeyDictionary Methods

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

Method	Description
add	Adds an object to an external key dictionary
putAtKey	Adds a specified key to a dictionary
remove	Removes an object from an external key dictionary

add

Signature `add(value: MemberType) updating;`

The **add** method of the **ExtKeyDictionary** class adds the object specified in the **value** parameter with a **null** associated key or keys to an external key dictionary.

If you add multiple objects to an external key dictionary by using the **add** method, the dictionary must be defined to allow duplicate keys. For most applications, this is not particularly useful.

If the external keys are known to the application, it is preferable to use the **putAtKey** method to insert the entries. If the keys are properties of the object, a member key dictionary is more appropriate. If the keys are not known or required, consider using a **Set**.

Note The **add** method is defined at the **Collection** class level to provide closure and is inherited by all subclasses of collection. However, use of the **Collection** class **add** method with external key dictionaries is not recommended because the method signature does not allow for the specification of external keys.

putAtKey

Signature `putAtKey(keys: KeyType;`
`value: MemberType) updating;`

The **putAtKey** method of the **ExtKeyDictionary** class adds the object specified in the **value** parameter to a dictionary. If duplicate entries are not allowed and an entry already exists for the key specified in the **keys** parameter, an exception is raised.

The following is an example of the use of the **putAtKey** method.

```
custNameDict.putAtKey(cust.name, cust);
```

The code fragments in the following examples show the use of the bracket (**[]**) subscript operators to assign values to a dictionary.

```
custNameDict[custName] := cust;

custNameDict["Mr Who", "11 Any Road", date] := cust;
```

remove

Signature `remove(value: MemberType) updating;`

The **remove** method of the **ExtKeyDictionary** class searches the external key dictionary for the object entry specified in the value **parameter** with **null** values in its associated key or keys. If a matching entry is found, the object is removed from the dictionary. If it is not found, a 1301 exception is raised (*Entry not found in collection*).

This method is useful only when objects have been added to the dictionary by using the **ExtKeyDictionary** class **add** method and the dictionary contains a single entry or it has been defined to allow duplicate key entries.

If you insert objects by using the **ExtKeyDictionary** class **putAtKey** method, you should use the matching **Dictionary**::**removeKey** or **Dictionary** class **removeKeyEntry** method to remove the entries, as these methods enable you to specify the external keys to be used for the search operation.

The following is an example of the use of the **remove** method.

```
wordIndex.remove(word);
```

Notes The **remove** method is defined at the **Collection** class level to provide closure and is inherited by all subclasses of collection. However, use of the **Collection** class **remove** method with external key dictionaries is not recommended because the method signature does not allow for the specification of external keys.

As external key dictionaries are not automatically maintained by the JADE system, it is your responsibility to manually remove the entry from the dictionary when a member is deleted.

To remove duplicate keys, use the **Dictionary** class **removeKeyEntry** method.

FatalError Class

The **FatalError** class is the transient class for serious internal faults.

Inherits From: Exception

Inherited By: (None)

File Class

The **File** class enables you to read and write disk files, either sequentially or with random access. The following example shows the use of properties and methods defined in the **File** class.

```
loadCustomers();
vars
    file               : File;
    cust               : Customer;
    line, tName        : String;
    tAddress, tContact : String;
    company            : Company;
begin
    create file;
    file.fileName := "c:\data\customers.txt";
    file.kind     := File.Kind_Unknown_Text;
    file.mode     := File.Mode_Input;
    file.open;
    beginTransaction;
        while not file.endOfFile do
            line      := file.readLine;
            tName     := line[1:29].trimRight;
            tContact  := line [31:24].trimRight;
            tAddress  := line [56:end].trimRight;
            create cust;
            cust.loadSelf(company, tName, tAddress, tContact);
        endwhile;
    commitTransaction;
epilog
    delete file;
end;
```

Note You cannot create persistent instances of the **File** class.

If you need to create shared transient instances of the **File** class, note that as the implicit opening of a shared transient file updates the state within the file object, you must explicitly open a file inside a transient transaction, as shown in the following example.

```
constants
    FileName : String = 'c:\data\jade\MyTestFile.txt';
vars
    file : File;
begin
    beginTransientTransaction;
        create file sharedTransient;
        file.fileName := FileName;
        file.open;
        ...           // do some processing here
        file.close;
        delete file;
    commitTransientTransaction;
end;
```

In JADE thin client mode, instances of the **File** class refer to files on the presentation client when the **FileNode** class **usePresentationFileSystem** property is set to **true** and the instances are *not* shared transient instances. Shared transient instances of the **File** class are processed on the application server, and the setting of the **usePresentationFileSystem** property is ignored. (A file opened on one presentation client cannot be accessed by another client.)

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

Inherits From: FileNode

Inherited By: (None)

File Class Constants

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

Constant	Value	Description
Kind_ANSI	2	ANSI text file (default for ANSI JADE)
Kind_Binary	1	Binary file
Kind_Unicode	3	Unicode text file (default for Unicode JADE)
Kind_Unicode_UTF16BE	5	Unicode Transformation Format (UTF) 16-bit, big-endian
Kind_Unicode_UTF16LE	6	Unicode Transformation Format (UTF) 16-bit, little-endian
Kind_Unicode_UTF32BE	9	Unicode Transformation Format (UTF) 32-bit, big-endian
Kind_Unicode_UTF32LE	10	Unicode Transformation Format (UTF) 32-bit, little-endian
Kind_Unicode_UTF8	8	Unicode Transformation Format (UTF) 8 bit
Kind_Unknown_Text	4	Not known if file is ANSI, binary, or Unicode
Mode_Append	3	Append file (output only)
Mode_IO	0	Input-Output file (the default)
Mode_Input	1	Input file only
Mode_Output	2	Output file only
Share_Exclusive	3	Enables the file to be opened for exclusive access, preventing another thread or process from opening the file concurrently
Share_Read	2	Enables another thread or process to open the file for read access
Share_ReadWrite	0	Enables another thread or process to open the file for read/write access (the default)
Share_Write	1	Enables another thread or process to open the file for write access

File Properties

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

Property	Description
allowReplace	Specifies if a file can be overwritten
endOfField	Contains the end of field string that delimits variable-length fields
endOfLine	Contains the **endOfLine** string that delimits variable-length records (or lines)
kind	Contains the kind of file that is to be opened
maxIOSize	Contains the maximum size in bytes for I/O operations when reading and writing a file
maxRecordSize	Contains the maximum size in file units of a buffer that can be read from a text file
mode	Contains the mode in which a file is accessed
recordSize	Contains the size in characters of a fixed-length record
shareMode	Restricts access to the file by other processes
unicodeBOM	Specifies whether a Unicode Byte Order Mark (BOM) is present in the file

allowReplace

Type: Boolean

The **allowReplace** property of the **File** class specifies whether a file can be overwritten when it is opened. For example, if the value of the **allowReplace** property is **false** and a file is opened in output mode (that is, the **mode** property has a value of **File.Mode_Output**) and a file of the same name already exists, the file is not overwritten and an exception is raised.

The default value is **true**; that is, the file can be overwritten if it already exists.

The value is ignored if the file is already open.

The code fragment in the following example shows the use of the **allowReplace** property when using the **File** class **extractSort** method to sort a file and replace the file with the output of the sort. The result of the sort is then written to the text box.

```
file.allowReplace := true;
file.extractSort(sortActorArray, file);
textBox2.text := file.readString(400);
resetOutputFile;
file.close;
```

endOfField

Type: String[6]

The **endOfField** property of the **File** class contains the end-of-field string that delimits variable-length fields when using the **extractSort** method to order variable-length field files.

By default, this property contains a **null** value.

endOfLine

Type: String[3]

The **endOfLine** property of the **File** class contains the end-of-line string that delimits variable-length records (or lines) when using the **extractSort**, **readLine**, and **writeLine** methods.

Setting the value of **endOfLine** to **""** (an empty, or null, string) specifies that any end-of-line value (**CR/LF**, **CR**, or **LF**) found in the file is used as a valid end of line when reading the file with the **readLine** method. This is the default action when the **PlatformOptions** parameter in the [JadeClient] section of the JADE initialization file is set to **MixedOS**.

When using the **writeLine** or **extractSort** method, the default end-of-line value for the platform on which the file is defined is used, unless you have explicitly set the value of the **endOfLine** property to a non-null value.

If the value of the **PlatformOptions** parameter is **PlatformOS**, the returned value is one of the following.

- The non-null value that has been set (if any).
- The default end-of-line value for the platform on which the file is currently located. The default value is **CR/LF**.

If the value of the **PlatformOptions** parameter is **MixedOS**, the returned value is one of the following.

- The non-null value that has been set (if any).
- If the file has not been read by using the **readLine** method, the default end-of-line value for the platform on which the file is currently located. The default value is **CR/LF**.
- If the file has been read by using the **readLine** method, the last end-of-line value found in the file (**CR/LF**, **CR**, or **LF**).

kind

Type: Integer

The **kind** property of the **File** class contains the kind of file that is to be opened.

File class constants are provided for the types of file, as listed in the following table.

Constant	Value	Description
Kind_ANSI	2	ANSI text file (default for ANSI JADE)
Kind_Binary	1	Binary file
Kind_Unicode	3	Unicode text file (default for Unicode JADE)
Kind_Unicode_ UTF16BE	5	Unicode Transformation Format (UTF) 16-bit, big-endian
Kind_Unicode_UTF16LE	6	Unicode Transformation Format (UTF) 16-bit, little-endian
Kind_Unicode_ UTF32BE	9	Unicode Transformation Format (UTF) 32-bit, big-endian
Kind_Unicode_UTF32LE	10	Unicode Transformation Format (UTF) 32-bit, little-endian
Kind_Unicode_UTF8	8	Unicode Transformation Format (UTF) 8 bit
Kind_Unknown_Text	4	Not known if file is ANSI or Unicode (the value is set when the file is opened)

If the **kind** property is not one of these types when the file is opened, an exception is raised.

In the big-endian variant of the UTF 16-bit and 32-bit formats, the high byte precedes the low byte. Conversely, the low byte precedes the high byte in the little-endian UTF 16-bit and 32-bit formats.

A Unicode text file (that is, the **kind** property has a **File** class constant value of **Kind_Unicode**) behaves the same as the **Kind_Unicode_UTF16LE** value.

JADE handles the **Kind_Unicode** constant by defaulting to the wide-character (UTF 16 or UTF 32) little-endian or big-endian format, as defined for the process operating system. Use the UTF constant values only if you want to enforce the Unicode output format.

Tip As the output to the kind of file fails if you use one of the UTF constant values to enforce specific coding requirements and the format is incompatible with the operating system in which the process is running, you should use the **Kind_Unicode** constant in most cases and let JADE handle the format for you.

The following example shows how to determine the kind of a text file.

```
vars
    file : File;
begin
    create file transient;
    file.kind := File.Kind_Unknown_Text;
    file.openInput("C:\temp\words.txt");
    write file.kind;
epilog
    delete file;
end;
```

maxIOSize

Type: Integer

The **maxIOSize** property of the **File** class contains the maximum size in bytes of an I/O operation. A read or write operation that is larger than the value of **maxIOSize** property is performed in a number of I/O operations, which do not exceed the value of **maxIOSize** property.

The default value of zero (**0**) units performs the read or write as a single I/O operation.

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

```
create file transient;
file.mode := File.Mode_Output;
file.kind := File.Kind_Binary;
file.fileName := "\\host\share\filename.dat";
file.maxIOSize := 16*1024;  //16KB I/O's
file.open;
buffer[80*1024*1024] := 'x'.Binary;  // Create an 80MB buffer
file.writeBinary(buffer);
```

This causes 5120 write I/O operations each 16K bytes in size to the destination file, rather than a single 80M byte write that would fail under current Windows operating systems raising an exception (5040 - Insufficient System Resources).

Note If the file is opened in shared mode and concurrent updates are allowed, data could become interleaved with other operating system processes writing to the same file. This issue would also arise, without using the **maxIOSize** property, if the code fragment was changed to perform a number of smaller write operations.

maxRecordSize

Type: Integer

The **maxRecordSize** property of the **File** class contains the maximum size in file units of a buffer that can be read from a text file by using the **readLine** method. If an attempt is made to read a line larger than **maxRecordSize**, the line is truncated to the number of characters specified by the **maxRecordSize** property and an exception is raised.

The default value is **8192** units.

The units of a file depend on the type of file; that is, the type determined by the **kind** property. For Unicode text files, units are Unicode characters (each Unicode character is two bytes). For ANSI text files, units are ANSI characters (each ANSI character is one byte).

The value of the **maxRecordSize** property can be changed at any time. The new value takes effect from the next **read** operation, enabling the maximum record length of a file to be changed dynamically as the file is read.

mode

Type: Integer

The **mode** property of the **File** class contains the mode in which a file is accessed. This property can be associated with one of the **File** class constant values listed in the following table.

Constant	Value	Description
Mode_Append	3	Append file (output only)
Mode_IO	0	Input-Output file (the default)
Mode_Input	1	Input file only
Mode_Output	2	Output file only

recordSize

Type: Integer

The **recordSize** property of the **File** class contains the size in characters of a fixed-length record when sorting a file. Fixed-length records are not delimited by the **endOfLine** character sequence. When sorting fixed-length records, the entire length of the file must be divisible by the record size, with no remainder.

The default value is zero (**0**). If the value of the **recordSize** property is set to zero (**0**), the file is treated as a variable-length record file type.

The following example shows the use of the **recordSize** property to set the record size of the file to zero (**0**), indicating the file to be sorted has variable records. In this example, the records are delimited by carriage return and line feed characters.

```
file1.recordSize := 0;
file1.endOfLine  := CrLf;
```

shareMode

Type: Integer

The **shareMode** property of the **File** class restricts access to the file by other processes. If another process has a conflicting lock, an exception is raised.

Note To reduce the number of messages sent between the application server and the presentation client when reading a static file (one that has the **mode** property set to **Mode_Input**) from the presentation client when running JADE in thin client mode, set the **shareMode** property to **Share_Read** or **Share_Exclusive** so that the file is read by the application server in chunks from the presentation client and the file buffer management is then performed by the application server.

The **File** class constants provided for shared file access modes are listed in the following table.

Constant	Value	Enables…
Share_Exclusive	3	The file to be opened for exclusive access, preventing another thread or process from opening the file concurrently
Share_Read	2	Another thread or process to open the file for read access
Share_Write	1	Another thread or process to open the file for write access
Share_ReadWrite	0	Another thread or process to open the file for read/write access (the default)

unicodeBOM

Type: Boolean

The **unicodeBOM** property of the **File** class specifies if the Unicode Byte-Order Mark (BOM) is present at the start of a Unicode file. (A byte-order mark indicates the byte ordering of text in the file.)

If the file byte ordering differs from the native byte ordering, JADE converts the data as part of reading and writing the Unicode file.

If a Unicode file is opened in **Mode_Input** or **Mode_IO**, the **unicodeBOM** property is set to **true** if a Unicode byte order mark is present at the start of the file. As this property is set to **true** by default, set this property to **false** if you want to suppress the creation of a byte order mark when a Unicode file is opened in **Mode_Output**.

The **unicodeBOM** property has no meaning or effect for an ANSI or Binary file.

File Methods

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

Method	Description
close	Closes an open file
commit	Commits any outstanding updates that have been cached in memory to the file
currentOffset	Returns the current offset as an **Integer** value (in file units)
currentOffset64	Returns the current offset as an **Integer64** value (in file units)
currentOffsetDec	Returns the current offset as a **Decimal** value (in file units)

Method	Description
endOfFile	Returns **true** if the end of the file has been reached
errorCode	Returns the last operating system error code as an integer value
extractSort	Sorts the contents of the file into the specified sort order
fileLength	Returns the size of the file as an **Integer** value (in file units)
fileLength64	Returns the size of the file as an **Integer64** value (in file units)
fileLengthDec	Returns the size of the file as a **Decimal** value (in file units)
isAvailable	Returns **true** if the specified file is available
isOpen	Returns **true** if the specified file is currently open
lastAccessed	Returns the **TimeStamp** of the last access of the file
lastModified	Returns the **TimeStamp** of the last modification of the file
open	Opens the file
openInput	Opens the file for input
openOutput	Opens the file for output
peek	Returns the file record at the specified position
purge	Deletes the file and its contents from disk
readBinary	Returns the specified number of bytes from the current file offset
readLine	Returns the next line in the file
readString	Returns the specified number of characters from the file
rename	Changes the name of the file
seek	Repositions the file pointer at a specified **Integer** offset from the beginning of the file
seek64	Repositions the file pointer at a specified **Integer64** offset from the beginning of the file
seekDec	Repositions the file pointer at a specified **Decimal** offset from the beginning of the file
timeCreated	Returns the **TimeStamp** of the date and time at which the file was created
tryOpen	Opens the file and returns **true** if successful
writeBinary	Writes the specified number of bytes to the file
writeLine	Writes the specified line to the file
writeString	Writes the specified string to the file

close

Signature `close();`

The **close** method of the **File** class closes an open file.

Note As the file is automatically closed when the file object is deleted, this method is optional.

However, this applies only when you delete the file object yourself; for example, in the epilog of a method. If you leave it to JADE to delete the object when the process terminates, the file may still be held open and you will have to terminate the node that owned the process.

The following is an example of the use of the **close** method.

```
resetOutputFile1() updating;
begin
    // If an output file has been created, it is reset by deleting the
    // contents and then redefining the file name.
    if outputFile1.isAvailable then
        outputFile1.close;
        outputFile1.purge;
        outputFile1.allowCreate := true;
        outputFile1.fileName    := "c:\temp\outputFile1.txt";
    endif;
end;
```

commit

Signature `commit();`

The **commit** method of the **File** class physically commits to the file any outstanding updates that have been cached in memory for performance reasons. The commit flushes any pending "dirty" buffers and updates the file size maintained by the file system.

The following example shows the use of the **commit** method.

```
btnCommit_click(btn: Button input) updating;
begin
    if file <> null then
        file.commit;
    endif;
    showFileStatus;
end;
```

currentOffset

Signature `currentOffset(): Integer;`

The **currentOffset** method of the **File** class returns the current offset as an integer value (in file units).

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker is ignored; that is, **currentOffset = 0** is the first character of data.

Note If the length of the file is greater than 2G byte file units, use the **currentOffset64** method to retrieve the current offset.

The following is an example of the use of the **currentOffset** method.

```
btnReadSeq_click(btn: Button input) updating;
vars
    len : Integer;
begin
```

```
    len := file.fileLength - file.currentOffset;
    lblFileStatus.caption := file.readString(len);
end;
```

currentOffset64

Signature `currentOffset64(): Integer64;`

The **currentOffset64** method of the **File** class returns the current offset as an **Integer64** value (in file units).

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker is ignored; that is, **currentOffset64 = 0** is the first character of data.

The following is an example of the use of the **currentOffset64** method.

```
btnReadSeq_click(btn: Button input) updating;
vars
    len : Integer64;
begin
    len := file.fileLength - file.currentOffset64;
    lblFileStatus.caption := file.readString(len);
end;
```

currentOffsetDec

Signature `currentOffsetDec(): Decimal;`

The **currentOffsetDec** method of the **File** class returns the current offset (in file units) as a decimal value.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker is ignored; that is, **currentOffsetDec = 0** is the first character of data.

Note Use **Decimal[23,0]** if you are storing the returned value.

endOfFile

Signature `endOfFile(): Boolean;`

The **endOfFile** method of the **File** class returns **true** if the end of the file has been reached. The code fragment in the following example shows the use of the **endOfFile** method.

```
while not file.endOfFile do
    line     := file.readLine;
    tName    := line[1:29].trimRight;
    tContact := line[31:24].trimRight;
    tAddress := line[56:end].trimRight;
    create cust;
    cust.loadSelf(company, tName, tAddress, tContact);
endwhile;
```

The **endOfFile** method returns **false** if the file is not currently open.

errorCode

Signature `errorCode(): Integer;`

When a file open, close, or I/O operation fails at the operating system level, the **errorCode** method of the **File** class returns the operating system error code as an integer value.

If the error is not operating system-related or if the last I/O operation was successful, this method returns zero (**0**).

extractSort

Signature
```
extractSort(sortActorArray: SortActorArray;
            targetFile:     File);
```

The **extractSort** method of the **File** class sorts the contents of the file into the sort order indicated by the **sortActorArray** parameter and writes the results of the sort to the target file specified in the **targetFile** parameter.

Note You cannot run this method on a thin client (where the **FileNode** class **usePresentationFileSystem** property is set to **true**) or with a Unicode version.

If the value of the **recordSize** property is zero (**0**), the file is read as variable-length records delineated by **endOfLine** property values. If the **recordSize** value is not zero, the file is read as fixed-length records of the specified record size, and the value of the **endOfLine** property is not used.

The **SortActorArray** class is used to determine the precedence of records in a file, based on the properties of the **SortActor** class specified in the **sortActorArray** parameter. The sort actors that can be specified in the **sortActorArray** parameter are:

- ascending
- fieldNo
- length
- numeric
- startPosition
- sortType
- random

The locale in which the sorting is performed is defined by the **lcid** property of the **SortActorArray** class. The **lcid** property default value of **768** specifies an invalid locale id. If this default value is used, it is remapped to the current locale used by JADE. The locale determines the precedence of non-numeric fields and records.

The sorted order of duplicate records is not determined.

Notes When using the **extractSort** method, the numeric fields must be less than or equal to 14 characters and in the range represented by the **Min_Integer** and **Max_Integer** global constants. (For details, see "SystemLimits Category" in Appendix A of the *JADE Encyclopaedia of Primitive Types*.)

You cannot perform a numeric sort if the field that you are sorting contains decimals.

For details about specifying the directory in which temporary sort files are located, see the **SortDirectory** parameter under "JADE Extract Sort Section [JadeExtractSort]", in the *JADE Initialization File Reference*.

Example of Sorted Variable Fields or Records

The following example shows the use of the **extractSort** method to sort variable-length fields or records.

```
variableTest();
vars
    saa                    : SortActorArray;
    sortActor1, sortActor2 : SortActor;
    file                   : File;
begin
    create file transient;
    create saa transient;
    create sortActor1 transient;
    create sortActor2 transient;
    file.recordSize := 0; // Variable length records
    file.endOfLine := CrLf; // <cr><lf> record delimiter
    file.endOfField := ","; // Comma-delimited fields
    file.fileName := "c:\test\temp.txt";
    file.allowReplace := true; // Replace the source file
    // The first actor
    sortActor1.sortType := SortActor.SortType_String; // Invoke alphanumeric
    sortActor1.length := 8; // sort of length 8
    sortActor1.startPosition := 10; // Start at position 10
    sortActor1.fieldNo := 1; // of the first field
    sortActor1.ascending := true; // Ascending sort order
    sortActor1.random := false; // Not random order
    // The second actor
    sortActor2.sortType := SortActor.SortType_String; // Invoke alphanumeric
    sortActor2.length := 8; // sort of length 8
    sortActor2.startPosition := 1; // Start at position 1
    sortActor2.fieldNo := 2; // of the second field
    sortActor2.ascending := false; // Descending sort order
    saa[1] := sortActor1;
    saa[2] := sortActor2;
    file.extractSort(saa, file); // Sort the file now
epilog
    delete file;
    delete saa;
    delete sortActor1;
    delete sortActor2;
end;
```

Example of Sorted Fixed Fields or Records

The following example shows the use of the **extractSort** method to sort fixed-length fields or records.

```
fileTest();
vars
    sa         : SortActorArray;
    sortActor1 : SortActor;
    sortActor2 : SortActor;
    inputFile  : File;
    outputFile : File;
begin
    create sa transient;
```

```
    create sortActor1 transient;
    create sortActor2 transient;
    create inputFile transient;
    create outputFile transient;
    inputFile.recordSize     := 84;            // Fixed-length records
    inputFile.endOfField     := "";            // No field terminator
                                               // The first actor
    sortActor1.sortType := SortActor.SortType_String; // Invoke alphanumeric
    sortActor1.length        := 8;             // sort of length 8
    sortActor1.startPosition := 10;            // Start at position 10
    sortActor1.fieldNo       := 1;             // First field
    sortActor1.ascending     := true;          // Ascending sort order
    sortActor1.random        := false;         // Not random order
                                               // The second actor
    sortActor2.sortType := SortActor.SortType_String; // Invoke alphanumeric
    sortActor2.length        := 8;             // sort of length 8
    sortActor2.startPosition := 1;             // Start at position 1
    sortActor2.fieldNo       := 2;             // Second field
    sortActor2.ascending     := false;         // Descending sort order
    sa.add(sortActor1);
    sa.add(sortActor2);
    inputFile.fileName       := "c:\test\temp.sor";
    outputFile.fileName      := "c:\test\temp.std";
    outputFile.allowReplace  := true;    // Replace output file if necessary
    inputFile.extractSort(sa, outputFile);     // Sort the file now
epilog
    delete inputFile;
    delete outputFile;
    delete sa;
    delete sortActor1;
    delete sortActor2;
end;
```

fileLength

Signature `fileLength(): Integer;`

The **fileLength** method of the **File** class returns the size of the file (in file units). Use this method to test for an empty file. When the **fileLength** method is called for a file that is not open, the following occurs.

- If the **kind** property of the file is not **Kind_Binary**, the file is opened to determine the kind of text and then closed. If the file cannot be opened, an exception is raised.
- If the **kind** property of the file is **Kind_Binary**, the kind of the file is assumed to be correct. If the file cannot be accessed (for example, if it does not exist), a message is output to the **jommsg.log** file and a value of **-1** is returned.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored.

Note To find the length of a Unicode file in bytes instead of file units, open the file specifying the **kind** property of the file as **Kind_Binary**.

The code fragment in the following example shows the use of the **fileLength** method.

```
if (cmdFile.open = 0) then
    fileName      := cmdFile.fileName;
    create file;
    file.kind     := File.Kind_Binary;
    file.mode     := File.Mode_Input;
    file.fileName := fileName;
    photo         := file.readBinary(file.fileLength);
endif;
```

Note If the length of the file is greater than 2G byte file units, use the **fileLength64** method to retrieve the length of the file (in file units).

fileLength64

Signature `fileLength64(): Integer64;`

The **fileLength64** method of the **File** class returns the size of the file (in file units). Use this method to test for an empty file. When the **fileLength64** method is called for a file that is not open, the following occurs.

- If the **kind** property of the file is **Kind_Unknown_Text**, the file is opened to determine the kind of text and then closed. If the file cannot be opened, an exception is raised.
- If the **kind** property of the file is not **Kind_Unknown_Text**, the kind of the file is assumed to be correct. If the file cannot be accessed (for example, if it does not exist), a message is output to the **jommsg.log** file and a value of **-1** is returned.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored.

Note To find the length of a Unicode file in bytes instead of file units, open the file specifying the **kind** property of the file as **Kind_Binary**.

The code fragment in the following example shows the use of the **fileLength64** method.

```
if (cmdFile.open = 0) then
    fileName      := cmdFile.fileName;
    create file;
    file.kind     := File.Kind_Binary;
    file.mode     := File.Mode_Input;
    file.fileName := fileName;
    photo         := file.readBinary(file.fileLength);
endif;
```

fileLengthDec

Signature `fileLengthDec(): Decimal;`

The **fileLengthDec** method of the **File** class returns the size of the file (in file units) as a decimal value. Use this method to test for an empty file.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored.

isAvailable

Signature `isAvailable(): Boolean;`

The **isAvailable** method of the **File** class returns **true** if the associated file of the receiver is available; that is, if the file exists and it can be accessed with the requested mode.

Tip This method does not check that the file is in use by another user. Use the **File** class **tryOpen** method to determine if the file is in use.

The code fragment in the following example shows the use of the **isAvailable** method.

```
if file.isAvailable then
    // If the file exists, return the loaded picture
    return app.loadPicture(file.fileName);
endif;
```

isOpen

Signature `isOpen(): Boolean;`

The **isOpen** method of the **File** class returns **true** if the associated physical file is open by the current process. It does not return **true** if the file has been opened by another process.

Use the **File** class **tryOpen** method to determine if the file is in use by another process.

The code fragment in the following example shows the use of the **isOpen** method.

```
if not app.isValidObject(file) then
    create file transient;
elseif file.isOpen then
    file.close;
endif;
```

lastAccessed

Signature `lastAccessed(): TimeStamp;`

The **lastAccessed** method of the **File** class returns the **TimeStamp** of the last access of the file. See also the **FileNode** class **lastAccessedTime** property.

lastModified

Signature `lastModified(): TimeStamp;`

The **lastModified** method of the **File** class returns the **TimeStamp** of the last modification of the file. See also the **FileNode** class **lastModifiedTime** property.

open

Signature `open();`

The **open** method of the **File** class opens a file. If you do not explicitly open a file, the first **readLine**, **readString**, **seek**, **writeLine**, or **writeString** method opens the file automatically. Before the file is opened, set the **FileNode** class **allowCreate** and **fileName** properties and the **File** class **allowReplace**, **kind**, **mode**, and **shareMode** properties, as required.

Creating or Replacing Unicode Text Files

The Unicode File Marker is used by Unicode-aware applications (for example, by Notepad) to indicate that a file contains Unicode text rather than ANSI text. If a Unicode text file is created or replaced by using the **File** class **open** method, a Unicode File Marker is automatically written to the start of the file, by default.

You can create Unicode files without a Unicode File Marker by setting the **unicodeBOM** property to **false**, as the use of the marker is a recommended convention only and is not a standard.

The presence or absence of a Unicode File Marker is transparent to you so that you are presented with a consistent view of all Unicode files.

Immediately after a Unicode text file has been created in JADE, it is seen as an empty file even though it physically contains the two-byte marker on disk. The first character after the marker is the file offset **0**, and the length of the file seen from within JADE excludes the length of the marker. As a JADE application cannot access the marker, if two Unicode files contain identical text but one has a marker present, they appear identical from within JADE.

If you attempt to open a file containing an odd number of Unicode bytes, an exception is raised and the file is not opened.

openInput

Signature `openInput(fName: String) updating;`

The **openInput** method of the **File** class opens the file specified in the **fName** parameter for input.

The **openInput** method is equivalent to setting the file name, setting the mode to **Mode_Input**, and then opening the file.

openOutput

Signature `openOutput(fName: String) updating;`

The **openOutput** method of the **File** class opens the file specified in the **fName** parameter for output. This method is equivalent to setting the file name, setting the mode to **Mode_Output**, and then opening the file.

peek

Signature `peek(length: Integer): String;`

The **peek** method of the **File** class returns a string containing the number of characters specified in the **length** parameter, starting from the current position. The current file offset is unchanged.

The **peek** method automatically opens the file if it is not already open.

Notes You can use the **peek** method only with files opened as text files.

An exception is raised if the value specified in the **length** parameter is not a positive number; that is, greater than zero (**0**).

If you specify a number of characters greater than the length of the file, only the characters up to the end of the file are returned. An empty string is returned when the current position is the end of file.

The code fragment in the following example shows the use of the **peek** method.

```
str := file.peek(10);
if str[1:3] = "***" then
    file.readLine;
    return true;
endif;
```

purge

Signature `purge();`

The **purge** method of the **File** class deletes the specified file from disk.

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **File** class returns the number of bytes specified in the **length** parameter. An empty binary is returned if the end of file has been reached.

The **readBinary** method automatically opens the file if it is not already open.

This method is valid only for files that are opened with a **kind** property setting of **File.Kind_Binary**. An exception is raised if the **readBinary** method is attempted on a file opened as a text file.

The following example shows the use of the **readBinary** method.

```
vars
    bin  : Binary;
    file : File;
begin
    create file;
    file.kind := File.Kind_Binary;
    file.openInput('d:\bmps\lab3.bmp');
    bin := file.readBinary(file.fileLength);
    write bin;
    return;
epilog
    delete file;
end;
```

Note The maximum size of data that can be read by the **readBinary** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes.

If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

readLine

Signature `readLine(): String;`

The **readLine** method of the **File** class returns a string containing the next line in the file. A line is delimited by the **endOfLine** character sequence. An empty string is returned when the end of file has been reached. The **readLine** method automatically opens the file if it is not already open.

If the line (or record) exceeds the value of the **maxRecordSize** property, it is truncated to that value and an exception is raised.

This method is valid only for files that are opened as text file. An exception is raised if the **readLine** method is attempted on a file opened as binary.

If an ANSI version of JADE reads from a Unicode file, the string is automatically converted from Unicode to ANSI before it is returned. If a Unicode version of JADE reads from an ANSI file, the string is automatically converted from ANSI to Unicode before it is returned.

The following example shows the use of the **readLine** method.

```
modifyFile();
constants
    FileNameIn          : String = 'c:\jade\bin\x2.run';
    FileNameOut         : String = 'c:\jade\xx2.run';
vars
    fOut, fIn    : File;
    recordCount  : Integer;
    outputString : String;
    inputString  : String;
    temp2        : String;
begin
    create fIn;
    create fOut;
    fIn.kind      := File.Kind_Unknown_Text;
    fOut.mode     := File.Mode_Output;
    fIn.mode      := File.Mode_Input;
    fOut.fileName := FileNameOut;
    fIn.fileName  := FileNameIn;
    fOut.open;
    fIn.open;
    recordCount  := 1;
    while not fIn.endOfFile do
        inputString := fIn.readLine;
        inputString[60 : 4 ] := '0000';
        temp2 := recordCount.String;
        inputString [64 - temp2.length : temp2.length ] := temp2;
        fOut.writeLine(inputString);
        recordCount := recordCount + 1;
    endwhile;
    fOut.close;
    fIn.close;
epilog
    delete fOut;
    delete fIn;
end;
```

Note The maximum size of data that can be read by the **readLine** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes. If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

readString

Signature `readString(length: Integer): String;`

The **readString** method of the **File** class returns a string containing the number of characters specified in the **length** parameter from the file.

An empty string is returned when the end of file has been reached.

The **readString** method automatically opens the file if it is not already open.

This method is valid only for files that are opened as text files. An exception is raised if the **readString** method is attempted on a binary file.

If an ANSI version of JADE reads from a Unicode file, the string is automatically converted from Unicode to ANSI before it is returned. If a Unicode version of JADE reads from an ANSI file, the string is automatically converted from ANSI to Unicode before it is returned.

Note The maximum size of data that can be read by the **readString** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes.

If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

rename

Signature `rename(newname: String);`

The **rename** method of the **File** class changes the name of the file associated with the file object to the value specified in the **newname** parameter.

Notes An open file is closed before it is renamed.

This method does not change the value of the **FileNode** class **fileName** property.

The code fragment in the following example shows the use of the **rename** method.

```
myfile.rename(path & "changes.log");
```

JADE attempts to rename files across devices. This attempt could fail for various reasons (often related to an operating system restriction), as follows.

- File name on the output device is invalid
- Permission to create a file on the output device is denied
- Space on the output device is insufficient

seek

Signature `seek(offset: Integer);`

The **seek** method of the **File** class sets the file pointer to the position (in file units from the beginning of the file) specified in the **offset** parameter. The **seek** method automatically opens the file if it is not already open.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored. The next read or write operation occurs from the specified position.

Note If the length of the file is greater than 2G byte file units, use the **seek64** method to set the file pointer to the desired offset.

seek64

Signature `seek64(offset: Integer64);`

The **seek64** method of the **File** class sets the file pointer to the position (in file units from the beginning of the file) specified in the **offset** parameter. The **seek64** method automatically opens the file if it is not already open.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored. The next read or write operation occurs from the specified position.

seekDec

Signature `seekDec(offset: Decimal);`

The **seekDec** method of the **File** class sets the file pointer to the position as a decimal value (in file units from the beginning of the file) specified in the **offset** parameter. The **seekDec** method automatically opens the file if it is not already open.

For Unicode text, the file unit is character (not byte). For Unicode text files that contain a Unicode File Marker, the file marker specified in the **unicodeBOM** property is ignored. The next read or write operation occurs from the specified position.

timeCreated

Signature `timeCreated(): TimeStamp;`

The **timeCreated** method of the **File** class returns the **TimeStamp** of the date and time at which the file was created. See also the **FileNode** class **createdTime** property.

tryOpen

Signature `tryOpen(): Boolean;`

The **tryOpen** method of the **File** class opens the file using all attributes set (for example, its **fileName**, **mode**, and **shareMode** properties) and returns **true** if the open was successful. If the open operation is not successful, the **tryOpen** method returns **false**, without raising an exception.

writeBinary

Signature `writeBinary(bin: Binary);`

The **writeBinary** method of the **File** class writes the value of the **bin** parameter to the file. The **writeBinary** method automatically opens the file if it is not already open.

This method is valid only for files that are opened with a **kind** property setting of **File.Kind_Binary**. An exception is raised if the **writeBinary** method is attempted on a text file.

Note The maximum size of data that can be written by the **writeBinary** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes. If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

writeLine

Signature `writeLine(sline: String);`

The **writeLine** method of the **File** class writes the line specified in the **sline** parameter to the file. A line is delimited by the **endOfLine** character sequence. The **writeLine** method automatically opens the file if it is not already open. If the line to be written contains a **null** value, the line that is written ends at that point.

This method is valid only for files that are opened as text files. An exception is raised if the **writeLine** method is attempted on a binary file.

If an ANSI version of JADE writes to a Unicode file, the string is automatically converted from ANSI to Unicode before it is written. If a Unicode version of JADE writes to an ANSI file, the string is automatically converted from Unicode to ANSI before it is written.

Note The maximum size of data that can be written by the **writeLine** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes.

If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

---

writeString

Signature `writeString(str: String);`

The **writeString** method of the **File** class writes the string specified in the **str** parameter to the receiver file. This method automatically opens the file if it is not already open.

Use subscript operators if only part of the string specified in the **str** parameter is to be written to the file; for example:

```
writeString(s[1:10]);
```

The **writeString** method is valid only for files that are opened as text files. An exception is raised if the **writeString** method is attempted on a binary file.

If an ANSI version of JADE writes to a Unicode file, the string is automatically converted from ANSI to Unicode before it is written. If a Unicode version of JADE writes to an ANSI file, the string is automatically converted from Unicode to ANSI before it is written.

Note The maximum size of data that can be written by the **writeString** method when the value of the **FileNode** class **usePresentationFileSystem** property is set to **true** is 2G bytes.

If this limit is exceeded, exception 5047 (*Invalid record size*) is raised.

FileException Class

The **FileException** class is the transient class that defines behavior for exceptions that occur as a result of file handling.

For details about file handling exceptions, see the error messages in the range 5000 through 5099 in "Error Messages and System Messages", in the **JADEMsgs.pdf** file.

For details about the methods defined in the **FileException** class, see "FileException Methods", in the following subsection.

Inherits From: NormalException

Inherited By: (None)

FileException Methods

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

Method	Returns an instance of the …
file	**File** or **FileFolder** class that was being used when the exception was raised
fileNode	**FileNode** class that was being used when the file exception was raised

file

Signature `file(): File;`

The **file** method of the **FileException** class returns a reference to the **File** or **FileFolder** instance that was being used when the exception was raised.

To preserve compatibility with existing application calls, this method returns a reference to a **FileFolder** instance if the exception was raised when using a **FileFolder** instance.

An exception is raised if the file cannot be opened; for example, if the file name is invalid.

fileNode

Signature `fileNode(): FileNode;`

The **fileNode** method of the **FileException** class returns a reference to the **FileNode** instance that was being used when the exception was raised.

FileFolder Class

The **FileFolder** class provides access to a collection of files or subdirectories on a specified file system folder or directory.

In JADE thin client mode, instances of the **FileFolder** class refer to folders on the presentation client when the **FileNode** class **usePresentationFileSystem** property is set to **true** and the instances are *not* shared transient instances.

Shared transient instances of the **FileFolder** class are always processed on the application server, and the setting of the **usePresentationFileSystem** property is ignored. For shared transient instances of the **FileFolder** class, the **FileNode** class **usePresentationFileSystem** property defaults to **false**. (A folder opened on one presentation client cannot be accessed by another client.)

For details about the property and methods defined in the **FileFolder** class, see "FileFolder Property" and "FileFolder Methods", in the following subsections.

Inherits From: FileNode

Inherited By: (None)

FileFolder Property

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

Property	Description
mask	Contains the masking string that is used to access the files in the **FileFolder** object

mask

Type: String

The **mask** property of the **FileFolder** class contains the masking string that is used to access the files in the **FileFolder** object returned from the **files** method. The default value is **"*.*"**.

To specify multiple masks, separate them using the vertical bar (**|**) character; for example, **"*.txt | *.log | *.cat"**.

Note Specifying **"*.*"** as a mask value results in all files and subfolders in the folder being returned by the **FileFolder** class **files** method.

FileFolder Methods

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

Method	Description
browseForAppServerFolder	Returns the path of the selected folder on the application server node
browseForFolder	Returns the path of the selected folder from your local standard or thin client
browseForServerFolder	Returns the path of the selected folder on the database server node
files	Returns an array of all files in the folder

Method	Description
isAvailable	Returns **true** if the associated folder is available
isValidPathName	Returns **true** if the specified path name is valid
make	Creates a folder on the file system
purge	Deletes the folder from disk
rename	Changes the name of the folder

browseForAppServerFolder

Signature `browseForAppServerFolder(description: String; startFolder: String): String;`

The **browseForAppServerFolder** method of the **FileFolder** class displays a file dialog and returns the folder from the specified application server (or standard client) node. The value specified in the **description** parameter is displayed in the dialog. The search for the folder starts in the folder (directory) specified in the **startFolder** parameter.

The following example shows the use of the **browseForAppServerFolder** method.

```
vars
    fileFolder : FileFolder;
    dir        : String;
begin
    create fileFolder;
    dir := fileFolder.browseForAppServerFolder("Select Data Directory", "");
    if dir <> null then
        dataDir.text := dir;
    endif;
epilog
    delete fileFolder;
end;
```

See also the **FileFolder** class **browseForFolder** method, which enables you to browse for and return a folder on a local disk when running in thin client mode.

browseForFolder

Signature `browseForFolder(caption: String; startFolder: String): String;`

The **browseForFolder** method of the **FileFolder** class displays the common File dialog and returns the folder whose name is specified in the **caption** parameter. The search for the folder starts in the folder (directory) specified in the **startFolder** parameter.

The **Folder** text box is displayed below the directory list box, so that the user can specify the name of the required folder, using a standard (common) Microsoft dialog.

Note In thin client mode, this method always runs on the presentation client.

The method shown in the following example initializes the database from text files.

```
vars
    dataLoader : InitialDataLoader;
```

```
    fileFolder : FileFolder;
    dirPath    : String;
begin
    // Ask for the directory containing the initial data files
    create fileFolder transient;
    dirPath := fileFolder.browseForFolder("Select data file directory",
                                          app.dbPath);
    if dirPath <> null then
        // Create the data loader and initialize the database
        create dataLoader transient;
        dataLoader.loadData(dirPath);
    endif;
epilog
    delete dataLoader;     // does nothing if dataLoader is null
    delete fileFolder;     // does nothing if fileFolder is null
end;
```

See also the **FileFolder** class **browseForServerFolder** method, which enables you to browse for and return a folder from the database server node.

browseForServerFolder

Signature `browseForServerFolder(description: String;
startFolder: String): String;`

The **browseForServerFolder** method of the **FileFolder** class displays a file dialog and returns the folder from the specified database server node. The value specified in the **description** parameter is displayed in the dialog. The search for the folder starts in the folder (directory) specified in the **startFolder** parameter.

The following example shows the use of the **browseForServerFolder** method.

```
vars
    fileFolder : FileFolder;
    dir        : String;
begin
    create fileFolder;
    dir := fileFolder.browseForServerFolder("Select Backup Directory", "");
    if dir <> null then
        backDir.text := dir;
    endif;
epilog
    delete fileFolder;
end;
```

See also the **FileFolder** class **browseForFolder** method, which enables you to browse for and return a folder from your local (client) node.

files

Signature `files(): FileNodeArray updating;`

The **files** method of the **FileFolder** class returns a reference to an array of all files that are contained in the folder that match any of the values specified in the **mask** property. The order of files within the array depends on the operating system (for example, platform files are returned in alphabetic order), but the files are grouped according to the list of mask values, because a separate pass over the folder's files is required for each mask value.

Note Specifying **"*.*"** in the **mask** property results in all files and subfolders in the folder being returned.

isAvailable

Signature `isAvailable(): Boolean;`

The **isAvailable** method of the **FileFolder** class returns **true** if the file folder exists.

The following example shows the use of the **isAvailable** method.

```
vars
   diskFolderObj : FileFolder;
begin
   create diskFolderObj transient;
   diskFolderObj.fileName := "E:\";
   if diskFolderObj.isAvailable() then
       write true;
   else
       write false;
   endif;
end;
```

isValidPathName

Signature `isValidPathName(name: String): Boolean;`

The **isValidPathName** method of the **FileFolder** class returns **true** if the path specified in the name parameter is a valid path name.

The code fragment in the following example shows the use of the **isValidPathName** method.

```
if not myFolder.isValidPathName(dirName) then
    return false;
endif;
```

make

Signature `make();`

The **make** method of the **FileFolder** class creates the specified file folder. If the **FileNode** class **usePresentationFileSystem** property is set to **true** and the trailing directory separator is not present, the **make** method adds it.

Note Do not end a file or directory name with a trailing space or a period. (Although the underlying file system may support this, the operating system may not.)

purge

Signature `purge();`

The **purge** method of the **FileFolder** class deletes the specified file folder from disk.

rename

Signature `rename(newname: String);`

The **rename** method of the **FileFolder** class changes the name of the folder associated with the file folder object to the value specified in the **newname** parameter.

Notes This method does not change the value of the **FileNode** class **fileName** property.

The code fragment in the following example shows the use of the **rename** method.

```
myfolder.rename("c:\Old");
```

JADE attempts to rename folders across devices. This attempt could fail for various reasons (often related to an operating system restriction), as follows.

- Folder name on the output device is invalid
- Permission to create a folder on the output device is denied
- Space on the output device is insufficient

FileNode Class

The **FileNode** class is an abstract class that contains the properties and methods common to the **File** class and **FileFolder** class. For details about the properties and methods defined in the **FileNode** class, see "FileNode Properties" and "FileNode Methods", in the following subsections.

Inherits From: Object

Inherited By: File, FileFolder

FileNode Properties

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

Property	Description
allowCreate	Specifies if the file can be created
archive	Specifies if the **archive** flag is set
createdTime	Creation time of the file or folder
fileName	Contains the full path name of the file
hidden	Specifies if the **hidden** flag is set
lastAccessedTime	Last accessed time of the file or folder
lastModifiedTime	Last modified time of the file or folder
name	Contains the file or directory associated with the **FileNode** instance
readOnly	Specifies if the **read only** flag is set
systemFile	Specifies if the **system** flag is set
usePresentationFileSystem	Specifies whether methods in the **File** and **FileFolder** class are processed on the application server or presentation client when the receiver is running in JADE thin client mode

allowCreate

Type: Boolean

Set the **allowCreate** property of the **FileNode** class to **false** before the file is opened to specify that the file cannot be created. The default value is **true**; that is, the file can be created if it does not exist.

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

```
vars
    fileName : String;
    file     : File;
    count    : Integer;
begin
    count := 0;
    create file transient;
    file.mode         := File.Mode_IO;
    file.kind         := File.Kind_Unknown_Text;
    file.allowReplace := false;
```

```
    file.allowCreate  := false;
    file.open(fileName);
    ...           // do some processing here
end;
```

When this property is set to **false** and the file does not exist, an exception is raised. For example, if the **mode** property has a value of **File.Mode_Output**, **File.Mode_IO**, or **File.Mode_Append** and the file is missing, the file is not created.

Note This property applies only to the **File** class. It is ignored when set for the **FileFolder** class.

archive

Type: Boolean

The **archive** property of the **FileNode** class specifies if the **archive** flag is set.

Note This property applies only when the file exists.

createdTime

Type: TimeStamp

The **createdTime** property of the **FileNode** class contains the time that the file or folder was created.

fileName

Type: String

The **fileName** property of the **FileNode** class contains the fully qualified path and name of the file or folder. If the full path is not specified, the current default directory is assumed.

The following code fragments show the use of the **fileName** property.

```
file1.fileName := "C:\temp\myTempFile";

dir1.fileName := "C:\";     // The disk designator "C:" needs a backslash "C:\"
                            // to be a fully qualified path
```

Note You must set this property before a file is opened.

hidden

Type: Boolean

The **hidden** property of the **FileNode** class specifies if the **hidden** flag is set.

Note This property applies only when the file exists.

lastAccessedTime

Type: TimeStamp

The **lastAccessedTime** property of the **FileNode** class contains the time that the file or folder was last accessed.

lastModifiedTime

Type: TimeStamp

The **lastModifiedTime** property of the **FileNode** class contains the time that the file or folder was last modified.

name

Type: String

The read-only **name** property of the **FileNode** class contains the name of the file or directory associated with the **FileNode** instance. The name is the bottom level "node" in a directory or file path name.

This property is valid only for file nodes that are retrieved by using the **FileFolder** class **files** method.

readOnly

Type: Boolean

The **readOnly** property of the **FileNode** class specifies if the **read only** flag is set. The default value for file creation is **false**; that is, files can be written to or they can be read.

Note This property applies only when the file exists.

systemFile

Type: Boolean

The **systemFile** property of the **FileNode** class specifies if the **system** flag is set.

Note This property applies only when the file exists.

usePresentationFileSystem

Type: Boolean

The **usePresentationFileSystem** property of the **FileNode** class specifies whether methods for this class are 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 property value has no effect.

For non-shared transient instance when running in JADE thin client mode, the **usePresentationFileSystem** property defaults to **true**. Set it 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.

Shared transient instances of the **File** and **FileFolder** class are always processed on the application server, regardless of the setting of this property. (For shared transient instances of these classes, the **usePresentationFileSystem** property defaults to **false**. A file or a folder opened on one presentation client cannot be accessed by another client.)

FileNode Methods

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

Method	Description
directorySeparator	Returns a string containing the directory separator of the receiver
isAvailable	Specifies if the associated file of the receiver is available
purge	Deletes the specified file from disk

directorySeparator

Signature `directorySeparator(): String;`

The **directorySeparator** method of the **FileNode** class returns a string containing the directory separator of the receiver; that is, the backslash character **"\"** is returned in a Microsoft Windows operating system.

isAvailable

Signature `isAvailable(): Boolean;`

The **isAvailable** method of the **FileNode** class returns **true** if the **File** or **FileFolder** class instance is available; that is, if the file exists and it can be accessed with the requested mode. (For details, see the **File** class **mode** property.)

The code fragment in the following example shows the use of the **isAvailable** method.

```
create file;
file.fileName := myFileOpen.fileName;
file.mode     := File.Mode_Input;
if file.isAvailable then
    file.open;
    beginTransaction;
else ...             // do some processing here
```

purge

Signature `purge();`

The **purge** method of the **FileNode** class deletes the specified **File** or **FileFolder** class instance from disk.

FileNodeArray Class

The **FileNodeArray** class is the transient class that encapsulates behavior required to access **FileNode** objects in an array.

The file nodes are referenced by their position in the collection.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a file node array.

Inherits From: ObjectArray

Inherited By: (None)

Global Class

The **Global** class provides a means by which application-specific data can be shared among users of an application.

A subclass of **Global** (with a default name of ***Gapplication-class-name***, which can be overridden at subschema creation time) is created whenever you create a new subschema. A single instance of the ***Gapplication-class-name*** class is also created. It is your responsibility to declare properties and methods for a **Global** subclass.

At run time, a single instance of the ***Gapplication-class-name*** subclass is shared by any active applications defined for the subschema. You can refer to this single instance in your logic, by using the **global** system variable. Because the **global** object is persistent, any updates to properties of **global** must be made in transaction state.

Note As all users of an application share an instance of the **global** object, if you update this object frequently, it may severely affect runtime performance.

You can use the **getAndValidateUser** and **isUserValid** methods to apply user validation for your applications. (For more details, see "User-Validation Support", in Chapter 2 of the *JADE Object Manager Guide*.) For details about the methods and event defined in the **Global** class, see "Global Methods" and "Global Event", in the following subsections.

Inherits From: Object

Inherited By: RootSchemaGlobal

Global Methods

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

Method	Description
alert	Plays a waveform sound
beep	Plays a waveform sound
getAndValidateUser	Gets and validates user codes and passwords
isUserValid	Validates the user code and password
isValidObject	Establishes if the object specified in the **obj** parameter exists
jadeReportWriterSystemName	Returns a string containing the name of the JADE system
lockExceptionHandler	Initiates the global lock exception handler to retry a lock operation

alert

Signature `alert(soundName: Integer);`

The **alert** method of the **Global** class plays the waveform sound specified in the **soundName** parameter.

In JADE thin client mode, this method always executes on the presentation client.

The waveform sound for each sound type is identified by an entry in the **Sounds** section of the registry.

Note Assign sounds to system events by using the **Sounds and Multimedia** program item of the standard Windows Control Panel.

You can use the **Sounds** category global constant values, listed in the following table, in the **soundName** parameter.

Global Constant	Integer Value
Snd_Asterisk	#40
Snd_Beep	-1
Snd_Default	0
Snd_Exclamation	#30
Snd_Hand	#10
Snd_Question	#20

beep

Signature `beep();`

The **beep** method of the **Global** class plays the .**wav** file associated with the Default Beep option (specified in the **Sound Events** list box on the **Sounds** sheet of the Sounds and Multimedia Properties dialog accessed by using the **Sounds and Multimedia** program item of the standard Windows Control Panel) of the current locale.

Use this method to sound the beep at the workstation of the user who invoked the method.

Note The beep alert is sounded on a workstation regardless of whether a sound card is installed.

getAndValidateUser

Signature
```
getAndValidateUser(usercode: String output;
                   password: String output): Boolean;
```

The **getAndValidateUser** primary validation method of the **Global** class and its associated protocol enables you to provide a customized user logon form and associated validation logic. Your validate method can create forms and it can access and update the database.

Use this method to handle the end-user interaction required to solicit a user code and optional password from the end-user and to perform any primary validation and retry logic.

When you use this method with the **isUserValid** method, you should return the password in encrypted form.

The **isUserValid** method (or secondary validation) then becomes a "rubber stamp" action that revalidates the user code and the already encrypted password, preferably at the server. (For details, see "User-Validation Support" and "Opening a Process", in Chapters 2 and 3, respectively, of the *JADE Object Manager Guide*.)

The **getAndValidateUser** method *must* return a user code at a minimum. This user code is passed by the JADE Object Manager to server nodes.

The **getAndValidateUser** method has no input parameters. The output parameters are listed in the following table.

Parameter	Description
usercode	User-supplied user code
password	User-supplied password

As the **usercode** output parameter is the **signOnUserCode** or **userCode** property of the **Process** class instance, you do not need to code a method that provides the ability to change user codes.

The values returned by this method are listed in the following table.

Value	Result
true	The user is authorized
false	The user is not authorized

When no user code is supplied in the **jomSignOn** Application Programming Interface (API) call to open a process, the JADE Object Manager invokes the **getAndValidateUser** method on the subschema global instance, as follows.

- If the method indicates success, the **jomSignOn** API proceeds to secondary validation by using the **isUserValid** method.
- If the method signals failure (that is, the user is not authorized), the JADE Object Manager discontinues the tentative application process and a **null** process handle is returned to the caller. No exception is raised; error handling and reporting are left to your user method.

The default **getAndValidateUser** method is defined and implemented in the **RootSchemaGlobal** global class of the Root Schema. The default implementation manufactures a user code that consists of the workstation name that has a suffix of the operating system process ID, and it returns this user code, a **null** password, and a result of **true**.

The default method definition also defines the method name and signature so that you are aware that you are correctly reimplementing the method.

isUserValid

Signature
```
isUserValid(usercode: String;
            password: String): Boolean [serverExecution];
```

The **isUserValid** method of the **Global** class is the secondary validation method that is invoked on the subschema global instance as a result of a **jomSignOn** API call to open a process.

For details, see "User-Validation Support" and "Opening a Process", in Chapters 2 and 3, respectively, of the *JADE Object Manager Guide*.

When no user code is supplied in the **jomSignOn** Application Programming Interface (API) call to open a process, the user code and password returned by the **getAndValidateUser** method are used.

If the user code and password are specified in the **jomSignOn** call, these are used, as follows.

- If the method indicates success, the **jomSignOn** API returns a valid process handle to the caller, which allows the application to proceed as usual.
- If the method signals failure (that is, the user is not authorized), the JADE Object Manager discontinues the tentative application process and a **null** process handle is returned to the caller. No exception is raised.

When this method is reimplemented, your user method is responsible for validating (or revalidating) the user code and password and for returning the appropriate result.

Reimplement this method to:

- Validate user codes and passwords passed from non-JADE clients or the JADE ODBC interface.
- Ensure that *secondary* validation always occurs on the server.

  To guarantee that secondary validation occurs on the server, the **isUserValid** method must be marked for server execution.

The values returned by this method are listed in the following table.

Value	Result
true	The user is authorized
false	The user is not authorized

The default **isUserValid** method that returns **true** is defined and implemented in the **RootSchemaGlobal** global class of the RootSchema. The default implementation also defines the method name and signature, so that you are aware that you are correctly reimplementing the server user validation method.

isValidObject

Signature `isValidObject(obj: Object): Boolean;`

The **isValidObject** method of the **Global** class is used to establish if the object specified in the **obj** parameter exists, by returning **true**.

This method returns **false** if the specified object has been deleted.

jadeReportWriterSystemName

Signature `jadeReportWriterSystemName(): String;`

The **jadeReportWriterSystemName** method of the **Global** class is called by the JADE Report Writer Designer application to return the name of the system.

Although the default return value is null (that is, **""**), you can reimplement this method in your user schemas to return any value that you require. For example, you can use this method to return an overall system identifier (variable) for use in report or page headers, as shown in the following example.

```
jadeReportWriterSytemName(): String;
vars
begin
    return "My Test Company":
end;
```

lockExceptionHandler

Signature `lockExceptionHandler(le: LockException io): Integer;`

The **lockExceptionHandler** method of the **Global** class is an exception handler that can be used to retry a lock operation.

The following example shows the use of this method to arm the exception handler.

```
buttonRead_click(btn: Button input) updating;
vars
```

```
    cust         : Customer;
    customerName : String;
begin
    // Initiate the exception for locking errors
    on LockException do global.lockExceptionHandler(exception);
    // Define the customer object that has been selected
    if listBoxCustomer1.listIndex > 0 then
       cust :=
           listBoxCustomer1.itemObject[listBoxCustomer1.listIndex].Customer;
       customerName := cust.name;
    else
       app.msgBox("Please select a Customer", "No Customer Selected",
                  MsgBox_OK_Only);
       return;
    endif;
end;
```

Note Each process can have up to 128 global exception handlers armed at any one time.

Global Event

The event defined in the **Global** class is summarized in the following table.

Event	Description
initialize	Performs any function common to all application users for this database

initialize

Signature `initialize() updating;`

Definition of the **initialize** event of the **Global** class is optional. If the **initialize** event is defined, it is automatically called by the application before the **initialize** event of the application is called and before the start-up form of the application is invoked.

This event can perform any function that is common to all users of this application defined for this JADE database.

Note The **initialize** event is performed once for each user of the application.

If the event creates a form and does not subsequently unload it, the start-up form of the application is not invoked.

GUIClass Class

The **GUIClass** class, a subclass of the **Class** class, is the metaclass of all JADE Graphical User Interface (GUI) classes and contains the definition of all GUI classes.

For details about handling class instances, see "Caveat When Handling Persistent Class Instances" and "Caveat When Handling Shared Transient Class Instances" under "Class Class", earlier in this chapter.

For details about the method defined in the **GUIClass** class, see "GUIClass Method", in the following subsection.

Inherits From: Class

Inherited By: ActiveXGUIClass

GUIClass Method

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

Method	Description
createPrintForm	Creates a print form at run time

createPrintForm

Signature `createPrintForm(): InstanceType updating;`

The **createPrintForm** method of the **GUIClass** class creates a print form at run time, as shown in the following example.

```
vars
    pform : PrintForm;
begin
    pform := PrintForm.createPrintForm;
    ...                                 // do some processing here
end;
```

When you use this method to create a form:

- No actual windows are created, except for **Ocx**, **OleControl**, **ActiveXControl**, and **MultiMedia** controls
- The entire GUI process is simulated (as it is with JADE Web forms)

The form and controls therefore have no window handle (that is, no value for the **hwnd** property inherited from the **Window** class) and cannot be accessed using any Windows external Application Programming Interface (API). Requests to show the form are also rejected.

Use the **create** ***print-form*** syntax to create GUI windows.

Using the **createPrintForm** method compared with using the **create** instruction is irrelevant to the print output that is produced. The only impact is the reduction in the amount of GUI windows resources that are used.

Tip Use the **createPrintForm** method to create 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).

HugeStringArray Class

The **HugeStringArray** class is an ordered collection of large objects of type **String**; that is, **String** objects with a length in the range **0** through **2,047** characters.

Huge strings, with a membership of **2,047** characters, are referenced by their position in the collection.

Huge string arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a huge string array.

Inherits From: Array

Inherited By: (None)

IDispatch Class

The **IDispatch** class is the abstract class that provides a superclass for all ActiveX control and automation interface classes created when you import an ActiveX control or automation type library into JADE.

A class is generated as a subclass of the **IDispatch** class for each required ActiveX interface. Each of these generated classes has properties and methods added to it that map to the properties and methods of the interface. The properties (using mapping methods) and methods call an external method that updates the object.

To use an event interface, you must register your interest in a specific event, which involves specifying the event method of the interface and the method that you want executed when the event is triggered. (For details, see the **beginNotifyAutomationEvent** method.)

When you create an instance (that is, a subclass) of the JADE **ActiveXAutomation** or **ActiveXControl** class that corresponds to the ActiveX object that you want to use in JADE, a transient instance of the default interface is also created as a subclass of the **IDispatch** class. A reference is established between the ActiveX class and its default interface.

When you call JADE ActiveX class methods (as you do for any other JADE class), these method calls are passed by the default interface to the actual ActiveX object. If the ActiveX object returns a reference to another interface in response to a method call or the getting of a property, JADE creates an instance of the corresponding JADE interface class and returns a reference to that instance instead.

Note You can create neither transient nor persistent instances of the **IDispatch** class. Instances of this class are created by JADE at run time when an ActiveX object supplies an interface pointer and they are deleted when the ActiveX object that is using them is deleted.

Use the **getInterface** method defined in the **ActiveXAutomation** and **ActiveXControl** classes to access any interface for an imported ActiveX object.

The **IJadeAutoFont**, **IJadeAutoFontEvents**, and **IJadeAutoPicture** standard interface classes were created as subclasses of the **IDispatch** class when the OLE Automation library that has been preloaded into the **RootSchema** was imported.

For details about:

- The methods defined in the **IDispatch** class, see "IDispatch Methods", in the following subsection.
- **IUnknown** class and the **ActiveXInterface** class, see "IUnknown Class" and "ActiveXInterface Class", elsewhere in this chapter.
- Using ActiveX Control and Automation Server Libraries, see Chapter 11 of the *JADE External Interface Developer's Reference*.
- The **OLE_Automation** object preloaded into JADE, see "ActiveXAutomation Class", earlier in this chapter.
- The methods and properties defined in the **OLE_Automation** class and the **IJadeAutoFont**, **IJadeAutoFontEvents**, and **IJadeAutoPicture** subclasses, refer to your COM documentation.

Inherits From: IUnknown

Inherited By: **IJadeAutoFont**, **IJadeAutoFontEvents**, and **IJadeAutoPicture** classes preloaded into the **RootSchema** with the **OLE_Automation** object class, ActiveX control and automation object classes imported by developers

IDispatch Methods

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

Method	Description
beginNotifyAutomationEvent	Registers the receiver to be notified when an event occurs on an ActiveX automation object
endNotifyAutomationEvent	Terminates a previous **beginNotifyAutomationEvent** event
getInterface	Returns the specified ActiveX interface, if it exists

The JADE methods that are defined in the **IJadeAutoPicture** class preloaded into the **RootSchema** with the OLE Automation object are summarized in the following table.

Method	Description
loadPicture	Creates a picture object from an external file
makePicture	Creates a picture object from a JADE binary
savePicture	Saves the image of a picture to the specified external file

beginNotifyAutomationEvent

Signature

```
beginNotifyAutomationEvent(receiver:          Object;
                           eventClassRefName: String) updating;
```

The **beginNotifyAutomationEvent** method of the **IDispatch** class registers the receiver to be notified when a specified event occurs on an ActiveX automation object.

The object that invokes the **beginNotifyAutomationEvent** is referred to as the *subscriber*. An object that subscribes to an automation notification is notified when the nominated event occurs for that object.

The parameters for this method are listed in the following table.

Parameter	Description
receiver	The object that is to receive the event notification
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

A method implemented by the **eventClassRefName** parameter is executed each time its corresponding automation event occurs.

This event notification continues until the JADE automation object is deleted or until the **endNotifyAutomationEvent** method (which has the same signature as this **beginNotifyAutomationEvent** method) is called.

Caution There may be an impact on performance, particularly in JADE thin client mode or on a slow communications link, if you register for large numbers of automation events or events that are triggered often; for example, a cell change event in the Excel automation type library. (For details about achieving maximum performance in the JADE thin client mode of operation, see "JADE Thin Client Performance Considerations", in Appendix A of the *JADE Thin Client Guide*.)

For more details about automation events, see "Using Automation Events", in Chapter 4 of the *JADE External Interface Developer's Reference*.

endNotifyAutomationEvent

Signature `endNotifyAutomationEvent(receiver: Object; eventClassRefName: String);`

The **endNotifyAutomationEvent** method of the **IDispatch** class terminates a previous **beginNotifyAutomationEvent** method. The parameters for this method, listed in the following table, must be the same as the parameters specified in the **beginNotifyAutomationEvent** method.

Parameter	Description
receiver	The object that is to receive the event notification
eventClassRefName	The name of the reference (an instance of the **IDispatch** subclass) that implements the notification events

For more details about automation events, see "Using Automation Events", in Chapter 4 of the *JADE External Interface Developer's Reference*.

getInterface

Signature `getInterface(interfaceClass: Class): IDispatch;`

The **getInterface** method of the **IDispatch** class returns the ActiveX class specified in the **interfaceClass** parameter, if it exists. (As ActiveX interfaces are created as subclasses of the **IDispatch** class when an ActiveX type library is imported, use the Class List of the Class Browser to obtain the names of ActiveX interfaces, if required.)

If the specified class does not exist, a **null** value is returned.

loadPicture

Signature `loadPicture(fileName: String): IJadeAutoPicture;`

The **loadPicture** method of the **IDispatch** class creates a picture object from the external file specified in the **fileName** parameter, which can be a valid file name or it can be the fully qualified path and name of a valid picture file.

If the specified file does not exist or you specify an invalid file name, an exception is raised.

The **IJadeAutoPicture** class is a standard subclass of the **OLE_Automation** class, which was created as a subclass of the **IDispatch** class when the OLE automation library was preloaded into the JADE **RootSchema**.

makePicture

Signature `makePicture(binary: Binary): IJadeAutoPicture;`

The **makePicture** method of the **IDispatch** class creates a picture object from the JADE binary specified in the **binary** parameter.

If the binary value represents an invalid picture, an exception is raised.

The **IJadeAutoPicture** class is a standard subclass of the **OLE_Automation** class, which was created as a subclass of the **IDispatch** class when the OLE automation library was preloaded into the JADE **RootSchema**.

savePicture

Signature `savePicture(filename: String);`

The **savePicture** method of the **IDispatch** class saves the image of a picture to the external file specified in the **filename** parameter, which can be a valid file name or it can be the fully qualified path and name of a valid picture file.

If the picture is unable to be saved (for example, you specify an invalid file name), an exception is raised.

The **IJadeAutoPicture** class is a standard subclass of the **OLE_Automation** class, which was created as a subclass of the **IDispatch** class when the OLE automation library was preloaded into the JADE **RootSchema**.

IDispatchArray Class

The **IDispatchArray** class is the transient class that encapsulates behavior required to access **IDispatch** objects (that is, ActiveX interface classes) in an array.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a database file array.

Inherits From: ObjectArray

Inherited By: (None)

Integer64Array Class

The **Integer64Array** class is an ordered collection of **Integer64** values in which the values are referenced by their position in the collection.

Integer64 arrays inherit the methods defined in the **Array** class.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from an Integer64 array.

For details about the methods defined in the **Integer64Array** class, see "Integer64Array Methods", in the following section.

Inherits From: Array

Inherited By: (None)

Integer64Array Methods

The methods defined in the **Integer64Array** class are summarized in the following table.

Method	Description
binarySearch	Specifies whether the element exists at the position specified by an **Integer** value
binarySearch64	Specifies whether the element exists at the position specified by an **Integer64** value

binarySearch

Signature
```
binarySearch(search: Integer64;
             index:  Integer io): Boolean;
```

The **binarySearch** method of the **Integer64Array** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter if found, or to the position at which it should be added if it does not exist.

This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch** method.

```
if not bigNumbers.includes(num) then
    bigNumbers.binarySearch(num, pos);
    bigNumbers.insert(pos + 1, num);
endif;
```

Note Use the **binarySearch64** method instead of the **binarySearch** method, if the number of entries in the array could exceed the maximum integer value of 2,147,483,647.

binarySearch64

Signature
```
binarySearch64(search: Integer64;
               index:  Integer64 io): Boolean;
```

The **binarySearch64** method of the **Integer64Array** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter as an **Integer64** value if found or to the position at which it should be added if it does not exist.

This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch64** method.

```
if not bigNumbers.includes(num) then
    bigNumbers.binarySearch64(num, pos);
    bigNumbers.insert(pos + 1, num);
endif;
```

IntegerArray Class

The **IntegerArray** class is an ordered collection of **Integer** values in which the values are referenced by their position in the collection. Bracket (**[]**) subscript operators enable you to assign values to and receive values from an Integer array.

Integer arrays inherit the methods defined in the **Array** class.

For details about the methods defined in the **IntegerArray** class, see "IntegerArray Methods", in the following section.

Inherits From: Array

Inherited By: (None)

IntegerArray Methods

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

Method	Description
binarySearch	Specifies whether the element exists at the position specified by an **Integer** value
binarySearch64	Specifies whether the element exists at the position specified by an **Integer64** value

binarySearch

Signature
```
binarySearch(search: Integer;
             index:  Integer io): Boolean;
```

The **binarySearch** method of the **IntegerArray** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter if found or to the position at which it should be added if it does not exist.

This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch** method.

```
if not rowPositions.includes(top) then
    rowPositions.binarySearch(top, pos);
    rowPositions.insert(pos + 1, top);
endif;
```

Note Use the **binarySearch64** method instead of the **binarySearch** method, if the number of entries in the array could exceed the maximum integer value of 2,147,483,647.

binarySearch64

Signature
```
binarySearch64(search: Integer;
               index:  Integer64 io): Boolean;
```

The **binarySearch64** method of the **IntegerArray** class sets the **index** parameter to the position in the array of the element specified in the **search** parameter if found or to the position at which it should be added if it does not exist.

This method returns **true** if another specified element is located. If no element is found, this method returns **false** and places the position in the array at which the element should be added in the **index** parameter.

The code fragment in the following example shows the use of the **binarySearch64** method.

```
if not rowPositions.includes(top) then
    rowPositions.binarySearch64(top, pos);
    rowPositions.insert(pos + 1, top);
endif;
```

IntegrityViolation Class

The **IntegrityViolation** class is reserved for future use as the transient class that defines the behavior of exceptions raised as a result of integrity rule violations.

Inherits From: SystemException

Inherited By: (None)

InternetPipe Class

The **InternetPipe** class, a subclass of the **NamedPipe** class, provides an interface for communicating with JADE applications from the Internet through an Internet server.

Note This class is available only under a Windows operating system that supports services. To access your JADE applications from the Internet, the JADE server node and the workstation running the JADE application must be running a Windows operating system that supports services.

To communicate with the **jadehttp** library file on the Internet server using the pipe channel, the JADE application creates a transient instance of the **InternetPipe** class and then offers the named pipe for opening with the name of the JADE application. When the pipe is connected, it waits for Internet requests to be sent over the pipe. If no named pipe is open, the Internet user is advised that the service is not available.

Multiple instances of the pipe can be opened from the same application or by running multiple copies of the JADE application from the same **jade.exe** executable program, where each application opens the same pipe name.

For details about the methods defined in the **InternetPipe** class, see "InternetPipe Methods", in the following subsection.

Inherits From: NamedPipe

Inherited By: (None)

InternetPipe Methods

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

Method	Description
openPipeCallback	Initiates an asynchronous read of the opened pipe
readPipeCallback	Performs Web session evaluation processing
sendReply	Sends the formatted HyperText Markup Language (HTML) page to the opened pipe

openPipeCallback

Signature `openPipeCallback(pipe: InternetPipe) updating;`

The **openPipeCallback** method of the **InternetPipe** class is called when the **jadehttp** library file opens the Internet server end of the pipe, to initiate an asynchronous read of the opened pipe.

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.

readPipeCallback

Signature
```
readPipeCallback(pipe: InternetPipe;
                 msg:  Binary) updating;
```

The **readPipeCallback** method of the **InternetPipe** class is called when data is available on the pipe, to perform Web session evaluation processing.

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.

sendReply

Signature `sendReply(html: Binary) updating;`

The **sendReply** method of the **InternetPipe** class sends the formatted HyperText Markup Language (HTML) page back to the opened pipe and starts the next read request.

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.

Iterator Class

The **Iterator** class encapsulates the behavior required to sequentially access elements of a collection.

Instances of the **Iterator** class are referred to as *iterators*. Use iterators to iterate two or more collections where the iterations are not nested or when the state of an iteration (that is, the position in a collection) needs to be remembered and the iteration resumed later. (The **foreach** instruction enables you to iterate through two collections simultaneously only if one iteration is nested within the other.) The order in which instances are returned when iterating a virtual collection is not significant.

Create an iterator by using the **createIterator** method of the **Collection** class. (Instances of the **Dictionary** class provide methods that enable you to specify the start position of an iterator.)

For details about the methods defined in the **Iterator** class, see "Iterator Methods", in the following subsection.

Inherits From: Object

Inherited By: ArrayIterator, DictIterator, ExternalIterator, MergeIterator, SetIterator, SetMergeIterator

Iterator Methods

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

Method	Description
back	Accesses entries in reverse order in the collection to which the iteration is attached
current	Returns the last value iterated by the **back** or **next** method
excludeOfflineObjects	Specifies whether objects stored in offline partitions should be excluded from the iteration
getCollection	Returns the collection associated with the receiver
getCurrentKey	Retrieves a single key from a dictionary while iterating through the dictionary
getCurrentKeys	Retrieves keys from a dictionary while iterating through the dictionary
isValid	Returns **true** if the receiver is a valid iterator
next	Accesses successive entries in the collection to which the iteration is attached
reset	Initializes an iterator
startAtIndex	Sets the starting position of the iterator to a relative index in the attached collection
startAtObject	Sets the starting position of the iterator at the position of the specified object
startNearIndex	Sets the starting position of the iterator in the attached collection approximate to a relative index

back

Signature `back(value: Any output): Boolean updating;`

The **back** method of the **Iterator** class accesses entries in reverse order one at a time in the collection to which the iteration is attached. This method returns **true** when it has returned a value or it returns **false** when the iterator is positioned in front of the first entry in the collection.

The **value** parameter receives the prior entry in the collection and must be of the same type as the members in the collection. The following example shows the use of the **back** method.

```
getRelativePosition(pObj: Object): Integer;
vars
    coll : Collection;
    pos  : Integer;
    obj  : Object;
    iter : Iterator;
begin
    coll := self.getCollection;
    iter := coll.createIterator;
    pos  := coll.size;
    while iter.back(obj) do
        pos := pos - 1;
        if obj = pObj then
            break;
        endif;
    endwhile;
    return pos;
end;
```

current

Signature `current(value: Any output): Boolean;`

The **current** method of the **Iterator** class returns the last value iterated by using the **next** or **back** method.

The **current** method returns **true** if the iterator is positioned on an entry in the collection, or it returns **false** if the iterator is reset or it is positioned beyond the start or end of the collection.

The **value** parameter receives the entry of the current iterator position in the collection and must be of the same type as the members in the collection.

excludeOfflineObjects

Signature `excludeOfflineObjects(enable: Boolean): Boolean updating;`

The **excludeOfflineObjects** method of the **Iterator** class, when called with the value of the **enable** parameter set to **true**, specifies that the receiver is to exclude objects stored in offline partitions from the iteration and takes effect on the next call to the **next** or **back** method.

The method returns the prior exclusion state, which user logic can restore, if required, when calls are nested.

getCollection

Signature `getCollection(): Collection;`

The **getCollection** method of the **Iterator** class returns a reference to the collection associated with the receiver. A **null** value is returned if no collection is associated with the receiver.

The code fragments in the following examples show the use of the **getCollection** method.

```
if iter.getCollection = null then
    app.msgBox("No collection defined", "Error", MsgBox_OK_Only);
endif;
```

```
lock(iter.getCollection, Share_Lock, Transaction_Duration, 1000);
```

getCurrentKey

Signature `getCurrentKey(ordinal: Integer): Any;`

The **getCurrentKey** method of the **Iterator** class is an abstract method that is implemented only by dictionary iterators.

This method retrieves the keys from an iterator while iterating through the dictionary and returns the value of a single key at the current position of the iterator in the associated dictionary.

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 iterator without having to access the member object itself.

The **ordinal** parameter specifies the relative key by ordinal position of the key 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.)

This method can be used as an alternative to the **getIteratorKeys** of the **Dictionary** class to avoid share locking the associated dictionary on each call.

getCurrentKeys

Signature `getCurrentKeys(keys: ParamListType output);`

The **getCurrentKeys** method of the **Iterator** class is an abstract method that is implemented only by dictionary iterators.

This method retrieves one or more keys at the current iterator position in the associated dictionary. It 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 a dictionary 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.

```
vars
    iter : Iterator;
    cust : Customer;
    name, city : String;  // variables to receive dictionary key values
begin
    iter := app.myBank.allCustomers.createIterator;
    while iter.next(cust) do
        // retrieve the first key
        iter.getCurrentKeys(name);
        // retrieve the first two keys
        iter.getCurrentKeys(name, city);
    endwhile;
epilog
    delete iter;
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.)

This method can be used as an alternative to the **getIteratorKeys** of the **Dictionary** class to avoid share locking the associated dictionary on each call.

isValid

Signature `isValid(): Boolean;`

The **isValid** method of the **Iterator** class returns **true** if the receiver is a valid iterator.

next

Signature `next(value: Any output): Boolean updating;`

The **next** method of the **Iterator** class accesses successive entries one at a time in the collection to which the iteration is attached. The **value** parameter receives the next entry in the collection and must be of the same type as the members in the collection.

This method returns **true** when it has returned a value or it returns **false** when the iterator is positioned after the last entry in the collection.

The following examples show the use of the **next** method.

```
getRelativePosition(pObj: Object): Integer;
vars
    coll : Collection;
    pos  : Integer;
    obj  : Object;
    iter : Iterator;
begin
    coll := self.getCollection;
    iter := coll.createIterator;
    while iter.next(obj) do
        pos := pos + 1;
        if obj = pObj then
            break;
        endif;
    endwhile;
    return pos;
end;

buttonNext_click(btn: Button input) updating;
begin
    if self.cust <> app.myCompany.allCustomers.last then
        self.iter.next(self.cust);
        listBoxCustomers.listIndex := listBoxCustomers.listIndex + 1;
    endif;
    self.displayInstance;
end;
```

reset

Signature `reset() updating;`

The **reset** method of the **Iterator** class restarts an iteration. After this method, the following **next** method invocation starts at the first entry in the collection or the next **back** method invocation starts at the end of the collection. The following example shows the use of the **reset** method.

```
load() updating;
begin
   self.centreWindow;
   self.iter := app.myCompany.allCustomers.createIterator;
   self.iter.reset;
   self.iter.next(cust);
   if cust <> null then
      textBoxName.text    := self.cust.name;
      textBoxAddress.text := self.cust.address;
      textBoxContact.text := self.cust.contact;
      listBoxCustomers.listCollection(app.myCompany.allCustomers, true, 0);
      listBoxCustomers.listIndex := 1;
   else
      textBoxName.text := " No Customer Instances";
   endif;
end;
```

startAtIndex

Signature `startAtIndex(index: Integer64) updating;`

The **startAtIndex** method of the **Iterator** class is the abstract method that sets the starting position of the iterator in the attached collection to a relative index. Specify the required position in the **index** parameter.

Note This method is not implemented for iterations of virtual collections.

startAtObject

Signature `startAtObject(object: Object) updating;`

The **startAtObject** method of the **Iterator** class is the abstract method that sets the starting position of the iterator in the attached collection at the position of the object specified in the **object** parameter.

Notes This method is not implemented for iterations of virtual collections.

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.

startNearIndex

Signature `startNearIndex(index: Integer64) updating;`

The **startNearIndex** method of the **Iterator** class is the abstract method that sets the starting position of the iterator in the attached collection approximate to a relative index. Specify the required position in the **index** parameter.

Note This method is implemented only for iterations of **Array**, **Set**, and **Dictionary** classes.

IUnknown Class

The **IUnknown** class is the abstract class that all COM objects implement and all other ActiveX interfaces inherit.

Note You can create neither transient nor persistent instances of the **IUnknown** class.

For details about:

- ActiveX interfaces, see "ActiveXInterface Class", earlier in this chapter
- **IDispatch** subclass, see "IDispatch Class", earlier in this chapter
- ActiveX automation servers, see "ActiveXAutomation Class", earlier in this chapter
- ActiveX controls, see "ActiveXControl Class", in Chapter 2
- Importing ActiveX type libraries, see "Using ActiveX Control and Automation Server Libraries", in Chapter 4 of the *JADE External Interface Developer's Reference*

Inherits From: ActiveXInterface

Inherited By: IDispatch

JadeAuditAccess Class

The **JadeAuditAccess** class framework encapsulates the behavior required to access information recorded in database transaction journals. The ability to analyze journals from a different JADE system is supported.

The **JadeAuditAccess** class provides:

- The ability to operate out-of-band; that is, in a separate JADE system
- Optional access to all of the embedded properties in created, updated, or deleted objects
- Optional access to a JADE dynamic object containing changed property values for updated objects
- Access to additional audited control information and events; for example, reorganization discontinuities and user sign-on and sign-off events

To use the functionality provided by the **JadeAuditAccess** class, set the **EnableDeltaLogging** parameter in the [PersistentDb] section of the JADE initialization file to **true** (the default) in the JADE system in which the journals were produced (that is, *not* in the JADE system in which they are analyzed).

The **UseJournalDescriptions** parameter in the [PersistentDb] section of the JADE initialization file activates the automatic matching of a description of the JADE system (that is, the classes and properties) with the audit journal records used by the **JadeAuditAccess** module. The description file used to identify classes and properties is created only when the **UseJournalDescriptions** parameter exists and it is set to **true**. (When the **EnableDeltaLogging** parameter is set to **false**, the **UseJournalDescriptions** parameter is ignored.)

A description of a JADE system is created at the completion of a reorganization of the JADE system, and by user request. The description is written into the current directory of the audit journal and the event is audited with the description file identification timestamp. This timestamp is also saved in the database control file. When a new audit file is first used, the most-recent description file timestamp is written in the audit file header record to identify the corresponding description file.

The **JadeAuditAccess** instance recognizes the description file timestamp in an audit file header record and in any subsequent record announcing the creation of a new description file, and it automatically loads (or reloads) the identified description file if it is available.

The **JadeAuditAccess** class provides methods that enable you to access the description of the class of the current journal record accessed and the values of properties of that class. You cannot access a JADE object of the user data from the journal record. Access to references yields a string containing the value of the object identifier (oid) or null (**""**). Access to an embedded blob or slob yields a **Binary** or **String** value containing its length and edition.

If no description is available for the class of the user data in the current journal record, the audited buffer, if requested, is available only as a **Binary** primitive type value.

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

For an example of a journal reader method that utilizes functionality of the **JadeAuditAccess** framework, see "JadeAuditAccess Class Method Example", later in this chapter.

Inherits From: Object

Inherited By: (None)

JadeAuditAccess Class Constants

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

Constant	Value	Constant	Value
Jaa_AccessMode_Long	1	Jaa_AccessMode_Standard	0
Jaa_Object_Blob	2	Jaa_Object_Collection	9
Jaa_Object_CollectionBlock	5	Jaa_Object_Null	0
Jaa_Object_Object	1	Jaa_Type_AbortTransaction	53
Jaa_Type_AuditSwitch	54	Jaa_Type_BeginTransaction	51
Jaa_Type_Blob	11	Jaa_Type_ChangeUser	98
Jaa_Type_CommitTransaction	52	Jaa_Type_Create	11
Jaa_Type_DatabaseClose	50	Jaa_Type_DatabaseOpen	49
Jaa_Type_Delete	12	Jaa_Type_NoAuditDiscontinuity	64
Jaa_Type_ReorgDiscontinuity	80	Jaa_Type_Slob	1
Jaa_Type_Update	17	Jaa_Type_UserSignOff	97
Jaa_Type_UserSignOn	96		

Note The **Jaa_Type_Blob** and **Jaa_Type_Slob** class constants refer to the property types **Binary** and **String**, respectively, and are values that can be returned in the **pType** parameter of a **getBlobProperty** or **getClassProperty** method call. For more details, see the **getBlobProperty** and **getClassProperty** methods.

JadeAuditAccess Properties

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

Property	Description
autoDescription	Specifies whether the description file is automatically loaded when required
currentClassNumber	Contains the class number associated with the currently retrieved audit record
currentObjectType	Contains the type of object associated with the currently retrieved audit record
currentOid	Contains the oid of the object associated with the currently retrieved audit record
currentRecordType	Contains the record type associated with the currently retrieved audit record
descriptionFilename	Contains the name used for the last description load attempt
descriptionPath	Contains the path used for the last description load attempt
descriptionTS	Contains the creation timestamp associated with the current description file

autoDescription

Type: Boolean

The read-only **autoDescription** property of the **JadeAuditAccess** class specifies whether the description file is automatically loaded when required.

By default, the value of this property is **true** (that is, the description file is automatically loaded).

The value of this property is set to **false** when you call the **loadDescription** method, to manually load a description file. When the value of this property is **false**, the **loadDescription** method prompts the user to specify the description file that is to be loaded. The standard File Open dialog is initialized with the assumed file path and title (which might not exist). If the File Open dialog is cancelled, subsequent journal access continues without description file data.

currentClassNumber

Type: Integer

The read-only **currentClassNumber** property of the **JadeAuditAccess** class contains the class number associated with the currently retrieved audit record.

This property contains zero (**0**) if no class is associated with the record.

currentObjectType

Type: Integer

The read-only **currentObjectType** property of the **JadeAuditAccess** class contains the type of object associated with the currently retrieved audit record.

Value	Class Constant	Description
0	Jaa_Object_Null	No object; that is, control record
1	Jaa_Object_Object	Object
2	Jaa_Object_Blob	Blob (binary large object)
5	Jaa_Object_CollectionBlock	Collection block
9	Jaa_Object_Collection	Collection

currentOid

Type: String[48]

The read-only **currentOid** property of the **JadeAuditAccess** class contains the oid of the object associated with the currently retrieved audit record.

This property contains null (**""**) if no object is associated with the record.

currentRecordType

Type: Integer

The read-only **currentRecordType** property of the **JadeAuditAccess** class contains the record type associated with the currently retrieved audit record.

The audit type can be one of the values listed in the following table.

Integer Value	JadeAuditAccess Class Constant	Action
11	Jaa_Type_Create	Create object

Integer Value	JadeAuditAccess Class Constant	Action
12	Jaa_Type_Delete	Delete object
17	Jaa_Type_Update	Update object
49	Jaa_Type_DatabaseOpen	Database open
50	Jaa_Type_DatabaseClose	Database close
51	Jaa_Type_BeginTransaction	Begin transaction
52	Jaa_Type_CommitTransaction	Commit transaction
53	Jaa_Type_AbortTransaction	Abort transaction
54	Jaa_Type_AuditSwitch	Audit switch
64	Jaa_Type_NoAuditDiscontinuity	No-audit discontinuity
80	Jaa_Type_ReorgDiscontinuity	Reorganization
96	Jaa_Type_UserSignOn	User sign-on
97	Jaa_Type_UserSignOff	User sign-off
98	Jaa_Type_ChangeUser	User change

descriptionFilename

Type: String

The read-only **descriptionFilename** property of the **JadeAuditAccess** class contains the name for the last description load attempt. The **getJournal**, **getNextJournal**, and **getNextRecord** method calls can result in the value of this property changing.

descriptionPath

Type: String

The **descriptionPath** property of the **JadeAuditAccess** class contains the path used for the last description load attempt.

The description file path is used to locate the description files as they are required while processing journals. The **getJournal**, **getNextJournal**, and **getNextRecord** method calls can result in a new description being loaded.

If the value of this property is null (**""**) when an attempt to load a description file occurs, a default value is established. The **getJournalPath** method is called to get the path of the journals. This path value is examined for a last directory level of **"current"** and, if it exists, it is removed. The resulting value is assigned to this property. If a non-default location is used, this property must be set before calling the **getJournal** method.

descriptionTS

Type: TimeStamp

The read-only **descriptionTS** property of the **JadeAuditAccess** class contains the creation timestamp associated with the current description file.

The **getJournal**, **getNextJournal**, and **getNextRecord** methods update the value of this property when it is detected that the description file corresponding to the journal (or record) returned by one of these calls differs from the description file currently in use.

This property contains null ("") if no description file is currently loaded.

JadeAuditAccess Methods

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

Method	Description
clearRegisteredFilters	Clears any registered class filters
descriptionClassIsSubclass	Returns **true** if the class number specified in the **classNumber** parameter is a subclass of the class number specified in the **superclassNumber** parameter in the loaded description
generateDescription	Creates a description file containing a list of the names and required metadata of all schemas, classes, and properties
getAfterImage	Returns a Binary containing a copy of the audit after-image beginning with the oid and followed by the object properties
getAfterPropertyValue	Returns the value of the specified property in the after-image
getBeforeImage	Returns a Binary containing a copy of the audit before-image beginning with the oid and followed by the object properties
getBeforePropertyValue	Returns the value of the specified property in the before-image
getBlobProperty	Returns the attributes of the specified blob or slob property
getBlobValue	Reassembles the complete before-image and after-image of the specified blob or slob
getChangedPropertyNames	Populates the passed array with the names of the properties that are partially or wholly spanned by changes in the journal record
getChangeUserData	Returns information from a journal record that audits changes to the user code of a process
getClassName	Returns the schema and class names of the specified class number
getClassNumber	Returns the class number of the specified class in the specified schema
getClassProperty	Retrieves the attributes of the property of the specified class number
getClassPropertyNames	Populates the passed array with the names of the properties of the specified class number
getCollectionBlockKeys	Populates the passed array with dynamic objects describing the additions, removals, and changes in this edition of the **MemberKeyDictionary** collection block
getCollectionBlockOid	Returns the string representation of the oid of the collection block record
getCollectionBlockOids	Populates the passed added and removed oid arrays with the string representations of the oids added and removed in this edition of the collection block
getJournal	Locates the journal (or the first journal) to be accessed
getJournalName	Returns the name of the current journal
getJournalNumber	Returns the number of the current journal

Method	Description
getJournalPath	Returns the path of the current journal
getNextJournal	Locates the next journal to be accessed
getNextRecord	Returns the next (relevant) record retrieved from the current journal file
getNextRecordUTC	Returns the next (relevant) record retrieved from the current journal file with additional UTC timestamp information
getProperty	Returns the attributes of the specified property for the class of the current audit record
getUTCBias	Returns the UTC bias value of the current journal
getUserData	Returns user name and index or the user index of the current record
loadDescription	Enables the user to select a description file to load
loadDescriptionByName	Loads a specified description file
nextAuditRecord	(Deprecated) Use the **getNextRecord** method
registerFilterClass	Specifies class filtering for the specified class number
registerFilterClassName	Specifies class filtering for the specified schema and class name
registerFilterCollection	Specifies collection class filtering for the specified class number
registerFilterCollectionName	Specifies collection class filtering for the specified schema and class name
registerFilterTimeRange	Specifies the first and last timestamps for filtering accessible audit records
registerFilterTimeRangeUTC	Specifies the first and last timestamps in Coordinated Universal Time (UTC) for filtering accessible audit records
setAccessMode	Sets the journal access mode
setFilterExcludes	Changes the record filtering mechanism to exclude mode

For an example of a journal reader method that utilizes functionality of the **JadeAuditAccess** framework, see "JadeAuditAccess Class Method Example", later in this chapter.

clearRegisteredFilters

Signature `clearRegisteredFilters();`

The **clearRegisteredFilters** method of the **JadeAuditAccess** class clears any registered class and collection class filters. For details about registering filters, see the **registerFilterClass**, **registerFilterClassName**, **registerFilterCollection**, and **registerFilterCollectionName** methods.

descriptionClassIsSubclass

Signature
```
descriptionClassIsSubclass(classNumber:      Integer;
                           superclassNumber: Integer): Boolean;
```

The **descriptionClassIsSubclass** method of the **JadeAuditAccess** class returns **true** if the class number specified in the **classNumber** parameter is a subclass of the class number specified in the **superclassNumber** parameter in the loaded description.

When operating:

- *Out of band* (that is, where journals and description files are from a different system), you cannot use the class numbers from audit records as input to meta-programming constructs in the executing process, as the schema definition does not exist.
- *In band*, the issue remains as to whether the description file in use is the same as that which is current (that is, there are no outstanding reorganizations that have yet to be traversed).

You must therefore use the **descriptionClassIsSubclass** method instead of the **Object** class **isKindOf** method.

For an example of the use of this method, see "*Sample Journal Dump Method*.under "*Putting Things Together*" in the *Audit Access White Paper*.

generateDescription

Signature `generateDescription(): TimeStamp;`

The **generateDescription** method of the **JadeAuditAccess** class creates a description file containing a list of the names and required metadata of all schemas, classes, and properties.

This **serverExecution** method extracts the metadata from your JADE environment and writes a list of the names and required metadata of all schemas, classes, and properties to the description file. The file is in the format required by the **loadDescription** method. The description file name is generated as **description<*timestamp*>.txt**, where the **<*timestamp*>** value is the current date and time in the format ***yyyyMMddhhmmss***.

The location of the generated file is specified by the **JournalRootDirectory** parameter in the [PersistentDb] section of the JADE initialization file. If this parameter is not set, the default location is the root journal directory.

This **generateDescription** function is invoked automatically (depending on the values of the **EnableDeltaLogging** and **UseJournalDescriptions** parameters in the [PersistentDb] section of the JADE initialization file) at the end of any reorganization or by calling this method from your JADE code.

Notes The description file is automatically created only when the **EnableDeltaLogging** parameter and the **UseJournalDescriptions** parameter in the [PersistentDb] section of the JADE initialization file are set to **true**.

As this method updates the database and is audited, the process that calls it must be in transaction state.

getAfterImage

Signature `getAfterImage(): Binary;`

The **getAfterImage** method of the **JadeAuditAccess** class returns a **Binary** containing a copy of the audit after-image beginning with the object identifier (oid) and followed by the object properties after the original object was updated.

The Binary result is null if there is no after-image (that is, it is not a create or update record).

Tip This method is primarily of diagnostic benefit.

getAfterPropertyValue

Signature `getAfterPropertyValue(pName: String): Any;`

The **getAfterPropertyValue** method of the **JadeAuditAccess** class returns the after-image value of the property specified in the **pName** parameter. A valid description, including the required class, must be available and loaded. If the description of the class is not available, a null value is returned. If the property name or after-image is invalid, an exception is raised.

The value returned for a binary large object (blob) or a string large object (slob) is a **Binary** or a **String** primitive type value, respectively, containing a string in the following format.

```
"iiiiii L mmmmmm Ed nnnnnn"
```

In this format, the ***iiiiii*** value is the name of the blob or the slob, the ***mmmmmm*** value is the length of the blob or slob data, and the ***nnnnnn*** value is the edition of the blob or slob.

getBeforeImage

Signature `getBeforeImage(): Binary;`

The **getBeforeImage** method of the **JadeAuditAccess** class returns a **Binary** containing a copy of the audit before-image beginning with the object identifier (oid) and followed by the object properties before the original object was updated.

The Binary result is null if there is no before-image (that is, it is not an update or delete record).

Tip This method is primarily of diagnostic benefit.

getBeforePropertyValue

Signature `getBeforePropertyValue(pName: String): Any;`

The **getBeforeImage** method of the **JadeAuditAccess** class returns the before-image value of the property specified in the **pName** parameter. A valid description, including the required class, must be available and loaded. If the description of the class is not available, a null value is returned.

If the property name or before-image is invalid, an exception is raised.

The value returned for a binary large object (blob) or a string large object (slob) is a **Binary** or a **String** primitive type value, respectively, containing a string in the following format.

```
"iiiiii L mmmmmm Ed nnnnnn"
```

In this format, the ***iiiiii*** value is the name of the blob or the slob, the ***mmmmmm*** value is the length of the blob or slob data, and the ***nnnnnn*** value is the edition of the blob or slob.

getBlobProperty

Signature
```
getBlobProperty(pOid:         String;
                pParentClass: Integer output;
                pParentOID:   String output;
                pName:        String output;
                pType:        Integer output): Boolean;
```

The **getBlobProperty** method of the **JadeAuditAccess** class returns the attributes of the blob or slob property specified in the **pOid** parameter.

The **pParentClass** parameter retrieves the class number of the class in which the blob or slob is declared, the **pParentOID** retrieves the owning instance of the parent class (that is, the value returned in the **pParentClass** parameter), the **pName** parameter retrieves the name of the blob or slob, and the **pType** parameter retrieves whether it is a blob (**Jaa_Type_Blob**) or a slob (**Jaa_Type_Slob**). A valid description of the **pParentClass** class must be available.

A return value of **false** indicates that a valid class or property description is not available, or that the oid is not a blob or a slob.

getBlobValue

Signature
```
getBlobValue(pBeforeImageLength: Integer output;
             pBeforeImage:       Binary output;
             pAfterImageLength:  Integer output;
             pAfterImage:        Binary output;
             pBoolean:           Boolean output): Boolean;
```

When the current audit record retrieved by the **getNextRecord** method has an **pObjectType** parameter value of **Jaa_Object_Blob**, you can call the **getBlobValue** method of the **JadeAuditAccess** class to retrieve the before-image and after-image value of the blob or slob.

The retrieved values of the **pBeforeImageLength** and **pAfterImageLength** parameters are set to the total size of the before-image and after-image, respectively.

The values of the **pBeforeImage** and **pAfterImage** parameters are set to the before- and after-images, respectively.

The **pBoolean** parameter is reserved and exists for compatibility with prior releases.

The **getBlobValue** method returns a **Boolean** value for compatibility reasons. This value is always **true**.

getChangeUserData

Signature
```
getChangeUserData(pUserName:   String  output;
                  pIndex:      Integer output;
                  pByUserName: String  output;
                  pByIndex:    Integer output);
```

The **getChangeUserData** method of the **JadeAuditAccess** class retrieves data from a **Jaa_Type_ChangeUser** audit record. The **pUserName** parameter is the new user code for the process identified by the **pIndex** parameter. The **pByUserName** and **pByIndex** parameters identify the process that performed the user code change operation.

getChangedPropertyNames

Signature
```
getChangedPropertyNames(pNames: JadeIdentifierArray input): Boolean;
```

The **getChangedPropertyNames** method of the **JadeAuditAccess** class clears the array specified in the **pNames** parameter and then populates it with the names of the properties that are partially or wholly spanned by changes in the journal record; that is, the object properties that have changed value. A valid description of the required class must be available.

This method returns **false** if a valid class description is not available or there is no change information in the current journal record.

getClassName

Signature
```
getClassName(pClassNumber: Integer): String;
```

The **getClassName** method of the **JadeAuditAccess** class returns the schema and class names of the class specified in the **pClassNumber** parameter. Values in the returned string are separated by colon (**:**) characters.

A null result (**""**) indicates that valid class description information is not available or that the specified class number is not valid.

getClassNumber

Signature
```
getClassNumber(pSchemaName: String;
               pClassName:  String): Integer;
```

The **getClassNumber** method of the **JadeAuditAccess** class returns the class number of the class with schema name and class name specified by the values of the **pSchemaName** and **pClassName** parameters, respectively.

A return value of zero (**0**) indicates that there is no class with the specified class name in the specified schema.

getClassProperty

Signature
```
getClassProperty(pClassNumber: Integer;
                 pName:        String;
                 pType:        Integer output;
                 pLength:      Integer output;
                 pPrecision:   Integer output;
                 pScale:       Integer output;
                 pRefClass:    Integer output): Boolean;
```

The **getClassProperty** method of the **JadeAuditAccess** class obtains property information. From the input class number specified by the **pClassNumber** parameter, it returns the information for the property specified by the **pName** parameter as loaded from the description file. The description file is a text file generated using JADE meta schema programming.

The **pType** output parameter is the property type value. Given an instance **prop** of Property:

A line in the description file …	Is generated using the string expression …
p=<property name>	'p=' & prop.name;
; t=<type name>/<type number>	'; t=' & prop.type.name & '/' & prop.type.number.String;

The **pType** parameter can be any of the property types that JADE implements. The following code example is similar to the code that generates the type information for the attributes of a class **pClass** in a description file.

```
if pClass.allProperties.size > 0 then
    foreach prop in pClass.allProperties do
        if prop.virtual then
            continue;
        endif;
        str:= "prop=" & prop.name;
        if prop.isKindOf(Attribute) then
            str:= str & " [" & prop.type.name & "(" &
                  prop.type.number.String & ")]";
        endif;
        write str;
    endforeach;
endif;
```

This may or may not be of interest to the **JadeAuditAccess** code you implement. It is used internally by **JadeAuditAccess** to process attributes of a specific type (for example, blobs and slobs) appropriately.

When operating *out of band* (where journals and description files are from a different system), you cannot use the class and property information from **JadeAuditAccess** as input to meta-programming constructs in the executing process, as the schema definition does not exist.

When operating *in band*, the issue remains as to whether the description file in use is the same as that which is current (that is, there are no outstanding reorganizations that **JadeAuditAccess** has yet to traverse).

getClassPropertyNames

Signature `getClassPropertyNames(pClassNumber: Integer; pNames: JadeIdentifierArray input): Boolean;`

The **getClassPropertyNames** method of the **JadeAuditAccess** class clears the array specified in the **pNames** parameter and then populates it with the names of the properties of the class specified in the **pClassNumber** parameter.

A valid description of the specified class must be available. A return value of **false** indicates that a valid class description is not available.

getCollectionBlockKeys

Signature `getCollectionBlockKeys(pEntries: JadeDynamicObjectArray input);`

The **getCollectionBlockKeys** method of the **JadeAuditAccess** class clears the array specified in the **pEntries** parameter and then populates it with dynamic objects describing the additions, removals, and changes in this edition of the **MemberKeyDictionary** collection block.

The dynamic objects have the name attribute value of **JAA_MemberKeyDictionaryEntryName** (**"JAAMemberKeyDictionaryEntry"**) and the type attribute value of **JAA_MemberKeyDictionaryEntryType** (**50**).

The dynamic attributes for the **JAAMemberKeyDictionaryEntry** object type are listed in the following table.

Name	Type	Description
entryOid	String	Oid of the entry
keyBytes	Integer	Length of keys in bytes
beforeKeys	Binary	The entry keys value in the block before image
afterKeys	Binary	The entry keys value in the block after image

Entries added to the block have an empty **beforeKeys** value. Entries removed from the block have an empty **afterKeys** value. Entries with changed keys have before and after key values.

getCollectionBlockOid

Signature `getCollectionBlockOid(): String;`

The **getCollectionBlockOid** method of the **JadeAuditAccess** class returns the string representation of the collection block oid if the current object type is a collection block.

An empty string (**""**)is returned If the current object type is not a collection block.

getCollectionBlockOids

Signature `getCollectionBlockOids(pAddedOids: StringArray input;
pRemovedOids: StringArray input);`

The **getCollectionBlockOids** method of the **JadeAuditAccess** class clears the arrays specified in the **pAddedOids** and **pRemovedOids** parameters and then populates the **pAddedOids** array with the string representations of the oids added, and populates the **pRemovedOids** array with the string representations of the oids removed, in this edition of the collection block.

getJournal

Signature
```
getJournal(pDirectory:     String;
           pJournalNumber: Integer;
           pRecordOffset:  Integer io): Integer updating;
```

The **getJournal** method of the **JadeAuditAccess** class locates the journal (or first journal) to be accessed and opens the file.

The **pDirectory** parameter specifies the absolute path of the journal folder on the server on which the application is running, the **pJournalNumber** parameter specifies the required audit journal, and the **pRecordOffset** parameter specifies the number of characters to be skipped at the beginning of the file.

The file is positioned to the first audit record after the specified offset and that file position is returned in the **pRecordOffset** parameter.

This method returns zero (**0**) if the audit journal is available. If it is not available, an exception is raised.

If there is a description file associated with the current journal and offset, an attempt is made to load it. The **descriptionPath** property, if null (**""**), is assigned a default value and the **descriptionFilename** property is assigned. The **descriptionTS** property is assigned only after a successful description load. If the description file is already loaded and its timestamp has not changed, it is not reloaded.

If the description file is not available, all methods requiring class or property names as parameters, or in their results, will not function (and they will usually return a **false** result).

getJournalName

Signature `getJournalName(): String;`

The **getJournalName** method of the **JadeAuditAccess** class returns the name of the current journal.

getJournalNumber

Signature `getJournalNumber(): Integer;`

The **getJournalNumber** method of the **JadeAuditAccess** class returns the number of the current journal.

getJournalPath

Signature `getJournalPath(): String;`

The **getJournalPath** method of the **JadeAuditAccess** class returns the path of the current journal.

getNextJournal

Signature `getNextJournal(): Integer updating;`

The **getNextJournal** method of the **JadeAuditAccess** class locates the next journal to be accessed. The file is positioned at the first audit record after the journal header record.

This method returns zero (**0**) if the audit journal is available. If it is not available, an exception is raised.

If there is a description file associated with the next journal, an attempt is made to load it. The **descriptionPath** property, if null (**""**), is assigned a default value and the **descriptionFilename** property is assigned. The **descriptionTS** property is assigned only after a successful description load. If the description file is already loaded and its timestamp has not changed it is not reloaded.

If the description file is not available, all methods requiring class or property names as parameters, or in their results, will not function (and they will usually return a **false** result).

getNextRecord

Signature
```
getNextRecord(pType:          Integer output;
              pObjectType:    Integer output;
              pRecordOffset:  Integer output;
              pTimestamp:     TimeStamp output;
              pSerialNumber:  Decimal output;
              pTransactionId: Decimal output;
              pOid:           String output;
              pClassNumber:   Integer output;
              pEdition:       Integer output): Boolean updating;
```

The **getNextRecord** method of the **JadeAuditAccess** class returns the next (relevant) record retrieved from the current journal file. The current journal file must have been previously opened by using the **getJournal** or **getNextJournal** method.

This method does the same as the **JadeAuditAccess** class **getNextRecordUTC** method, except that it does not return the Coordinated Universal Time (UTC) timestamp and UTC bias values of the record. For details about the values returned by each of the parameters in the method, see the **getNextRecordUTC** method.

If you want to retrieve a record with the UTC timestamp and UTC bias values, use the **getNextRecordUTC** method.

getNextRecordUTC

Signature

```
getNextRecordUTC(pType:          Integer output;
                 pObjectType:    Integer output;
                 pRecordOffset:  Integer output;
                 pUTCTimestamp:  TimeStamp output;
                 pUTCBias:       Integer output;
                 pTimestamp:     TimeStamp output;
                 pSerialNumber:  Decimal output;
                 pTransactionId: Decimal output;
                 pOid:           String output;
                 pClassNumber:   Integer output;
                 pEdition:       Integer output): Boolean updating;
```

The **getNextRecordUTC** method of the **JadeAuditAccess** class returns the next (relevant) record retrieved from the current journal file. The current journal file must have been previously opened by using the **getJournal** or **getNextJournal** method.

This method returns **true** if a journal record matching the current filtering was located in the current journal file or it returns **false** if there are no further audit records in the current journal file to access.

If there is a new description file associated with the record, it is loaded and the **descriptionFilename** and **descriptionTS** properties are updated. An exception (*5003 - Requested file not found*) is raised if a new description file cannot be opened. The extended error text contains the timestamp of the missing description file.

Notes The **getNextRecordUTC** method returns details of user-defined classes only. Details of system classes are not returned.

Records read from the audit journal are examined to retrieve description modification information prior to any filtering action.

The **getNextRecordUTC** method returns the Coordinated Universal Time (UTC) timestamp and UTC bias values of the record. If you want to retrieve a record without returning the UTC timestamp and UTC bias values, use the **getNextRecord** method of the **JadeAuditAccess** class.

The possible values returned in the **pType** parameter are listed in the following table.

Integer Value	JadeAuditAccess Class Constant	Action
11	Jaa_Type_Create	Create object
12	Jaa_Type_Delete	Delete object
17	Jaa_Type_Update	Update object
49	Jaa_Type_DatabaseOpen	Database open
50	Jaa_Type_DatabaseClose	Database close
51	Jaa_Type_BeginTransaction	Begin transaction
52	Jaa_Type_CommitTransaction	Commit transaction
53	Jaa_Type_AbortTransaction	Abort transaction
54	Jaa_Type_AuditSwitch	Audit switch
64	Jaa_Type_NoAuditDiscontinuity	No-audit discontinuity

Integer Value	JadeAuditAccess Class Constant	Action
80	Jaa_Type_ReorgDiscontinuity	Reorganization
96	Jaa_Type_UserSignOn	User sign-on
97	Jaa_Type_UserSignOff	User sign-off
98	Jaa_Type_ChangeUser	Change user code

The possible values returned in the **pObjectType** parameter are listed in the following table.

Integer Value	JadeAuditAccess Class Constant	Description
0	Jaa_Object_Null	No object; that is, control record
1	Jaa_Object_Object	Object
2	Jaa_Object_Blob	Blob (binary large object)
5	Jaa_Object_CollectionBlock	Collection block
9	Jaa_Object_Collection	Collection

The values returned by the other parameters in the method are listed in the following table.

Parameter	Description
pRecordOffset	Offset of the record in the journal.
pUTCTimestamp	Timestamp of the record in UTC.
pUTCBias	UTC bias when the record was timestamped, in minutes.
pTimestamp	Timestamp of the record as the local time of the system that created the journal.
pSerialNumber	Audit serial number.
pTransactionId	Transaction identifier.
pOid	Object identifier is returned as a **String** primitive type (and not as an **Object**, which could be invalid).
pClassNumber	Class number of the object.
pEdition	Update count of the object.

When the value of the **pObjectType** parameter is a collection block, the **pOid** parameter is the oid of the collection. Use the **getCollectionBlockOid** method to retrieve the oid of the collection block record.

Note Blob and collection block records are returned only when journal access mode is set to **Jaa_AccessMode_Long**. See the **setAccessMode** method.

getProperty

Signature
```
getProperty(pName:       String;
            pType:       Integer output;
            pLength:     Integer output;
            pPrecision:  Integer output;
            pScale:      Integer output;
            pRefClass:   Integer output): Boolean updating;
```

The **getProperty** method of the **JadeAuditAccess** class returns the attributes of the property specified in the **pName** parameter for the class of the current audit record.

A valid description of the required class must be available. This method returns **false** if a valid class or property description is not available.

getUTCBias

Signature
```
getUTCBias(): Integer;
```

The **getUTCBias** method of the **JadeAuditAccess** class returns the UTC bias value (in minutes) of the current journal so record timestamps in Coordinated Universal Time (UTC) can be converted to original local time.

An exception is raised if no journal is open.

getUserData

Signature
```
getUserData(pUserName: String output;
            pIndex:    Integer output);
```

The **getUserData** method of the **JadeAuditAccess** class returns the user name and index, or the user index, of the current record.

If the current record is a sign-on or sign-off, both the name and index are returned. If the current record is a begin transaction, commit transaction, or abort transaction, the index is returned.

An exception is raised if the current record is any other record type.

loadDescription

Signature
```
loadDescription(): Boolean updating;
```

The **loadDescription** method of the **JadeAuditAccess** class attempts to load a user-specified description file. The standard File Open dialog is displayed, to enable you to specify the file to load. If the File Open dialog is cancelled, processing continues with no description file loaded.

This method sets the value of the **autoDescription** property to **false**, so any subsequent automatic description file load request is presented to the user for confirmation. At such time, you can accept the automatically identified file, specify an alternative file, or proceed without a description file.

The **descriptionPath** property, if null (**""**), is assigned a default value and the **descriptionFilename** property is assigned. The **descriptionTS** property is assigned only after a successful description load.

The **loadDescription** method is automatically invoked by the **getNextRecord**, **getJournal**, and **getNextJournal** methods, to load the specified description file whenever a change is detected.

loadDescriptionByName

Signature `loadDescriptionByName(fileName: String): Boolean updating;`

The **loadDescriptionByName** method of the **JadeAuditAccess** class attempts to load the description file specified in the **fileName** parameter, rather than displaying the common File Open dialog. This method returns **true** if the specified description file loads correctly; otherwise it returns **false**.

The value specified in the **fileName** parameter is appended to the value of the **descriptionPath** property, to access the description file. If the description file is not available, the **loadDescriptionByName** method returns **false**. In addition, a two-digit or a four-digit year value in the file name of a description file is handled correctly.

This method ignores the value of the **autoDescription** property.

The **descriptionTS** property value is assigned only after a successful description load.

registerFilterClass

Signature `registerFilterClass(classNumber: Integer);`

The **registerFilterClass** method of the **JadeAuditAccess** class specifies a class number in the **classNumber** parameter for filtering.

If any classes are registered, only instances of those classes are returned by calls to the **getNextRecord** method. (See also the **setFilterExcludes** method.)

registerFilterClassName

Signature
```
registerFilterClassName(pSchemaName: String;
                        pClassName:  String): Boolean;
```

The **registerFilterClassName** method of the **JadeAuditAccess** class specifies class filtering for the schema and class name specified in the **pSchemaName** and **pClassName** parameters, respectively.

If any classes are registered, only instances of those classes are returned by calls to the **getNextRecord** method. (See also the **setFilterExcludes** method.)

The **getJournal** method must have been successfully called to locate the required audit journal and to load the associated description file before this method can be used. The **Boolean** return value indicates whether the filter was set.

registerFilterCollection

Signature
```
registerFilterCollection(pClassNumber:       Integer;
                         pParentClassNumber: Integer);
```

The **registerFilterCollection** method of the **JadeAuditAccess** class specifies collection class filtering. If any collection classes are registered, only instances of those collection classes are returned by calls to the **getNextRecord** method. (See also the **setFilterExcludes** method.)

A zero (**0**) value of the **pParentClassNumber** parameter specifies that monitoring is performed regardless of the parent class. A non-zero value of the **pParentClassNumber** parameter specifies that monitoring is performed if the collection is on the specified parent class.

registerFilterCollectionName

Signature `registerFilterCollectionName(pSchemaName: String; pParentClassName: String; pRefPropertyName: String): Boolean;`

The **registerFilterCollectionName** method of the **JadeAuditAccess** class specifies collection class filtering, using the schema, parent class, and reference property names specified in the **pSchemaName**, **pParentClassName**, and **pRefPropertyName** parameters, respectively.

The **getJournal** method must have been successfully called to locate the required audit journal and to load the associated description file before this method can be used.

If any collection classes are registered, only instances of those collection classes are returned by calls to the **getNextRecord** method. (See also the **setFilterExcludes** method.)

The **registerFilterCollectionName** method returns **true** if the filter was set.

registerFilterTimeRange

Signature `registerFilterTimeRange(pStartTime: TimeStamp; pEndTime: TimeStamp): Integer;`

The **registerFilterTimeRange** method of the **JadeAuditAccess** class specifies the first and last timestamps for filtering accessible audit records.

This method returns zero (**0**) if there is no error or it returns the applicable error code.

Specify the start and end times in the local time of the system that created the journal file or files. To specify the start and end times in Coordinated Universal Time (UTC), use the **registerFilterTimeRangeUTC** method.

If the value of the **pStartTime** parameter is not null, the first accessible audit record has a timestamp equal to or greater than the start timestamp value specified in the parameter.

If the value of the **pEndTime** is not null, the last accessible audit record has a timestamp less than the end timestamp value specified in the parameter.

If required, the value of the **pStartTime** parameter must be registered before the journal file is opened by calling the **getJournal** method.

The specified time range is not affected by the value of the **setFilterExcludes** method parameter.

registerFilterTimeRangeUTC

Signature `registerFilterTimeRangeUTC(pStartTime: TimeStamp pEndTime: TimeStamp): Integer;`

The **registerFilterTimeRangeUTC** method of the **JadeAuditAccess** class specifies the first and last timestamps for filtering accessible audit records in Coordinated Universal Time (UTC).

This method returns zero (**0**) if there is no error or it returns the applicable error code.

Specify the start and end times in UTC time. To specify the start and end times in the local time of the system that created the journal file or files, use the **registerFilterTimeRange** method.

If the value of the **pStartTime** parameter is not null, the first accessible audit record has a UTC timestamp equal to or greater than the start timestamp value specified in the parameter.

If the value of the **pEndTime** parameter is not null, the last accessible audit record has a UTC timestamp less than the end timestamp value specified in the parameter.

If required, the value of the **pStartTime** parameter must be registered before the journal file is opened by calling the **getJournal** method.

The specified time range is not affected by the value of the **setFilterExcludes** parameter.

setAccessMode

Signature `setAccessMode(pMode: Integer);`

The **setAccessMode** method of the **JadeAuditAccess** class sets the journal access mode to one of the values listed in the following table.

JadeAuditAccess Class Constant	Value	Description
Jaa_AccessMode_Long	1	The **getNextRecord** method additionally returns journal records that create, update, or delete long **String** (slob) and long **Binary** (blob) property attributes, or collection blocks.
Jaa_AccessMode_Standard	0	The **getNextRecord** method returns journal records that create, update, or delete objects, and additional control information such as transaction boundaries, reorganization discontinuities, and user sign-on and sign-off events. Access to long **String** (slob) long **Binary** (blob) attributes and to collection blocks is not available.

setFilterExcludes

Signature `setFilterExcludes(pExclude: Boolean);`

The **setFilterExcludes** method of the **JadeAuditAccess** class changes the record filtering mechanism to exclude mode when this **pExclude** parameter is set to **true**; that is, only instances of classes or collection classes not registered are returned by calls to the **getNextRecord** method.

JadeAuditAccess Class Method Example

The method in the following example uses the functionality of the **JadeAuditAccess** class framework to read a journal. (For further examples, see the *Audit Access White Paper*.)

```
dumpJournal();
vars
    file: File;
    lyne: String;
    jDir: String;
    jaa: JadeAuditAccess;
    journalNum: Integer;
    offset: Integer;
    time: Time;
    date: Date;
    type: Integer;
    objType: Integer;
    ts: TimeStamp;
    serial: Decimal[20, 0];
```

```
    tranId: Decimal[20, 0];
    strOid: String;
    classNum: Integer;
    editn: Integer;
    userSlot: Integer;
    userName: String;
    byUserSlot: Integer;
    byUserName: String;
begin
    jDir:= 'c:\jadexxdev\system\journals\current\';
    journalNum:= 535;
    create file transient;
    file.fileName:= jDir & 'dump_of_journal_' & journalNum.String & '.txt';
    file.allowReplace:= true;
    file.kind:= File.Kind_ANSI;
    file.mode:= File.Mode_Output;
    file.open;
    if file.isAvailable then
        create jaa transient;
        jaa.getJournal(jDir, journalNum, offset);
        while jaa.getNextRecord(type, objType, offset, ts, serial, tranId,
                                strOid, classNum, editn) do
            lyne:= ts.Time.String & ' ' & serial.String & ' (' &
                   journalNum.String & ', ' & offset.String & ') ';
            if type = jaa.Jaa_Type_BeginTransaction then
                lyne:= lyne & 'beginTransaction';
            elseif type = jaa.Jaa_Type_CommitTransaction then
                lyne:= lyne & 'commitTransaction';
            elseif type = jaa.Jaa_Type_AbortTransaction then
                lyne:= lyne & 'abortTransaction';
            elseif type = jaa.Jaa_Type_UserSignOn then
                lyne:= lyne & 'signOn';
                jaa.getUserData(userName, userSlot);
                lyne:= lyne & ' index=' & userSlot.String &
                              ', name=' & userName;
            elseif type = jaa.Jaa_Type_UserSignOff then
                lyne:= lyne & 'signOff';
                jaa.getUserData(userName, userSlot);
                lyne:= lyne & ' index=' & userSlot.String &
                              ', name=' & userName;
            elseif type = jaa.Jaa_Type_ChangeUser then
                lyne:= lyne & 'changeUser';
                jaa.getChangeUserData(userName, userSlot, byUserName,
                                      byUserSlot);
                lyne:= lyne & ' index=' & userSlot.String &
                              ', name=' & userName &
                              ', by index=' & byUserSlot.String &
                              ', name=' & byUserName;
            elseif type = jaa.Jaa_Type_DatabaseOpen then
                lyne:= lyne & 'dbOpen';
            elseif type = jaa.Jaa_Type_DatabaseClose then
                lyne:= lyne & 'dbClose';
            elseif type = jaa.Jaa_Type_NoAuditDiscontinuity then
                lyne:= lyne & 'disc-noAudit';
            elseif type = jaa.Jaa_Type_ReorgDiscontinuity then
```

```
                lyne:= lyne & 'disc-reorg';
            elseif type = jaa.Jaa_Type_AuditSwitch then
                lyne:= lyne & 'disc-switch';
            elseif type = jaa.Jaa_Type_Create then
                lyne:= lyne & 'create';
                if objType = jaa.Jaa_Object_Object then
                    lyne:= lyne & 'object';
                elseif objType = jaa.Jaa_Object_Blob then
                    lyne:= lyne & 'blob';
                elseif objType = jaa.Jaa_Object_Collection then
                    lyne:= lyne & 'collection';
                endif;
            elseif type = jaa.Jaa_Type_Delete then
                lyne:= lyne & 'delete';
                if objType = jaa.Jaa_Object_Object then
                    lyne:= lyne & ' object';
                elseif objType = jaa.Jaa_Object_Blob then
                    lyne:= lyne & ' blob';
                elseif objType = jaa.Jaa_Object_Collection then
                    lyne:= lyne & ' collection';
                endif;
            elseif type = jaa.Jaa_Type_Update then
                lyne:= lyne & 'update';
                if objType = jaa.Jaa_Object_Object then
                    lyne:= lyne & ' object';
                elseif objType = jaa.Jaa_Object_Blob then
                    lyne:= lyne & ' blob';
                elseif objType = jaa.Jaa_Object_Collection then
                    lyne:= lyne & ' collection';
                endif;
            endif;
            file.writeLine(lyne);
        endwhile;
    endif;
epilog
    delete file;
    delete jaa;
end;
```

JadeBytes Class

The **JadeBytes** class is a collection subclass with a membership of the **Byte** primitive type. It provides an efficient way to store and retrieve instances of unstructured data (such as text, graphic images, sound or video streams) of arbitrary size.

The **JadeBytes** type provides an alternative to defining an attribute of the **Binary** (or **String**) primitive type and checking the **Maximum Length** check box on the Define Attribute dialog, to define a blob (or slob).

Note The maximum length of a **JadeBytes** instance is approximately 1,019G bytes.

The key benefits provided of the **JadeBytes** type over the existing blob and slob variants of the **Binary** and **String** primitive types are:

- The size of a **JadeBytes** instance is not limited by the size of object cache or by process virtual memory requirements
- Access to the data within a **JadeBytes** instance is efficient, random, and piece-wise
- **JadeBytes** instances can be stored and retrieved without displacing other objects cached by a node

You can create transient and shared transient instances of the **JadeBytes** class. You can create only persistent instances of a subclass of the **JadeBytes** class, as shown in the following code example.

```
vars
    jbytes : JadeBytes;
    ubytes : UserBytes;             // User-defined subclass of JadeBytes
begin
    create jbytes transient;        // Allowed
    beginTransientTransaction;
    create jbytes sharedTransient;  // Allowed
    commitTransientTransaction;
    beginTransaction;
    create jbytes persistent;       // Not allowed
    create ubytes persistent;       // Allowed
    commitTransaction;
end;
```

You can define attributes of type **JadeBytes** or a user-defined subclass. A **JadeBytes** attribute, like a **StringArray** or other primitive collection attribute, is an exclusive property; that is, the **JadeBytes** subobject is created and deleted with its parent object.

A **JadeBytes** attribute can be directly mapped to an SQL **BLOB** type in an RPS mapping; for example, the **IMAGE** type in SQL Server.

For details about the properties and methods defined in the **JadeBytes** class, see "JadeBytes Properties" and "JadeBytes Methods", later in this document.

Inherits From: Collection

Inherited By: (None)

Shared File JadeBytes

Shared file **JadeBytes** instances (that is, when the value of the **singleFile** property is set to **false**) store their content in a series of one or more data segments.

The data segments that make up a **JadeBytes** object are not created until they are required to store non-null data; for example, when storage is first allocated using the **allocate** method, segments are virtually allocated but are not created.

When data is inserted in one or more segments, the segment or segments are created and the state is considered committed. When data in non-committed segments is accessed, null values are returned. This deferred allocation behavior results in efficient memory and disk utilization for applications requiring a sparsely allocated byte array.

When a **JadeBytes** object is allocated explicitly using the **allocate** method, it behaves as though it contains all null values up to the specified length, even though no data segments have been created. Inserting one byte in the middle of a **JadeBytes** object with a length in the order of megabytes or gigabytes results in the creation of exactly one data segment to hold that byte. Data segments that are logically in front of or behind that committed segment are not created. Methods that retrieve the binary content of a sparsely allocated **JadeBytes** object return a stream of bytes of the allocated length with null values in all locations that have not been set.

Dedicated File JadeBytes

Each single file **JadeBytes** instance (that is, when the value of the **singleFile** property is set to **true**) stores its content directly in a unique disk file. No additional data is added to the content. This allows an external program such as Microsoft Word to directly open the file containing the instance content.

The disk file occupies space up to and including the last byte stored. The logical length of the instance can be greater than the length of the disk file. A request for content between the end of the disk file and the logical length (for example, set by the **allocate** method) returns a stream of null bytes. You can save more disk space by storing a sparse **JadeBytes** instance as non-single file.

Single file instances are expected to be updated using full content replacement operations rather than partial-update operations. The full content replacement operations (for example, **setContent** or **loadFromFile**) generate smaller journal records, as before images are not required.

The content of single file instances is never placed in the database disk cache. The disk file content is buffered using the File System cache of the operating system.

JadeBytes Properties

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

Property	Description
singleFile	Causes the binary content for the instance to be stored in its own dedicated file
readOnly	Controls whether the **singleFile** instance can be updated
unaudited	Controls whether changed content of a **singleFile** instance is included in journal records

singleFile

Type: Boolean

The **singleFile** property of the **JadeBytes** class specifies whether the binary content is stored in its own dedicated file.

Notes This property can be set to **true** for persistent instances only.

Once the **singleFile** property is set to **true** or the instance has been stored in the database, the value of this property cannot be changed. You should therefore set this property as soon as possible after an instance is created.

For persistent **JadeBytes** objects with the value of the **singleFile** property set to **false** (the default), the binary content is stored in a common file. The name of the file is constructed from the name of the map file followed by **_udr** (where **_udr** stands for Unstructured Data Resource), as follows.

```
mapFile_udr.dat
```

For persistent **JadeBytes** objects with the value of the **singleFile** property set to **true**, the binary content is stored in separate files, one for each object. The names of these files are constructed from the name of the map file followed by **_udr** (where **_udr** stands for Unstructured Data Resource) and the oid of the object, as follows.

```
mapFile_udr[oid].dat
```

Certain operations on a persistent **JadeBytes** object in a dedicated file effectively become file system operations (which the file system is optimized to perform). For example, content is appended to a **JadeBytes** object by appending to the end of the file. There is no costly free-space management involved as there would be for a standard database file. Similarly, truncating data becomes a simple file system operation, and deleting an object amounts to deleting the file.

Less journal space is used when an instance is updated by full content replacement, as only the new content is included in the journal records. Transaction abort and crash recovery is handled by renaming the original file from ***name*.dat** to ***name.transaction-number*.bak** and creating a new file for the content. If the transaction completes successfully, the .**bak** file is removed. If the transaction is aborted, the new .**dat** file is removed and the .**bak** file is renamed to .**dat**.

When the value of the **singleFile** property is set to **true**, the associated dedicated file is created as an empty file.

Each JadeBytes class singleFile instance includes an MD5 checksum of its contents. Full content-replacement operations (for example, **setContent** or **loadFromFile**) update this checksum as part of the operation. Partial-update operations (for example, **putData** or **appendData**) clear the checksum and it is your responsibility to invoke the **updateChecksum** method, if required. To determine if unexpected changes have been made to the content, invoke the **matchChecksum** method.

A potential drawback of storing persistent **JadeBytes** objects in separate files is an operating system limit on the number of files that a process can have open at one time.

readOnly

Type: Boolean

The **readOnly** property of the **JadeBytes** class specifies whether the instance can be modified.

This property can be set to **true** only if the value of the **singleFile** property is **true**.

When the value of the **readOnly** property is set to **true**, the instance is marked as read-only and attempts to update its content cause exception 1348 or 3182. In addition, the dedicated file has its **Read-only** attribute set to **true**, as a hint to external programs that the file contents should not be modified.

When the property is set to **false**, the instance is marked as able to be updated. In addition, the dedicated file has its **Read-only** attribute reset.

You should set the value of the **readOnly** property to **true** before you expose the dedicated file for direct access by an external process, to prevent that process from updating the file and JADE processes from updating the contents while it is exposed.

unaudited

Type: Boolean

The **unaudited** property of the **JadeBytes** class controls whether changes to content are written to the journal.

You can set the **unaudited** property to **true** only if the value of the **singleFile** property is also **true**.

When the value of the **unaudited** property is **true**, journal records generated by content-updating operations (for example, **putData**, **setContent**, or **loadFromFile**) do not include the changed content, which makes those journal records much smaller. However, the content changes:

- Made by partial-update operations (for example, **putData**) cannot be undone if the transaction is aborted
- Cannot be replayed during crash recovery or roll-forward recovery
- Cannot be replicated on SDS or RPS nodes

JadeBytes Methods

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

Method	Description
add	Appends one byte at the end of the binary content of the receiver
allocate	Allocates a specified amount of virtual storage on the receiver for binary content
appendData	Appends binary data at the end of the binary content of the receiver
at	Returns the byte at the specified offset from the binary content of the receiver
atPut	Places a specified byte value at a specified offset in the binary content of the receiver
clear	Clears the binary content of the receiver
copy	Copies the binary content of the receiver to an empty **JadeBytes** object
createIterator	Creates an iterator for the receiver that can iterate in the forwards iteration only
display	Returns a string containing a textual description of the state and binary content of the receiver
extractToFile	Extracts the binary content of the receiver to a file with a specified file name
extractToFileDirect	Extracts the binary content of the receiver to a file with a specified file name
extractUsingFile	Extracts the binary content of the receiver to a file using a specified **File** object
first	Returns the first byte of the binary content of the receiver
getContent	Returns the binary content of the receiver
getData	Returns a specified number of bytes from the binary content of the receiver starting at a specified offset
getFileTitle	Returns the path and file name of the dedicated file associated with a **singleFile** instance

Method	Description
getLength	Returns the length allocated to the receiver for storage of binary content
getSegmentCount	Returns the number of segments allocated to the receiver for storage of binary content
getSegmentSize	Returns the system assigned segment size in bytes
getStatistics	Populates a dynamic object with structural statistics
grow	Increases the virtual storage allocated to the receiver for binary content to the specified length
isEmpty	Returns **true** if no content has been assigned and no storage space has been allocated
last	Returns the last byte of the binary content of the receiver
loadFromFile	Loads the binary content of the receiver from a file with a specified file name
loadFromFileDirect	Loads the binary content of the receiver from a file with a specified file name
loadUsingFile	Loads the binary content of the receiver using a specified **File** object
matchChecksum	Calculates the MD5 checksum of the current binary contents and returns **true** if it matches the current stored checksum
purge	Clears the binary content of the receiver
putData	Places the specified binary data at a specified offset in the binary content of the receiver
setCaching	Enables or disables caching of the binary content of the receiver
setContent	Sets or replaces the binary content of the receiver with the specified data
setExpectedLength	Specifies the expected total length of the binary content of an empty receiver
truncate	Truncates the binary content of the receiver to the specified length
updateChecksum	Calculates the MD5 checksum of the current binary contents and updates the current stored checksum to match

add

Signature `add(value: MemberType) updating;`

The **add** method of the **JadeBytes** class appends a single byte specified in the **value** parameter at the end of the receiver. The length of the **JadeBytes** object is increased by one.

This partial-update method clears the stored checksum of **singleFile** instances.

The following example shows the use of the **add** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.add(#4A.Byte);
    bytes.add(#41.Byte);
    bytes.add(#44.Byte);
    bytes.add(#45.Byte);
    write bytes.getContent;      // Writes "JADE"
epilog
```

```
    delete bytes;
end;
```

allocate

Signature `allocate(length: Integer64) updating;`

The **allocate** method of the **JadeBytes** class allocates virtual storage on the receiver to hold binary content up to the specified length. The amount of virtual storage is specified by the value of the **length** parameter.

Use the **allocate** method to build binary content in a piece-wise manner with the **putData** or **atPut** methods. When data is loaded sequentially by using the **appendData** method or in its entirety by using the **loadFromFile** method, there is no need to allocate storage in advance.

Use the **allocate** method only on an empty **JadeBytes** object. If the **JadeBytes** object has a defined length greater than zero (**0**), use the **grow** method to increase the length (the amount of virtual storage).

The following example shows the use of the **allocate** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.allocate(5);
    bytes.add(#4A.Byte);
    write bytes.getContent;      // Writes "?????J", where ? is a null byte
epilog
    delete bytes;
end;
```

An invalid size specified in the **length** parameter (for example, less than zero (**0**) or greater than the maximum instance size of approximately 1,019G bytes) raises an exception.

appendData

Signature `appendData(data: Binary) updating;`

The **appendData** method of the **JadeBytes** class appends the binary data specified by the **data** parameter at the end of the receiver. The length of the **JadeBytes** object (the amount of virtual storage) is increased by the size of the binary data that is added.

You can append data to an empty **JadeBytes** object. If you know the total length of the binary content when assembling a **JadeBytes** object sequentially, you should use the **setExpectedLength** method to specify this length. This could result in more optimal segment size to store the binary content than the default value of 64K bytes.

This partial-update method clears the stored checksum of **singleFile** instances.

The following example shows the use of the **appendData** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("Jade".Binary);
    bytes.appendData("Bytes".Binary);
    write bytes.getContent;      // Writes "JadeBytes"
```

```
epilog
    delete bytes;
end;
```

at

Signature `at(offset: Integer64): Byte;`

The **at** method of the **JadeBytes** class returns the byte at the offset specified by the value of the **offset** parameter from the binary content of the receiver.

The **offset** parameter must contain a value between one and the length of the binary content in bytes.

The following example shows the use of the **at** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    write bytes.at(3);           // Writes 68 (ASCII value "D")
epilog
    delete bytes;
end;
```

atPut

Signature
```
atPut(offset: Integer64;
      value:  Byte) updating;
```

The **atPut** method of the **JadeBytes** class overwrites the specified byte value at the specified offset in the binary content of the receiver. The offset parameter must contain a value between one and the length of the data content in bytes.

This partial-update method clears the stored checksum of **singleFile** instances.

You can use the **allocate** method to allocate virtual storage before randomly inserting data into the receiver in a piece-wise fashion. You can use the **atPut** method to update parts of the binary content of a **JadeBytes** object, as shown in the following example.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    bytes.atPut(3, "N".Byte);
    write bytes.getContent;      // Writes "JANE"
epilog
    delete bytes;
end;
```

clear

Signature `clear() updating;`

The **clear** method of the **JadeBytes** class clears the binary content of the receiver and removes any virtual storage previously allocated to the object. The **clear** method is operationally identical to the **purge** method.

A **singleFile** instance has its associated disk file truncated to zero-length.

The following example shows the use of the **clear** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    bytes.clear;
    write bytes.getLength;       // Writes 0
epilog
    delete bytes;
end;
```

copy

Signature `copy(jadeBytes: JadeBytes input);`

The **copy** method of the **JadeBytes** class copies the binary content of the receiver **JadeBytes** object to the **JadeBytes** object specified by the **jadeBytes** parameter. If the object referred to by the **jadeBytes** parameter is not empty, an exception is raised.

If both the receiver and target are **singleFile** instances, the **loadFromFileDirect** method is used to transfer the content to the target instance; that is, the copy occurs within the server node and may use a fast operating system file copy routine.

The following example shows the use of the **copy** method.

```
vars
    bytes1, bytes2 : JadeBytes;
begin
    create bytes1;
    bytes1.setContent("JADE".Binary);
    create bytes2;
    bytes1.copy(bytes2);
    write bytes2.getContent;          // Writes "JADE"
epilog
    delete bytes1;
    delete bytes2;
end;
```

createIterator

Signature `createIterator(): Iterator;`

The **createIterator** method of the **JadeBytes** class creates an iterator for the **JadeBytes** object.

Use an iterator to remember the current byte position within the **JadeBytes** object. (For details about iterators, see the **Iterator** class.)

The following example shows the use of the **createIterator** method.

```
vars
    bytes : JadeBytes;
    iter  : Iterator;
    byte  : Byte;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    iter := bytes.createIterator;
    while iter.next(byte) do
        write byte;              // Writes 74  65  68  69 in turn
    endwhile;
epilog
    delete iter;
    delete bytes;
end;
```

display

Signature `display(): String;`

The **display** method of the **JadeBytes** class returns a string containing a textual description of the state and binary content of the receiver.

The following example shows the use of the **display** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    write bytes.display;
epilog
    delete bytes;
end;
```

The following output is written to the Jade Interpreter Output Viewer.

```
---JadeBytes/17144.1---
length = 4
segment size = 256
segments = 1

Content:
00000001 4A41 4445                                JADE
```

extractToFile

Signature
```
extractToFile(fileName:     String;
              allowReplace: Boolean);
```

The **extractToFile** method of the **JadeBytes** class extracts the binary content of the receiver to the file specified by the value of the **fileName** parameter, which must be a valid file name for the host machine executing the method.

The value of the **allowReplace** parameter determines whether an existing file with the same name can be replaced.

If the logical length of the receiver is zero, the output file is not created.

The following example shows the use of the **extractToFile** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    bytes.extractToFile("c:\example.txt", true);
epilog
    delete bytes;
end;
```

extractToFileDirect

Signature `extractToFileDirect(fileName: String; allowReplace: Boolean);`

The **extractToFileDirect** method of the **JadeBytes** class extracts the binary content of the receiver to the file specified by the value of the **fileName** parameter, which must be a valid file name for the machine running the server node.

The value of the **allowReplace** parameter determines whether an existing file with the same name can be replaced.

This method raises an exception if the receiver is not a **singleFile** instance.

If the logical length of the receiver is zero, the output file is not created.

The copy operation uses a fast operating system file copy method when the physical file length matches the logical length; otherwise a read/write loop is used, followed by a loop to write nulls to pad the new file to the logical length.

This non-updating method must be executed in transaction state so that the server can avoid blocking other users of **singleFile** instances while the copy operation is in progress.

The following example shows the use of the **extractToFileDirect** method.

```
vars
    bytes : MyJadeBytes;
begin
    beginTransaction;
    create bytes persistent;
    bytes.singleFile := true;
    bytes.setContent("JADE".Binary);
    bytes.extractToFileDirect("c:\example.txt", true);
epilog
    abortTransaction;
end;
```

extractUsingFile

Signature `extractUsingFile(file: File);`

The **extractUsingFile** method of the **JadeBytes** class extracts the binary content of the receiver using a **File** object specified by the **file** parameter. If the file specified by the **file** parameter is not open, an exception is raised.

Examples where you would use this method instead of the **extractToFile** method include:

- Converting a text file from ANSI to Unicode
- Opening the output file on a presentation client

The following example shows the use of the **extractUsingFile** method.

```
vars
    bytes : JadeBytes;
    file  : File;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    create file;
    file.fileName := "c:\example.txt";
    file.open;
    bytes.extractUsingFile(file);
epilog
    delete bytes;
    delete file;
end;
```

first

Signature `first(): MemberType;`

The **first** method of the **JadeBytes** class returns the first byte (that is, the **Byte** element at position **1**) from the binary content of the receiver. If the **JadeBytes** object is empty, an exception is raised.

The following example shows the use of the **first** method.

```
vars
    bytes : JadeBytes;
begin
    beginTransaction;
    create bytes;
    bytes.setContent("JADE".Binary);
    commitTransaction;
    write bytes.first;          // Writes 74 (ASCII value "J")
epilog
    delete bytes;
end;
```

getContent

Signature `getContent(): Binary;`

The **getContent** method of the **JadeBytes** class returns the binary content of the receiver. If the content you attempt to retrieve exceeds the value of the **JadeBytesGetContentLimit** parameter in the [JadeClient] section of the JADE initialization file, which is 64M bytes by default, an exception is raised.

Note For performance reasons, avoid using this method with a large **JadeBytes** object. When the following code fragment is executed, the local variable **bin** must store the entire binary content of the **bytes** object, which would require the use of virtual memory.

```
bin := bytes.getContent;
```

The following example shows the use of the **getContent** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    write bytes.getContent;      // Writes "JADE"
epilog
    delete bytes;
end;
```

getData

Signature
```
getData(offset: Integer64;
        length: Integer): Binary;
```

The **getData** method of the **JadeBytes** class returns a number of bytes from the binary content of the receiver starting at an offset specified by the value of the **offset** parameter and with a size specified by the value of the **length** parameter.

The value of the **offset** parameter must be between one and the byte length of the binary content.

The following example shows the use of the **getData** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    write bytes.getData(3,2);    // Writes "DE"
epilog
    delete bytes;
end;
```

The following code example shows the use of the **getData** method to read a large **JadeBytes** object in chunks. When the **getData** method is executed, a shared lock is acquired on the **JadeBytes** object. It is important to minimize locking activity by bracketing the reading of the chunks between **beginLoad** and **endLoad** instructions, which keeps the **JadeBytes** object locked for the entire read transaction.

```
vars
    length : Integer64;
```

```
    chunkSize : Integer;
    data : Binary;
    offset : Integer;
begin
    length := bytes.getLength;
    chunkSize := 64*1024;
    offset := 1;
    beginLoad;
    while length > 0 do
        if length < chunkSize then
            chunkSize := length.Integer;
        endif;
        data := bytes.getData(offset, chunkSize);
        // process data
        offset := offset + chunkSize;
        length := length - chunkSize;
    endwhile;
    endLoad;
end;
```

getFileTitle

Signature `getFileTitle(): String;`

The **getFileTitle** method of the **JadeBytes** class returns a string containing the path and file name associated with a **singleFile** instance. It returns an empty string for shared file instances.

The following example shows the use of the **getFileTitle** method.

```
vars
    bytes : MyJadeBytes;
begin
    beginTransaction;
    create bytes persistent;
    bytes.singleFile := true;
    write bytes.getFileTitle();
epilog
    abortTransaction;
end;
```

getLength

Signature `getLength(): Integer64;`

The **getLength** method of the **JadeBytes** class returns the length allocated to the receiver for storage. This is either the actual length of the content if it was loaded directly by using a method such as **appendData** or **loadFromFile**, or the virtual length (how many bytes the receiver can contain) if it was specified using the **allocate** method.

The following example shows the use of the **getLength** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.allocate(1000000);
```

```
    write bytes.getLength;       // Writes 1000000
epilog
    delete bytes;
end;
```

getSegmentCount

Signature `getSegmentCount(): Integer;`

The **getSegmentCount** method of the **JadeBytes** class returns the number of virtual segments allocated to the receiver for storage.

The following example shows the use of the **getSegmentCount** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.allocate(1000000);
    write bytes.getSegmentCount; // Writes 4
epilog
    delete bytes;
end;
```

getSegmentSize

Signature `getSegmentSize(): Integer;`

The **getSegmentSize** method of the **JadeBytes** class returns the system-assigned segment size in bytes. This method returns **Max_Integer** for **singleFile** instances.

The following example shows the use of the **getSegmentSize** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.allocate(1000000);
    write bytes.getSegmentSize;  // Writes 262144
epilog
    delete bytes;
end;
```

getStatistics

Signature `getStatistics(stats: JadeDynamicObject input);`

The **getStatistics** method of the **JadeBytes** class populates a dynamic object specified by the **stats** input parameter with structural statistics. The following example shows the use of the **getStatistics** method.

```
vars
    bytes : JadeBytes;
    stats : JadeDynamicObject;
begin
    create bytes;
    bytes.allocate(1000000);
```

```
    bytes.atPut(500000, "JADE".Byte);
    create stats;
    bytes.getStatistics(stats);
    write stats.display;
epilog
    delete bytes;
    delete stats;
end;
```

The following output is written to the Jade Interpreter Output Viewer.

```
---JStatsBytes(104)---
embeddedVector = true
entrySize = 1
length = 1000000
segmentSize = 262144
virtualSegments = 4
committedSegments = 2
tailSegmentLength = 237856
tailSegmentSize = 262185
```

grow

Signature `grow(length: Integer64) updating;`

The **grow** method of the **JadeBytes** class increases the virtual storage allocated to the binary content of the receiver to the length specified by the value of the **length** parameter.

If the specified length is less than or equal to the length that has already been allocated, the method has no effect.

The following example shows the use of the **grow** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.allocate(10);
    write bytes.getLength;       // Writes 10;
    bytes.grow(50);
    write bytes.getLength;       // Writes 50;
epilog
    delete bytes;
epilog
    delete stats;
end;
```

isEmpty

Signature `isEmpty(): Boolean;`

The **isEmpty** method of the **JadeBytes** class returns **true** if the receiver is empty; that is, no content has been assigned and no storage space has been allocated.

This method always returns **false** for **singleFile** instances, because the associated disk file is always present.

The following example shows the use of the **isEmpty** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    write bytes.isEmpty;          // Writes true
    bytes.allocate(1000);
    write bytes.isEmpty;          // Writes false
epilog
    delete bytes;
end;
```

last

Signature `last(): MemberType;`

The **last** method of the **JadeBytes** class returns the last byte that has been allocated for the binary content of the receiver. If the **JadeBytes** object is empty, an exception is raised.

The following example shows the use of the **last** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    write bytes.last;             // Writes 69 (ASCII value "E")
epilog
    delete bytes;
end;
```

loadFromFile

Signature `loadFromFile(fileName: String) updating;`

The **loadFromFile** method of the **JadeBytes** class loads the binary content of the receiver from the file specified by the value of the **fileName** parameter. If the file name is not valid for the host machine executing the method, an exception is raised.

The following example shows the use of the **loadFromFile** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.loadFromFile("c:\photo.jpg");
epilog
    delete bytes;
end;
```

loadFromFileDirect

Signature `loadFromFileDirect(fileName: String) updating;`

The **loadFromFileDirect** method of the **JadeBytes** class loads the binary content of the receiver from the file specified by the value of the **fileName** parameter. If the file name is not valid for the machine running the server node, an exception is raised.

This method raises an exception if the receiver is not a **singleFile** instance.

Use this method when the source data file is present on the machine running the server node, as it avoids moving the content to and from the node executing the method.

You can use this method to resynchronize the current content with the journal after direct external updates have been made. Pass the result of the **getFileTitle** method as the value of the **fileName** parameter of this **loadFromFileDirect** method. The current content is read to calculate the checksum and to write to the journal, which causes the current content to be replayed on SDS and RPS nodes and allows the current content to be re-established by a roll-forward recovery.

The following example shows the use of the **loadFromFileDirect** method.

```
vars
    bytes : MyJadeBytes;
begin
    beginTransaction;
    create bytes persistent;
    bytes.singleFile := true;
    bytes.loadFromFileDirect("c:\photo.jpg");
epilog
    abortTransaction;
end;
```

loadUsingFile

Signature `loadUsingFile(file: File) updating;`

The **loadUsingFile** method of the **JadeBytes** class loads the binary content of the receiver using a **File** object specified by the **file** parameter. If the file specified by the **file** parameter is not open, an exception is raised.

This method is an alternative to the **loadFromFile** method, with the following additional functionality.

- Greater control over file usage semantics; for example, converting a text file from ANSI to Unicode.
- Enabling the input file to be opened on the presentation client

The following example shows the use of the **loadFromFile** method.

```
vars
    bytes : JadeBytes;
    file  : File;
begin
    create file;
    file.fileName := "c:\photo.jpg";
    file.open;
    create bytes;
    bytes.loadUsingFile(file);
epilog
```

```
    delete bytes;
    delete file;
end;
```

matchChecksum

Signature `matchChecksum(): Boolean;`

The **matchChecksum** method of the **JadeBytes** class calculates the MD5 checksum for the current binary contents of a **singleFile** instance and returns **true** if it matches the stored checksum.

This method always returns **true** for shared file instances.

The following example shows the use of the **matchChecksum** method.

```
vars
    bytes : MyJadeBytes;
begin
    beginTransaction;
    create bytes persistent;
    bytes.singleFile := true;
    bytes.appendData("This is content".Binary);
    write bytes.matchChecksum();   // Writes "false"
    bytes.updateChecksum();
    write bytes.matchChecksum()    // Writes "true";
epilog
    abortTransaction;
end;
```

purge

Signature `purge() updating;`

The **purge** method of the **JadeBytes** class clears the binary content of the receiver and removes any virtual storage previously allocated to the object. The **purge** method is operationally identical to the **clear** method.

A **singleFile** instance has its associated disk file truncated to zero-length.

The following example shows the use of the **purge** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    bytes.purge;
    write bytes.getLength;       // Writes 0
epilog
    delete bytes;
end;
```

putData

Signature `putData(offset: Integer64; data: Binary) updating;`

The **putData** method of the **JadeBytes** class copies the binary data specified by the **data** parameter into the binary content of the receiver at the offset specified by the value of the **offset** parameter, overwriting the existing content. The **offset** parameter must contain a value between one and the current length of the binary content of the receiver. In addition, the value of the **offset** parameter plus the length of the binary data in the **data** parameter must not exceed the current length of the binary content of the receiver.

You can use the **allocate** method to allocate virtual storage before randomly inserting data into the receiver in a piece-wise fashion.

This partial-update method clears the stored checksum of **singleFile** instances.

You can use the **putData** method to update parts of the binary content of a **JadeBytes** object, as shown in the following example.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADE".Binary);
    bytes.putData(2, "UN".Binary);
    write bytes.getContent;      // Writes "JUNE"
epilog
    delete bytes;
end;
```

setCaching

Signature `setCaching(onOff: Boolean);`

The **setCaching** method of the **JadeBytes** class is used to enable or disable caching of the data content of the receiver by setting the value of the **onOff** parameter to **true** or **false**, respectively.

When caching is disabled (the default setting):

- At most, one data segment of the **JadeBytes** object can be resident in the object cache.
- The **JadeBytes** object is the first object to be replaced when the object cache becomes full.

When caching is enabled:

- More than one data segment of the **JadeBytes** object can be resident in the object cache.
- The **JadeBytes** object is treated like other objects when it comes to being replaced when the object cache becomes full.

The caching setting for the **JadeBytes** object applies for as long as it remains in the object cache.

This method has no effect on **singleFile** instances.

setContent

Signature `setContent(data: Binary) updating;`

The **setContent** method of the **JadeBytes** class sets or replaces the content of the receiver with the specified data and removes any virtual storage previously allocated to the object. The method uses the smallest number of segments and the smallest size of tail segment from the range of valid segment sizes, to provide the best fit for the content.

The following example shows the use of the **setContent** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("Jade".Binary);
    bytes.setContent("Bytes".Binary);
    write bytes.getContent;      // Writes "Bytes"
epilog
    delete bytes;
end;
```

setExpectedLength

Signature `setExpectedLength(length: Integer64) updating;`

The **setExpectedLength** method of the **JadeBytes** class specifies the expected total length of an empty **JadeBytes** object that will be assembled by using the **appendData** method. The expected length is taken into account when computing segment size.

Calling this method does not allocate any storage and the length remains set to zero (**0**).

Note An exception is raised if the **JadeBytes** object is not empty when the **setExpectedLength** method is called. Use the **clear** or **purge** method rather than **truncate**(**0**) or **setContent**(null) to empty a **JadeBytes** object.

The following example shows the use of the **setExpectedLength** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setExpectedLength(50);
    write bytes.getSegmentSize;  // Writes 256
    bytes.purge;
    bytes.setExpectedLength(1000000);
    write bytes.getSegmentSize;  // Writes 262144
epilog
    delete bytes;
end;
```

truncate

Signature `truncate(newLength: Integer64) updating;`

The **truncate** method of the **JadeBytes** class truncates the binary content of the receiver to the length specified by the value of the **newLength** parameter. The method uses the smallest number of segments and the smallest size of tail segment from the range of valid segment sizes, to provide the best fit for the truncated content.

If the value of the **newLength** parameter exceeds the current length of the content of the receiver, the method has no effect.

The following example shows the use of the **truncate** method.

```
vars
    bytes : JadeBytes;
begin
    create bytes;
    bytes.setContent("JADEBYTES".Binary);
    bytes.truncate(4);
    write bytes.getContent;      // Writes "JADE"
epilog
    delete bytes;
end;
```

updateChecksum

Signature `updateChecksum();`

The **updateChecksum** method of the **JadeBytes** class calculates the MD5 checksum for the current binary contents of a **singleFile** instance and updates the stored checksum to match. It does nothing for shared file instances.

The following example shows the use of the **updateChecksum** method.

```
vars
    bytes : MyJadeBytes;
begin
    beginTransaction;
    create bytes persistent;
    bytes.singleFile := true;
    bytes.appendData("This is content".Binary);
    write bytes.matchChecksum();   // Writes "false"
    bytes.updateChecksum();
    write bytes.matchChecksum()    // Writes "true";
epilog
    abortTransaction;
end;
```

JadeDatabaseAdmin Class

The **JadeDatabaseAdmin** class encapsulates the behavior required to write standalone or integrated database administration tools for your JADE applications; for example, for runtime JADE systems or for online database backups controlled from your user applications.

The **JadeDatabaseAdmin** class, in conjunction with the **DbFile** class, enables you to create database administration tools that provide:

- Database administration functions such as analyzing, compacting, certifying, or verifying your database.
- Full online backup to disk. (Backing up is not supported to any other medium.)
- Support for multiple backup destinations, backup concurrency, and data compression.

Note No support is provided for third-party backup tools.

For details about:

- The constants and methods defined in the **JadeDatabaseAdmin** class and event notifications, see "JadeDatabaseAdmin Class Constants", "JadeDatabaseAdmin Methods", and "JadeDatabaseAdmin Class Event Notifications", in the following subsections.
- Incorporating the JADE database administration framework to integrate online backup services into your own applications or to build standalone database administration applications, see Chapter 7, "Using the Database Administration Framework", of the *JADE Developer's Reference*.
- Database backup and recovery, see Chapter 3 of the *JADE Database Administration Guide*, "Administering the JADE Database". (See also "Subscribing to Backup Progress Events" and "Notification Event Methods" under "DbFile Class Event Notifications", earlier in this chapter.)
- External third-party snapshot backups, see "Non-JADE Backups" in the *Developing a Backup Strategy White Paper*.

Inherits From: Object

Inherited By: (None)

JadeDatabaseAdmin Class Constants

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

Constant	Integer Value	Description
BackupAbortedEvent	4000	Multiple file backup terminated by the user.
BackupCancelledEvent	8000	Multiple file backup has been cancelled by the user.
BackupCompleteEvent	3000	Multiple file backup completed normally.
BackupFailedEvent	9000	Multiple file backup has failed.
FileBackupCompleteEvent	2000	File backup has finished.
FileBackupStartEvent	1000	File backup has commenced.
JournalTransferEvent	6000	Recovery journal file has been transferred.

Constant	Integer Value	Description
Mode_Archive	4	Quiesces the database for archive backup.
Mode_Default	9	Restores the database to the initial mode and usage values.
Mode_Exclusive	1	Requests exclusive access to the database. If other users already have the database open, a database mode conflict is reported. Similarly, if one user already has the database open in exclusive mode, other users are prevented from opening the database.
Mode_Shared	0	Enables multiple current users to open the database.
Mode_Snapshot	6	Conditions the database for external third-party snapshot backup.
RpsStorageMode_Full	0	Full database replica RPS data store mode.
RpsStorageMode_MappedExtent	1	Mapped extent RPS data store mode.
RpsStorageMode_WorkingSet	2	Working set RPS data store mode.
Usage_NoAudit	2	Specifies that the recovery log is not maintained. This usage can result in better performance, but if an abnormal termination occurs and the database is not audited, recovery is not possible and the database must be restored from backup. Use the shared access mode to specify multiple users with no-audit usage. However, if any one application or shared-access user opens the database in no-audit mode, all users must request no-audit usage or a conflicts exception is raised.
Usage_ReadOnly	1	Specifies that applications have read-only access to the database. Multiple instances of the JADE Object Manager can access the database in read-only mode.
Usage_Update	0	Enables applications to update the database.

For details about these constants, see "JadeDatabaseAdmin Class Event Notifications" or the **changeDbAccessMode** method, later in this section.

JadeDatabaseAdmin Methods

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

Method	Description
abortBackup	Terminates an online backup transaction
backupAllDbFiles	Backs up all database files to a common directory
backupDbFiles	Backs up selected file kinds to a common directory
backupJournal	Copies the specified recovery journal file to backup
beginBackup	Starts an online backup transaction
changeDbAccessMode	Changes the access mode of the database

Method	Description
closeCurrentJournal	Closes and releases the current recovery journal file
commitBackup	Commits an online backup transaction
commitCoherentBackup	Commits an SDS Secondary backup transaction with coherent (commit-consistent) state
compactDbFiles	Compacts the specified files to optimize storage and reduce fragmentation
createRpsDatabase	Creates an RPS database for a specified schema and RPS mapping
disableByteProgressEvents	Disables operation and progress event notifications for the number of bytes of a file backed up
disableProgressEvents	Disables operation and progress event notifications for the percentage of a file backed up
doCheckpoint	Causes a database checkpoint operation to be queued to the database worker thread
doQuietpoint	Attempts to establish a database quiet point
enableByteProgressEvents	Enables operation and progress event notifications for the number of bytes of a file backed up
enableProgressEvents	Enables operation and progress event notifications for the percentage of a file backed up
getAbortJournalNumber	Returns the number of the journal containing the Begin Transaction record of the oldest active transaction
getAllDbFiles	Populates an array with references to database files
getArchiveJournalDirectory	Returns the name of the archive directory for transaction journals
getCreationTimestamp	Returns a timestamp containing the date and time the database was created
getCurrentJournalDirectory	Returns the current recovery journal file directory
getCurrentJournalName	Returns the current recovery journal file name
getCurrentJournalNumber	Returns the current recovery journal file number
getCurrentJournalOffset	Enables the calculation of amounts and rates of journal output
getDbFiles	Populates an array with references to files of selected kinds
getLastCheckpoint	Retrieves the journal number and byte offset of the last database checkpoint
getLatestBackupTimestamp	Returns the date and time the database was last backed up without error
getLatestFullBackupTimestamp	Returns the date and time all files in the database were last backed up without error
getOpenTimestamp	Returns the most recent date and time the database was opened
getReasonTrackingStoppedString	Returns a string containing a textual description of the **SDSStopTrackingCodes** global constant reason code

Method	Description
getRpsMappedFiles	Populates an array with all database files required for a specified RPS mapping
isArchival	Specifies whether archival recovery is enabled for the database
rpsAuditSqlScriptForReplay	Writes a journal record containing SQL to be replayed on an RPS node
rpsExtractData	Extracts a specified table or all tables, using specified parameter values
rpsExtractDataAll	Extracts all tables using specified parameter values
rpsExtractDataUsingIniOptions	Extracts a specified table or all tables using values stored in the [**JadeRps**] section of the JADE initialization file
rpsGetDatabaseParameters	Returns the schema name, RPS mapping name, and the storage mode of the RPS node
rpsStartDataPump	Starts the RPS **Datapump** application on the RPS node
rpsStopDataPump	Stops the RPS **Datapump** application on the RPS node
sdsAuditStopTracking	Specifies a number that is returned in the **userInfo** parameter of your user notification method when an **SDS_TrackingStopped** event occurs (primary, secondary, or RPS)
sdsDisablePrimaryConnection	Disables a connection from the current secondary database to the primary server (secondary or RPS only)
sdsDisablePrimaryConnectionAt	Disables a connection from a specified secondary database to the primary server (primary only)
sdsDisableReadAccess	Disallows read-only database access to the current secondary database (secondary only)
sdsDisableReadAccessAt	Disallows read-only database access at a specified secondary database (primary only)
sdsEnableReadAccess	Allows read-only database access to the current secondary database (secondary only)
sdsEnableReadAccessAt	Allows read-only database access at a specified secondary database (primary only)
sdsGetDatabaseRole	Returns the database role of the current server for the JADE system (primary, secondary, or RPS)
sdsGetDatabaseSubrole	Returns the database subrole of the server on which the method is executed
sdsGetMyServerInfo	Obtains an array describing the SDS attributes of the system (primary, secondary, or RPS)
sdsGetSecondaryInfo	Obtains an array containing the SDS attributes for a specified secondary system (primary only)
sdsGetSecondaryProxies	Obtains an array of secondary proxy dynamic objects (primary only)
sdsGetSecondaryProxy	Obtains information about a specific secondary proxy dynamic object (primary only)
sdsGetTransactions	Obtains an array of transaction dynamic objects on the current secondary system (secondary only)

Method	Description
sdsGetTransactionsAt	Obtains an array of transaction dynamic objects on a specified secondary system (primary only)
sdsInitiateHostileTakeover	Initiates a hostile take-over by the executing secondary system (secondary only)
sdsInitiateTakeover	Initiates a negotiated take-over by a specified secondary server so that it becomes the primary server (primary only)
sdsIsInitialized	Returns **true** if SDS is initialized for the system (primary. secondary, or RPS)
sdsIsRunning	Returns **true** if SDS is running for this system (primary, secondary, or RPS)
sdsReconnectNow	Prompts the secondary database to attempt a reconnect to its primary server (secondary or RPS only)
sdsReplayNextJournal	Initiates a replay of the next ready journal on a secondary server when journal replay is suspended (secondary or RPS only)
sdsReplayNextJournalAt	Initiates a replay of the next ready journal on the specified secondary server when journal replay is suspended (primary only)
sdsResume	Resumes replaying journals after tracking has been interrupted (secondary or RPS only)
sdsResumeAt	Resumes replaying journals on a specified secondary server when tracking has been interrupted (primary only)
sdsStartService	Starts an SDS on the database server of the system (primary only)
sdsStartTracking	Starts tracking on the secondary database that calls this method (secondary or RPS only)
sdsStartTrackingAt	Starts tracking on a specified secondary database (primary only)
sdsStopService	Stops an SDS on the database server of the system (primary only)
sdsStopTracking	Stops tracking on the secondary database that calls this method (secondary or RPS only)
sdsStopTrackingAt	Stops tracking on a specified secondary database (primary only)
verifyJournal	Verifies the consistency of the specified recovery journal file

abortBackup

Signature `abortBackup() updating;`

The **abortBackup** method of the **JadeDatabaseAdmin** class terminates a backup transaction and cancels pending backup operations. Although the **abortBackup** method does not remove any files that have been copied, the backup will not be usable.

File backups that are in progress can be interrupted only if you have enabled progress event notifications. (For details, see the **enableProgressEvents** method.) If backup progress events are disabled (the default), file backups that are in progress continue until files have been copied.

If the database is in a quiescent read-only mode, the **abortBackup** method ends this mode, permitting updating transactions to be processed.

The following example shows the use of the **abortBackup** method.

```
cancelBackup();
begin
    // signal our dba to abort the current backup operation
    self.dba.abortBackup;
end;
```

backupAllDbFiles

Signature

```
backupAllDbFiles(backupDir:       String;
                 includeSysFiles: Boolean;
                 verifyFiles:     Boolean;
                 compressFiles:   Boolean;
                 overwriteFiles:  Boolean;
                 quiesce:         Boolean;
                 droppedFiles:    DbFileArray input) updating;
```

The **backupAllDbFiles** method of the **JadeDatabaseAdmin** class initiates a backup of all physical database files, optionally excluding system files, to the directory specified in the **backupDir** parameter. (The backup directory must be a valid directory that is relative to the server.)

Set the **includeSysFiles** parameter to **false** if you do not want to include system files in your backup process (that is, files categorized by **Kind = DbFile.Kind_System**). For details about the kinds of database files that you can back up, see the **DbFile** class **kind** property or "DbFile Class Constants", earlier in this chapter.

For the majority of database backups, it is not necessary to back up system files, as they are not updated in development or runtime systems and can be shared by multiple JADE environments (by using the **SystemFileDirectory** parameter in the [PersistentDb] section of the JADE initialization file). System files are updated only when the **RootSchema** or JADE patch files are loaded by using the Schema Load utility. (For details, see the *Jade Schema Load User's Guide*.)

Set the **verifyFiles** parameter to **true** if you want the backed up file checked, or verified. In a checked file backup, objects are read using the database access routines and object caching mechanisms, and at the same time, a verification of the data and indexes is performed. The verification performs various consistency checks similar to a database certify, to ensure the integrity of the backup. Furthermore, additional checksum information is added to the backup, to allow restore operations to verify the integrity of the backup as the data is restored.

Set the **compressFiles** parameter to **true** if you want to compress data on the fly as it is backed up. You can compress data in a checked or an unchecked backup.

Set the **overwriteFiles** parameter to **true** if you want to allow file backups to overwrite existing files in the destination backup directory. When this parameter is **false**, an exception is raised if an existing file is detected.

Use the **quiesce** parameter to allow a quiesced read-only backup transaction to be specified. When you set this parameter to **true**, the database is placed in a quiescent state by first allowing current active transactions to be completed, flushing modified buffers for cache to the stable database. In this mode, physical database files contain all committed updates to the database, and the files are opened in read-only mode with shared read access allowing external backup processes to safely copy database files.

In the quiescent mode, updating transactions are not permitted and attempts to execute database transactions raise a database exception. When a backup is performed in the quiescent mode, the physical database files are guaranteed to contain all database updates and a quiesced backup does not require backup recovery.

Set the **quiesce** parameter to **false** to allow updates during the backup process. When restoring a database backed up fully online (that is, this parameter is set to **false**), the restoration process requires the recovery of backed up transaction journals. A backup recovery is required after restoring files that were fully backed up online (that is, the **quiesce** parameter was set to **false**).

The **droppedFiles** parameter specifies an array of the database files that are not backed up.

The following example shows the use of the **backupAllDbFiles** method.

```
backupDatabase();
vars
    dba : JadeDatabaseAdmin;
    droppedFiles : DbFileArray;
    file : DbFile;
    includeSysFiles, verifyFiles, compressFiles, allowOverwrite : Boolean;
    quiesce : Boolean;
    backupDirectory, title, msg : String;
begin
    create dba transient;
    create droppedFiles transient;
    backupDirectory := "n:\jade\backup";
    includeSysFiles := false;   // Exclude system files from backup
    verifyFiles     := true;    // Verify data during backup
    compressFiles   := false;   // Don't perform on-the-fly data compression
    allowOverwrite  := true;    // Overwrite existing files in directory
    quiesce         := false;   // Don't quiesce => full online backup
    dba.backupAllDbFiles(backupDirectory, includeSysFiles, verifyFiles,
                    compressFiles, allowOverwrite, quiesce, droppedFiles);
    // The backup completed without exceptions
    title           := "Database Backup Complete";
epilog
    if process.isInExceptionState then
        // A fatal exception has occurred during the backup, the activated
        // exception handler aborted the current action - report the failure
        title       := "Database Backup Failed";
    endif;
    // Report missing files (only valid if they have never been created)
    if droppedFiles.size > 0 then
        msg := 'The following files were not backed up :' & CrLf;
        foreach file in droppedFiles do
            msg := msg & file.name & '.dat' & CrLf;
        endforeach;
    endif;
    app.msgBox(msg & Cr, title, MsgBox_Exclamation_Mark_Icon + 65536);
    delete dba;
    delete droppedFiles;
end;
```

backupDbFiles

Signature
```
backupDbFiles(backupDir:      String;
              fileKinds:      Integer;
              verifyFiles:    Boolean;
              compressFiles:  Boolean;
              overwriteFiles: Boolean;
              quiesce:        Boolean;
              droppedFiles:   DbFileArray input) updating;
```

The **backupDbFiles** method of the **JadeDatabaseAdmin** class backs up the kinds (categories) of files specified in the **fileKinds** parameter to the directory specified in the **backupDir** parameter.

The backup directory specified in the **backupDir** parameter must be a valid directory that is relative to the server.

Use the **fileKinds** parameter to select files for backup by their kind, or category group. (For details about the kinds of database files that you can select, see the **DbFile** class **kind** property or "DbFile Class Constants", earlier in this chapter.) You can select multiple file kinds in a single call, by summing the kind constant values. For example, to select user schema files, environmental files, and user data files, you could pass the following value for the **fileKinds** parameter:

```
DbFile.Kind_Environmental + DbFile.Kind_User_Schema + DbFile.Kind_User_Data
```

Set the **verifyFiles** parameter to **true** if you want the backed up file checked, or verified. In a checked file backup, objects are read using the database access routines and object caching mechanisms, and at the same time, a verification of the data and indexes is performed. The verification performs various consistency checks similar to a database certify, to ensure the integrity of the backup. Furthermore, additional checksum information is added to the backup, to allow restore operations to verify the integrity of the backup as the data is restored.

Set the **compressFiles** parameter to **true** if you want to compress backed up data. You can compress data in a checked or an unchecked backup.

Set the **overwriteFiles** parameter to **true** if you want file backups to overwrite existing files in the destination backup directory. When this parameter is **false**, an exception is raised if an existing file is detected.

Use the **quiesce** parameter to allow a quiesced read-only backup transaction to be specified. When you set this parameter to **true**, the database is placed in a quiescent state by first allowing current active transactions to be completed, flushing modified buffers for cache to the stable database. In this mode, physical database files contain all committed updates to the database, and the files are opened in read-only mode with shared read access allowing external backup processes to safely copy database files.

In the quiescent mode, updating transactions are not permitted and attempts to execute database transactions raise a database exception. When a backup is performed in the quiescent mode, the physical database files are guaranteed to contain all database updates and a quiesced backup does not require backup recovery.

Set the **quiesce** parameter to **false** to allow updates during the backup process. When restoring a database backed up fully online (that is, this parameter is set to **false**), the restoration process requires the recovery of backed up transaction journals. A backup recovery is required after restoring files that were fully backed up online (that is, the **quiesce** parameter was set to **false**).

The **droppedFiles** parameter specifies an array of the database files that are not backed up.

backupJournal

Signature
```
backupJournal(number:         Integer;
              sourceDir:      String;
              backupDir:      String;
              verify:         Boolean;
              compress:       Boolean;
              overwriteDest:  Boolean);
```

The **backupJournal** method of the **JadeDatabaseAdmin** class backs up the transaction journal identified by the **number** parameter from the journal directory specified in the **sourceDir** parameter to the directory specified in the **backupDir** parameter. (The backup directory must be a valid directory relative to the server.) If the value of the **sourceDir** parameter is **null**, the current journal directory is used.

Set the **verify** parameter to **true** if you want the backed up file checked, or verified. In a checked file backup, objects are read using the database access routines and object caching mechanisms, and at the same time, a verification of the data and indexes is performed. The verification performs various consistency checks similar to a database certify, to ensure the integrity of the backup. Furthermore, additional checksum information is added to the backup, to allow restore operations to verify the integrity of the backup as the data is restored.

Set the **compress** parameter to **true** if you want to compress backed up data. You can compress data in a checked or an unchecked backup.

When both the **verify** and **compress** parameters are set to **false**, a fast file backup is performed. In a fast file backup, database files are backed up in a similar fashion to a standard file copy, using large buffers and asynchronous I/O to speed up the copy process. The fast backup mode bypasses the database access-routines and cache management, and does not verify data as it is backed up.

Set the **overwriteDest** parameter to **true** if you want transaction journal backups to overwrite existing journals in the destination backup directory. When this parameter is **false**, an exception is raised if an existing transaction journal is detected. The following example shows the use of the **backupJournal** method.

```
vars
    dba                                          : JadeDatabaseAdmin;
    verifyFiles, compressFiles, allowOverwrite : Boolean;
    backupDirectory                              : String;
    jnlNum                                       : Integer;
begin
    create dba;
    jnlNum          := dba.getCurrentJournalNumber;
    backupDirectory := "z:\jade\backup5";
    verifyFiles     := true;
    compressFiles   := false;
    allowOverwrite  := true;
    dba.backupJournal(jnlNum - 1, null, backupDirectory, verifyFiles,
                      compressFiles, allowOverwrite);
epilog
    delete dba;
end;
```

beginBackup

Signature `beginBackup(backupDir: String; quiesced: Boolean);`

The **beginBackup** method of the **JadeDatabaseAdmin** class signals the start of a database backup transaction in which multiple database files are copied.

The **backupDir** parameter specifies a default location for file backups and the directory where control files are copied when the backup is committed. (The backup directory must be a valid directory that is relative to the server.)

Use the **quiesced** parameter to specify whether updates are allowed during the backup process (by setting this parameter to **false**) or if the backup is to be performed in read-only mode. When restoring a database backed up fully online (that is, this parameter is set to **false** and the database was available for both read and write access at the time of the backup), the restoration process requires the recovery of backed up transaction journals. (For details, see "Using the Restore Database Command", in Chapter 1 of the *JADE Database Administration Guide*.) When a database is backed up in the quiescent mode, the physical database files are guaranteed to contain all database updates, and a recovery is not required when the backed up database is restored.

Set the **quiesced** parameter to **true** to specify that the database is locked for write access during the backup operation and to enable archival recovery. (Database records can still be read in this mode, although not updated.) When a database is backed up in a quiescent mode, any threads that attempt to start a new transaction are first blocked, waiting for pending transactions to complete. All dirty buffers for objects resident in the file are flushed and the file is locked against further updates. At this point, threads blocked by the **beginTransaction** instruction are allowed to continue. (Database operations that attempt to update a file in read-only mode receive an exception.)

The code fragment in the following example shows the use of the **beginBackup** method.

```
// Begin a backup transaction.  The quiesce parameter is false, meaning
// that a 'hot' or online backup will be performed, allowing online
// updating to continue during the backup processing.
self.dba.beginBackup(defaultDirectory, false /*=> online backup*/);
```

changeDbAccessMode

Signature `changeDbAccessMode(mode: Integer; usage: Integer) updating;`

The **changeDbAccessMode** method of the **JadeDatabaseAdmin** class changes the access mode of the database.

Use the **mode** parameter to specify the database access mode that you require. The values for the **mode** parameter are defined by the **JadeDatabaseAdmin** class constants listed in the following table.

Constant	Integer Value	Description
Mode_Archive	4	Quiesces the database for archive backup.
Mode_Default	9	Restores the database to the initial mode and usage values.
Mode_Exclusive	1	Requests exclusive access to the database. If other users already have the database open, a database mode conflict is reported. Similarly, if one user already has the database open in exclusive mode, other users are prevented from opening the database.

Constant	Integer Value	Description
Mode_Shared	0	Enables multiple current users to open the database.
Mode_Snapshot	6	Conditions the database for external third-party snapshot backup.

When a primary changes to archive mode, the SDS service, if active, is stopped. The service is restarted as necessary when exiting from archive mode.

For details about external third-party snapshot backups, see "Non-JADE Backups" in the *Developing a Backup Strategy White Paper*.

Use the **usage** parameter to specify the database usage that you require. The values for the **usage** parameter are defined by the **JadeDatabaseAdmin** class constants listed in the following table.

Constant	Integer Value	Description
Usage_NoAudit	2	Specifies that the recovery log is not maintained. This usage can result in better performance, but if an abnormal termination occurs and the database is not audited, recovery is not possible and the database must be restored from backup. Use the shared access mode to specify multiple users with no-audit usage. However, if any one application or shared-access user opens the database in no-audit mode, all users must request no-audit usage or a conflicts exception is raised.
Usage_ReadOnly	1	Specifies that applications have read-only access to the database. Multiple instances of the JADE Object Manager can access the database in read-only mode.
Usage_Update	0	Enables applications to update the database.

closeCurrentJournal

Signature `closeCurrentJournal();`

The **closeCurrentJournal** method of the **JadeDatabaseAdmin** class closes the active transaction journal and switches to a new transaction journal.

commitBackup

Signature `commitBackup();`

The **commitBackup** method of the **JadeDatabaseAdmin** class signals the successful completion of a database backup transaction in which multiple database files have been copied. This method triggers the copying of the database control file (and if archival recovery is disabled, the current transaction journal) to the default backup directory specified in the **beginBackup** method.

If the database is in a quiescent read-only mode, the **commitBackup** method ends this mode, permitting updating transactions to be processed.

The code fragment in the following example shows the use of the **commitBackup** method.

```
if not self.backupCancelled then
    // Commit the database backup transaction, which takes the database
    // out of backup state and finalizes the backup
    self.dba.commitBackup;
endif;
```

Note The **commitBackup** method also marks an online database backup as valid. If this has not completed, the backup cannot be used.

commitCoherentBackup

Signature `commitCoherentBackup();`

The **commitCoherentBackup** method of the **JadeDatabaseAdmin** class signals the successful completion of a database backup transaction on an SDS secondary.

This method arranges a database quiet point on the primary to establish commit-consistent end backup recovery state that is stored in the database control file, which is then backed up to the default backup directory specified in the **beginBackup** method. Backup recovery will stop tracking at end backup, saving replay state.

The commit-consistent checkpoint processing on the primary performs a checkpoint within a database quiet point. The primary **MaxWaitForQuietPoint** value specified in the [PersistentDb] section of the JADE initialization file is used when waiting for the quiet point.

You may encounter an exception 3077 (*Maximum time to wait for a quiet point was exceeded*), in which case, the backup remains active and you can retry the **commitCoherentBackup** method call or you can call the **abortBackup** method. The backup terminates with the exception if it is unhandled. Other errors (for example, the 3221 (*SDS primary not connected*) or 3212 (*SDS a response was not received within a reasonable timeframe*) error) terminate the backup with that exception.

compactDbFiles

Signature
```
compactDbFiles(dbFiles:         DbFileArray;
               workers:         Integer;
               workPath:        String;
               updatesAllowed:  Boolean);
```

The **compactDbFiles** method of the **JadeDatabaseAdmin** class compacts the files specified in the **dbFiles** parameter to optimize the use of storage and to reduce fragmentation.

The **workers** parameter enables you to specify the number of workers that you require. If you want to override the reorganization work and backup directories, specify a valid path in the **workPath** parameter. The **updatesAllowed** parameter specifies whether an updating or a read-only compact is performed.

See also the **ReorgBackupDirectory** and **ReorgWorkDirectory** parameters in the [JadeReorg] section of the JADE initialization file.

createRpsDatabase

Signature
```
createRpsDatabase(backupDir:      String;
                  schema:         Schema;
                  rpsMapping:     RelationalView;
                  rpsStorageMode: Integer;
                  verifyFiles:    Boolean;
                  overwriteFiles: Boolean;
                  quiesce:        Boolean);
```

The **createRpsDatabase** method of the **JadeDatabaseAdmin** class creates an RPS database for a specified schema and RPS mapping programmatically in the directory specified in the **backupDir** parameter. The backup directory must be a writable valid directory that is relative to the server.

Note This method can be executed on the primary system only.

Specify the schema and RPS mapping (that is, the RPS name) for which the RPS database is to be created in the **schema** and **rpsMapping** parameters, respectively.

Specify the storage mode of the created RPS database in the **rpsStorageMode** parameter, using one of the **JadeDatabaseAdmin** class constants listed in the following table.

Class Constant	Integer Value	Description
RpsStorageMode_Full	0	Full database replica RPS data store mode
RpsStorageMode_MappedExtent	1	Mapped extent RPS data store mode
RpsStorageMode_WorkingSet	2	Working set RPS data store mode

For details about the RPS database storage modes, see "RPS Data Store", see Chapter 2, "Relational Population Service (RPS) Support", in the *JADE Synchronized Database Service (SDS) Administration Guide*.

Set the **verifyFiles** parameter to **true** if you want the RPS mapping entities checked, or verified. In a checked RPS database creation, entities are read using the database access routines and object caching mechanisms, and at the same time, a verification of the data and indexes is performed. The verification performs various consistency checks similar to a database certify, to ensure the integrity of the RPS database creation. Furthermore, additional checksum information is added to the RPS database creation, to allow restore operations to verify the integrity of the RPS system.

Set the **overwriteFiles** parameter to **true** if you want to allow RPS database entities to overwrite existing entities in the destination RPS database. When this parameter is **false**, an exception is raised if an existing entity is detected.

Use the **quiesce** parameter to allow a quiesced read-only database transaction to be specified. When you set this parameter to **true**, the database is placed in a quiescent state by first allowing current active transactions to be completed and flushing modified buffers for cache to the stable database. In this mode, physical database files contain all committed updates to the database, and the files are opened in read-only mode with shared read access allowing the RPS create process to safely copy database files. In the quiescent mode, updating transactions are not permitted and attempts to execute database transactions raise a database exception. When an RPS database creation is performed in the quiescent mode, the physical database files are guaranteed to contain all database updates and a quiesced RPS database does not require backup recovery.

Set the **quiesce** parameter to **false** to allow updates during the RPS database creation process. When restoring an RPS database created with this option (that is, **false**), the restoration process requires the recovery of database transaction journals, which will be automatically transferred from the primary and the transactions applied.

disableByteProgressEvents

Signature `disableByteProgressEvents();`

The **disableByteProgressEvents** method of the **JadeDatabaseAdmin** class disables the notification of operation and progress events reporting the number of bytes of a file that have been backed up. Operation and progress events are disabled by default. For details about operation and progress events, see "JadeDatabaseAdmin Class Event Notifications" and "DbFile Class Event Notifications", elsewhere in this chapter.

The following example shows the use of the **disableByteProgressEvents** method.

```
finalise();
begin
    // Disable backup progress (as a number of bytes) events
    self.dba.disableByteProgressEvents;
end;
```

disableProgressEvents

Signature `disableProgressEvents();`

The **disableProgressEvents** method of the **JadeDatabaseAdmin** class disables the notification of operation and progress events reporting the percentage of a file that has been backed up. Operation and progress events are disabled by default.

For details about operation and progress events, see "JadeDatabaseAdmin Class Event Notifications" and "DbFile Class Event Notifications", elsewhere in this chapter.

The following example shows the use of the **disableProgressEvents** method.

```
finalise();
begin
    // Disable backup progress events
    self.dba.disableProgressEvents;
end;
```

doCheckpoint

Signature `doCheckpoint();`

The **doCheckpoint** method of the **JadeDatabaseAdmin** class causes a database checkpoint operation to be queued to the database worker thread.

This method is intended for use by backup algorithms and mechanisms that need knowledge of journal activity and recovery restart points and which require the ability to cause recovery restart point re-evaluation. See also the **getLastCheckpoint** method.

Caution The processing burden injected by the initiation of a checkpoint is not easily determined, as system activity is a major factor; however, there is always a cost to performance, as I/O must be performed. Inappropriate use of the **doCheckpoint** method could cause severe performance degradation.

doQuietpoint

Signature
```
doQuietpoint(maxWaitForQuietpoint: Integer;
             switchJournal       : Boolean
             tranID              : Integer64 output;
             journal             : Integer64 output;
             offset              : Integer64 output);
```

The **doQuietpoint** method of the **JadeDatabaseAdmin** class attempts to establish a database quiet point. If the **maxWaitForQuietpoint** parameter has a non-zero value, it specifies the maximum time in seconds that the operation will wait for there to be no transaction activity, and overrides the configured or default database value specified by the **MaxWaitForQuietPoint** parameter in the [PersistentDb] section of the JADE initialization file.

If a quiet point cannot be established, exception 3077 (*Maximum time to wait for quiet point was exceeded*) is raised.

The **switchJournal** parameter, if set to **true**, causes the journal to be switched when the quiet point is established.

The **tranID**, **journal**, and **offset** output parameters contain the next transaction identifier, and the next audit LSN (**journal** and **offset**) values as they were at the database quiet point.

enableByteProgressEvents

Signature `enableByteProgressEvents(increment: Integer);`

The **enableByteProgressEvents** method of the **JadeDatabaseAdmin** class enables the notification of operation and progress events for file backups at the progress interval specified in the **increment** parameter, which represents a number of bytes. A value of zero (**0**) for the **increment** parameter specifies the lowest allowed value of 128K bytes.

When operation and progress event notifications are enabled:

- A progress event is notified by each **DbFile** instance whenever the specified number of bytes of the file has been copied.
- An operation event is notified by each **DbFile** instance of the backup operation being performed on the file.

For details, see "JadeDatabaseAdmin Class Event Notifications" and "DbFile Class Event Notifications", elsewhere in this chapter.

The following example shows the use of the **enableByteProgressEvents** method.

```
initialise(backupDir: String;
           compress, verify, includeSystemFiles: Boolean;
           overwrite, quiesce: Boolean) updating;
begin
    // Save backup parameters for calls to database backup methods
    defaultDirectory := backupDir;
    compressFiles    := compress;
    verifyFiles      := verify;
    includeSysFiles  := includeSystemFiles;
    overwriteFiles   := overwrite;
    quiescedBackup   := quiesce;
    // Enable backup progress events to occur in increments of 1000000 bytes
    self.dba.enableByteProgressEvents(1000000);
end;
```

Note Enabling operation and progress notification is likely to have some impact on the elapsed time of file backups.

enableProgressEvents

Signature `enableProgressEvents(increment: Integer);`

The **enableProgressEvents** method of the **JadeDatabaseAdmin** class enables the notification of operation and progress events for file backups at the progress interval specified in the **increment** parameter, which represents a percentage of the size of the file.

When operation and progress event notifications are enabled:

- A progress event is notified by each **DbFile** instance whenever the specified percentage increment of the file has been copied.
- An operation event is notified by each **DbFile** instance of the backup operation being performed on the file.

For details, see "JadeDatabaseAdmin Class Event Notifications" and "DbFile Class Event Notifications", elsewhere in this chapter.

The following example shows the use of the **enableProgressEvents** method.

```
initialise(backupDir: String;
           compress, verify, includeSystemFiles: Boolean;
           overwrite, quiesce: Boolean) updating;
begin
    // Save backup parameters for calls to database backup methods
    defaultDirectory := backupDir;
    compressFiles    := compress;
    verifyFiles      := verify;
    includeSysFiles  := includeSystemFiles;
    overwriteFiles   := overwrite;
    quiescedBackup   := quiesce;
    // Enable backup progress events to occur in increments of 4% or greater
    self.dba.enableProgressEvents(4);
end;
```

Note Enabling operation and progress notification is likely to have some impact on the elapsed time of file backups.

getAbortJournalNumber

Signature `getAbortJournalNumber(): Integer;`

The **getAbortJournalNumber** method of the **JadeDatabaseAdmin** class returns the number of the journal containing the Begin Transaction record of the oldest active transaction. When this method is called on a secondary database, it returns the oldest journal required for an undo operation in the event of a hostile takeover or the next required replay journal if no journals have been replayed in the current session.

If there are no active transactions, the number of the oldest journal required for recovery is returned.

getAllDbFiles

Signature `getAllDbFiles(dbfiles: DbFileArray input);`

The **getAllDbFiles** method of the **JadeDatabaseAdmin** class populates a **DbFile** array with references to all database files found by searching all schemas from the RootSchema down through the schema hierarchy.

getArchiveJournalDirectory

Signature `getArchiveJournalDirectory(): String;`

The **getArchiveJournalDirectory** method of the **JadeDatabaseAdmin** class returns the name of the archive directory for transaction journals.

getCreationTimestamp

Signature `getCreationTimestamp(): TimeStamp;`

The **getCreationTimestamp** method of the **JadeDatabaseAdmin** class returns a timestamp containing the date and time the database was created.

getCurrentJournalDirectory

Signature `getCurrentJournalDirectory(): String;`

The **getCurrentJournalDirectory** method of the **JadeDatabaseAdmin** class returns the name of the current directory for transaction journals.

getCurrentJournalName

Signature `getCurrentJournalName(): String;`

The **getCurrentJournalName** method of the **JadeDatabaseAdmin** class returns the name of the current active transaction journal.

getCurrentJournalNumber

Signature `getCurrentJournalNumber(): Integer;`

The **getCurrentJournalNumber** method of the **JadeDatabaseAdmin** class returns the number of the current active transaction journal.

When this method is called on a secondary database, it returns the last replayed journal number or the next required replay journal if no journals have been replayed in the current session.

getCurrentJournalOffset

Signature
```
getCurrentJournalOffset(currentJournal:    Integer64 output;
                        currentOffset:     Integer64 output;
                        lastSwitchJournal: Integer64 output;
                        lastSwitchOffset:  Integer64 output;
                        nominalSize:       Integer64 output);
```

The **getCurrentJournalOffset** method of the **JadeDatabaseAdmin** class retrieves the journal number and byte offset of the last record written to the journal in the respective **currentJournal** and **currentOffset** parameters.

The **lastSwitchJournal** and **lastSwitchOffset** parameters retrieve the respective journal number and byte offset of the last record written to the penultimate journal.

The **nominalSize** parameter returns the nominal size of a journal file (that is, the value of the **JournalMaxSize** parameter in the [PersistentDb] section of the JADE initialization file).

These values enable you to calculate amounts and rates of journal output.

Use this method in conjunction with the **JadeDatabaseAdmin** class **sdsGetSecondaryProxy** method to determine the amount of journal data that has not been sent to the secondary.

getDbFiles

Signature
```
getDbFiles(fileKinds: Integer;
           dbfiles:   DbFileArray input);
```

The **getDbFiles** method of the **JadeDatabaseAdmin** class populates a **DbFile** array with references to database files of the kinds specified in the **fileKinds** parameter, by searching all schemas from the **RootSchema** down through the schema hierarchy.

Use the **fileKinds** parameter to select files for backup by their kind, or category group. (For details about the kinds of database files that you can select, see the **DbFile** class **kind** property or "DbFile Class Constants", earlier in this chapter.)

You can select multiple file kinds in a single call, by summing the kind constant values. For example, to select user schema files, environmental files, and user data files, you could pass the following value for the **fileKinds** parameter:

```
DbFile.Kind_Environmental + DbFile.Kind_User_Schema + DbFile.Kind_User_Data
```

The code fragment in the following example shows the use of the **getDbFiles** method.

```
// Obtain an array of references to user data and environmental files
create dbfiles transient;
self.dba.getDbFiles(DbFile.Kind_User_Data + DbFile.Kind_Environmental,
                dbfiles);
foreach dbFile in dbfiles do
// Since we have enumerated environmental files, we must exclude files
// whose excludeFromBackup attribute is set; for example, _environ.dat
    if not dbFile.excludeFromBackup then
    dbFile.backupFile(null,         // Use default directory
                      true,         // Verify during backup
                      true,         // Request data compression
                      false);       // Disallow overwrite of existing files
    endif;
    // Note that the backupCancelled attribute can be set (using a call
    // to cancelBackup) only when this method is executed asynchronously
    if self.backupCancelled then
        break;
    endif;
endforeach;
```

getLastCheckpoint

Signature `getLastCheckpoint(journal: Integer64 output;`
`offset: Integer64 output);`

The **getLastCheckpoint** method of the **JadeDatabaseAdmin** class retrieves the journal number and byte offset of the last database checkpoint in the **journal** and **offset** parameters, respectively.

This method is intended for use by backup algorithms and mechanisms that need knowledge of journal activity and recovery restart points and require the ability to cause recovery restart point re-evaluation. See also the **doCheckpoint** method.

getLatestBackupTimestamp

Signature `getLatestBackupTimestamp(): TimeStamp;`

The **getLatestBackupTimestamp** method of the **JadeDatabaseAdmin** class returns a timestamp containing the date and time the database was last backed up without error.

getLatestFullBackupTimestamp

Signature `getLatestFullBackupTimestamp(): TimeStamp;`

The **getLatestFullBackupTimestamp** method of the **JadeDatabaseAdmin** class returns a timestamp containing the date and time the database was last backed up without error and the backup included all database files; that is, the backup was a full backup.

getOpenTimestamp

Signature `getOpenTimestamp(): TimeStamp;`

The **getOpenTimestamp** method of the **JadeDatabaseAdmin** class returns a timestamp containing the most recent date and time the database was opened.

getReasonTrackingStoppedString

Signature `getReasonTrackingStoppedString(reason : Integer): String;`

The **getReasonTrackingStoppedString** method of the **JadeDatabaseAdmin** class returns a string containing a textual description (for example, **"Transition Halt"**) of the **SDSStopTrackingCodes** category global constant reason code passed in the **userInfo** parameter of the system **SDS_TrackingStopped** event.

The textual descriptions that can be returned are listed in the following table.

SDSStopTrackingCodes Category Global Constant	**Textual Description**
SDS_ReasonAdminAudited	Admin Audited
SDS_ReasonAdminDirect	Admin Direct
SDS_ReasonAutoUpgradeMismatch	Requires Upgrade
SDS_ReasonDeltaModeEntered	In delta mode
SDS_ReasonEnablingDbCrypt	Enabling Database Encryption
SDS_ReasonErrorHalt	Error Halt
SDS_ReasonRestart	Tracking restart
SDS_ReasonRpsAdminHalt	RPS Admin Halt
SDS_ReasonRpsReorgHalt	RPS Reorg Halt
SDS_ReasonRpsRestart	Data pump restart
SDS_ReasonRpsSnapshot	RPS Snapshot
SDS_ReasonTakeover	Takeover Halt
SDS_ReasonTransition	Transition Halt

getRpsMappedFiles

Signature `getRpsMappedFiles(schemaName: String; rpsMapName: String; dbfiles: DbFileArray input);`

The **getRpsMappedFiles** method of the **JadeDatabaseAdmin** class populates the **DbFile** array specified by the **dbfiles** input parameter with references to database files that are required for an RPS mapping. The RPS mapping is specified by the **rpsMapName** and **schemaName** parameters.

The following code example shows the use of the **getRpsMappedFiles** method.

```
vars
    dba : JadeDatabaseAdmin;
    dbfiles : DbFileArray;
begin
    create dba transient;
    create dbfiles transient;
    dba.getRpsMappedFiles("TestSchema","TestRpsMapping",dbfiles);
    // dbfiles now contains the database files for the RPS mapping
epilog
    delete dba;
    delete dbfiles;
end;
```

isArchival

Signature `isArchival(): Boolean;`

The **isArchival** method of the **JadeDatabaseAdmin** class returns **true** if database archival recovery is enabled for the database server node on which the JADE system is running.

rpsAuditSqlScriptForReplay

Signature `rpsAuditSqlScriptForReplay(sql: String);`

The **rpsAuditSqlScriptForReplay** method of the **JadeDatabaseAdmin** class, when invoked on the primary system, writes a special callback audit record to the current journal. The callback record contains the contents of the **sql** parameter.

When an RPS node replays the callback audit record, the SQL string stored in the record is passed to the **Datapump** application for execution. If the script was audited within a transaction on the primary, it is executed before that transaction has been replayed on the target relational database.

If execution of the SQL script encounters an error, the script contents are saved to a file and database replication is halted. The error file is in the format **Replay_*YYYYMMDD_HHMMSS*.log** and is saved in a **Failed** subdirectory of the directory specified in the **AutoScriptPath** parameter in the [JadeRps] section of the JADE initialization file.

An administrator can investigate what went wrong and take corrective action before restarting the **Datapump** application on the RPS node. On restart, audited SQL scripts that failed are skipped.

rpsExtractData

Signature

```
rpsExtractData(tableName:            String;
               executionLocation:    Integer;
               scriptFilePath:       String;
               dataFilesPath:        String;
               rdbDataFilesPath:     String;
               rdbName:              String;
               loadHistoricalTables: Boolean;
               serverName:           String;
               extractWorkers:       Integer): Process;
```

The **rpsExtractData** method of the **JadeDatabaseAdmin** 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.

Note You can execute this method on an RPS node only, not on the primary node. Running an RPS extract on an SDS node causes tracking to be stopped during the extract.

The **rpsExtractData** 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 **RelationalView** class **Load_ServerExecute** (0) and **Load_ClientExecute** (1) 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.
loadHistoricalTables	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

rpsExtractDataAll

Signature

```
rpsExtractDataAll(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 **rpsExtractDataAll** method of the **JadeDatabaseAdmin** 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.

Note You can execute this method on an RPS node only, not on the primary node. Running an RPS extract on an SDS node causes tracking to be stopped during the extract.

The **rpsExtractDataAll** method parameters are listed in the following table.

Parameter	Specifies the …
executionLocation	The location that will be used for loading the extracted data. Permitted values can be specified using the **RelationalView** class **Load_ServerExecute** (0) and **Load_ClientExecute** (1) 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 following **JadeDatabaseAdmin** class constants.

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 extraction)
ExtractOrderSelectedFirst	2	As specified in **extractFirst** parameter, then in default order

Parameter	Specifies the …
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 data pump application. If null, the default data pump 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.

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.

rpsExtractDataUsingIniOptions

Signature `rpsExtractDataUsingIniOptions(tableName: String): Process;`

The **rpsExtractDataUsingIniOptions** method of the **JadeDatabaseAdmin** 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.

Note You can execute this method on an RPS node only, 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 the [JadeRps] section of the JADE initialization file.

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

rpsGetDatabaseParameters

Signature
```
rpsGetDatabaseParameters(schemaName:     String output;
                         rpsMappingName: String output;
                         storageMode:    Integer output);
```

The **rpsGetDatabaseParameters** method of the **JadeDatabaseAdmin** class returns the name of the schema, the name of the RPS mapping, and the database replication mode (**Working Set**, **Mapped Extent**, or **Full**) of the RPS node.

An exception is raised if the database is not an RPS database.

rpsStartDataPump

Signature `rpsStartDataPump(userName: String;`
`password: String): Process updating;`

The **rpsStartDataPump** method of the **JadeDatabaseAdmin** class enables you to programmatically start the RPS **Datapump** application to resume tracking on an RPS node.

The value of the **userName** and **password** parameters can be **null**. If the values are not null, they are used to connect to the RDBMS.

This method returns the process for the RPS **Datapump** application if the application initializes successfully. If the application starts but does not initialize successfully, the reason for the failure is written to the **jommsg.log** file. An exception is raised if the database is not an RPS database or if the **Datapump** application is already running.

The **Datapump** application is configured using the RPS Manager utility Configure RPS Node dialog on the RPS node and the configuration information is stored in the [JadeRps] section of the JADE initialization file.

rpsStopDataPump

Signature `rpsStopDataPump(): Boolean;`

The **rpsStopDataPump** method of the **JadeDatabaseAdmin** class enables you to programmatically stop the RPS **Datapump** application on the RPS node. This method returns **true** if the application is stopped successfully, or **false** if it is not running.

An exception is raised if the database is not an RPS database or if the **Datapump** application fails to stop.

sdsAuditStopTracking

Signature
```
sdsAuditStopTracking(scope:   Integer;
                     reason:  Integer;
                     journal: Integer output;
                     offset:  Integer output);
```

The **sdsAuditStopTracking** method of the **JadeDatabaseAdmin** class specifies a system event number that is returned in the **userInfo** parameter of your user notification method when an **SDS_TrackingStopped** event occurs because of a programmatic, RPS node, or SDS Administration utility action or tracking is halted because of an error, to notify subscribers that tracking has been disabled.

When this method is invoked on the primary, it causes a **Stop Tracking** audit record to be written to the current journal. When this record is replayed on an RPS node or on an SDS secondary, tracking halts at that point in the audit trail.

The value of the **scope** parameter determines the type of secondary databases that actions the stop tracking command, and is represented by the **SDSStopTrackingCodes** category global constants listed in the following table.

Global Constant	Integer Value	Description
SDS_AuditStopTrackingAll	1	Stops tracking on all JADE and RPS secondary databases
SDS_AuditStopTrackingNative	2	Stops tracking on JADE native secondary databases
SDS_AuditStopTrackingRdb	3	Stops tracking on RPS secondary databases

The **reason** parameter determines the reason tracking was disabled, and is represented by the **SDSStopTrackingCodes** category global constants listed in the following table. This value is audited and passed to subscribers to the **SDS_TrackingStopped** system event in the userInfo parameter of the associated **userNotify** callback method.

Global Constant	Integer Value
SDS_ReasonAdminAudited	1
SDS_ReasonAdminDirect	2
SDS_ReasonAutoUpgradeMismatch	6
SDS_ReasonDeltaModeEntered	12
SDS_ReasonEnablingDbCrypt	13
SDS_ReasonErrorHalt	8
SDS_ReasonRestart	10
SDS_ReasonRpsAdminHalt	4
SDS_ReasonRpsReorgHalt	9
SDS_ReasonRpsRestart	11
SDS_ReasonRpsSnapshot	3
SDS_ReasonTakeover	7
SDS_ReasonTransition	5

The **SDS_ReasonTakeover** value indicates that tracking stopped during a takeover operation and the **SDS_ReasonRpsReorgHalt** value indicates that tracking stopped at transition.

The **SDS_ReasonErrorHalt** value indicates that tracking halted due to an error condition (the error code is saved in the **SDSSecondary** or **SDSSecondaryProxy** dynamic **lastErrorCode** attribute).

The **journal** and **offset** output parameters contain the journal number and byte offset within the journal of the **Stop Tracking** audit record. These two values together comprise a Log Sequence Number (LSN). When tracking is restarted on the secondary or RPS node, it resumes at the next audit record.

Notes Calling this method neither forces a quiet point nor closes the current journal.

The main purpose for this in an RPS context is to establish a journal trigger that coincides with a point-in-time on the primary database, to enable establishing a snapshot of the mapped extent in the target database frozen at that time.

sdsDisablePrimaryConnection

Signature `sdsDisablePrimaryConnection();`

The **sdsDisablePrimaryConnection** method of the **JadeDatabaseAdmin** class causes the secondary server or RPS node to close the connection to its primary server, if open, leaving the connection closed. This places a secondary database or RPS node in an offline mode in which it can continue tracking and providing read-only database access while not receiving further journals from the primary database.

Use the **sdsReconnectNow** method to attempt to re-establish a connection to the primary.

You can also use the **sdsDisablePrimaryConnection** method to clear and re-establish a disrupted network link. In the event that a connection becomes disrupted because of a problem in the network path, call the **sdsDisablePrimaryConnection** method to drop the failed connection and then the **sdsReconnectNow** method to attempt to establish a fresh network connection.

sdsDisablePrimaryConnectionAt

Signature `sdsDisablePrimaryConnectionAt(secondaryName: String);`

The **sdsDisablePrimaryConnectionAt** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, causes the secondary server specified in the **secondaryName** parameter to close the connection to its primary server, if open, leaving the connection closed. This places a secondary database in an offline mode in which it can continue tracking and providing read-only database access while not receiving further journals from the primary database.

sdsDisableReadAccess

Signature `sdsDisableReadAccess();`

The **sdsDisableReadAccess** method of the **JadeDatabaseAdmin** class, valid only if called from a secondary database system, disables read access to persistent objects in the database.

When read access is disabled, inquiry applications are not allowed access to persistent objects resident in the database. Any attempt by a user application to access persistent objects when read access is disabled raises an exception.

If successful, the value of the **ReadAccessDisabled** parameter in the [SyncDbService] section of the JADE initialization file is updated to **true** on the current secondary database server so that the setting is preserved when the server restarts.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsDisableReadAccessAt

Signature `sdsDisableReadAccessAt(secondaryName: String);`

The **sdsDisableReadAccessAt** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, disables read access to the secondary database that has the **MyName** parameter value in the [SyncDbService] section of the JADE initialization file matching the name specified in the **secondaryName** parameter.

When read access is disabled on the secondary database, inquiry applications cannot access persistent objects resident in the database server. Any attempt by a user application to access persistent objects when read access is disabled raises an exception.

If successful, the value of the **ReadAccessDisabled** parameter in the [SyncDbService] section of the JADE initialization file on the specified secondary database server is updated to **true** so that the setting is preserved when the server restarts.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsEnableReadAccess

Signature `sdsEnableReadAccess();`

The **sdsEnableReadAccess** method of the **JadeDatabaseAdmin** class, valid only if called from a secondary database system, requests the granting of read access to persistent objects in the database.

Read access is not granted immediately when interrupted transactions remain pending after a restart operation. Read access is granted only when all remaining interrupted transactions complete. Use the **sdsGetTransactions** method to determine the status of active (that is, incomplete) transactions.

If successful, the value of the **ReadAccessDisabled** parameter in the [SyncDbService] section of the JADE initialization file is updated to **false** on the current secondary database server so that the setting is preserved when the server restarts.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsEnableReadAccessAt

Signature `sdsEnableReadAccessAt(secondaryName: String);`

The **sdsEnableReadAccessAt** method of the **JadeDatabaseAdmin** class, valid only from a primary database system, requests the granting of read access to persistent objects in the secondary database specified in the **secondaryName** parameter.

Read access is not granted immediately when interrupted transactions remain pending after a restart operation. Read access is granted only when all remaining interrupted transactions complete. Use the **sdsGetTransactionsAt** method to determine the status of active (that is, incomplete) transactions on the secondary database server.

If successful, the value of the **ReadAccessDisabled** parameter in the [SyncDbService] section of the JADE initialization file is updated to **false** on the current secondary database server so that the setting is preserved when the server restarts.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsGetDatabaseRole

Signature `sdsGetDatabaseRole(): Integer;`

The **sdsGetDatabaseRole** method of the **JadeDatabaseAdmin** class returns an integer value that represents the database role of the server on which the method is executed.

Tip Use the **System** class **getDatabaseRole** method 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 enumerated by the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value
SDS_RolePrimary	1
SDS_RoleSecondary	2

Global Constant	Integer Value
SDS_RoleUndefined (returned when the method is invoked on a non-SDS-capable or non-RPS-capable system)	0

sdsGetDatabaseSubrole

Signature `sdsGetDatabaseSubrole(): Integer;`

The **sdsGetDatabaseSubrole** method of the **JadeDatabaseAdmin** class returns an integer value that represents the database subrole of the server on which the method is executed.

Tip Use the **System** class **getDatabaseSubrole** method 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 enumerated by 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
SDS_SubroleNative	1
SDS_SubroleRelational	2

sdsGetMyServerInfo

Signature `sdsGetMyServerInfo(myAttributes: JadeDynamicObject input);`

The **sdsGetMyServerInfo** method of the **JadeDatabaseAdmin** class, valid at the primary and secondary databases and RPS nodes, populates a **JadeDynamicObject** instance describing the SDS or RPS attributes of the JADE system. The caller is responsible for the creation and deletion of the input dynamic object parameter.

When this method is invoked from the primary database system, the input parameter is converted into an **SDSPrimary** dynamic object, whose **name** attribute has a value of **SDSPrimary** and **type** attribute has a value of **SDS_PrimaryType** (1).

The primary dynamic attributes are listed in the following table.

Name	Type	Description
connectedSecondaryServers	Integer	Number of secondary servers connected to the primary
currentJournalNumber	Integer	Number of the current write journal
currentJournalTimeStamp	TimeStamp	Timestamp of the current journal converted to a local value
currentJournalTimeStampUTC	TimeStamp	Timestamp of the current journal as a UTC value

Name	Type	Description
latestCommittedTimestamp	TimeStamp	Timestamp of the latest commit audit record appended to the audit trail converted to a local value
latestCommittedTimestampUTC	TimeStamp	Timestamp of the latest commit audit record appended to the audit trail as a UTC value
maxCommittedTranID	Decimal	Highest transaction id committed by the primary database, which may not be the latest transaction committed
myHostName	String	Computer name of the primary host
myName	String	Name of the primary, specified in the **MyName** parameter in the [SyncDbService] section of the JADE initialization file

The audit timestamp attributes with a UTC suffix hold UTC values that are not converted to local time. The audit timestamp attributes without a UTC suffix hold UTC values that are converted to local time. The timestamps of both the primary and secondary or RPS node are both derived from the UTC audit timestamp recorded by the primary and converted to local time on the secondary (catering for primary and secondary or RPS node time zones that differ).

When this method is invoked from a secondary database system or RPS node, the input parameter is converted into an **SDSSecondary** dynamic object, whose **name** attribute has a value of **SDSSecondary** and **type** attribute has a value of **SDS_SecondaryType** (3).

The secondary or RPS node dynamic attributes are listed in the following table.

Name	Type	Description
activeTransactions	Integer	Number of active transactions
connectionCheckInterval	Integer	Number of seconds at which the secondary database or RPS node polls the primary to determine reachability via the communication paths, specified in the **ConnectionPollInterval** parameter in the [SyncDbService] section of the JADE initialization file
connectionState	Integer	State of the connection to the primary (see the first of the following tables)
currentReplayJournalNumber	Integer	Number of the journal currently replaying
currentReplayJournalTimeStamp	TimeStamp	Timestamp of the journal currently replaying converted to a local value
currentReplayJournalTimeStampUTC	TimeStamp	Timestamp of the journal currently replaying as a UTC value
interruptedTransactions	Integer	Number of interrupted transactions
lastErrorCode	Integer	Number of the last error that occurred

Name	Type	Description
lastReplayJournalNumber	Integer	Number of the last journal that was replayed
lastReplayJournalTimeStamp	TimeStamp	Timestamp of the last journal that was replayed converted to a local value
lastReplayJournalTimeStampUTC	TimeStamp	Timestamp of the last journal that was replayed as a UTC value
latestReadyJournalNumber	Integer	Number of the latest journal that is ready to be replayed
latestReadyJournalTimeStamp	TimeStamp	Timestamp of the latest journal that is ready to be replayed converted to a local value
latestReadyJournalTimeStampUTC	TimeStamp	Timestamp of the latest journal that is ready to be replayed as a UTC value
latestReplayedAuditTimeStamp	TimeStamp	Timestamp of the last audit record replayed by the database tracker converted to a local value
latestReplayedAuditTimeStampUTC	TimeStamp	Timestamp of the last audit record replayed by the database tracker as a UTC value
latestStableAuditTimeStamp	TimeStamp	Timestamp of the last audit record written to disk in block write mode converted to a local value
latestStableAuditTimeStampUTC	TimeStamp	Timestamp of the last audit record written to disk in block write mode as a UTC value
myHostName	String	Computer name of the secondary host
myName	String	Name of the secondary or RPS node, specified in the **MyName** parameter in the [SyncDbService] section of the JADE initialization file
nextReplayJournalNumber	Integer	Number of the next journal to replay
nextReplayJournalTimeStamp	TimeStamp	Timestamp of the next journal to replay converted to a local value
nextReplayJournalTimeStampUTC	TimeStamp	Timestamp of the next journal to replay as a UTC value
primaryHostName	String	Computer name of the primary host
primaryServerName	String	Name of the primary, specified in the **PrimaryServerName** parameter in the [SyncDbService] section of the JADE initialization file
readAccessDisabled	Boolean	Specifies whether read access is disabled (the value of the **ReadAccessDisabled** parameter in the [SyncDbService] section of the JADE initialization file)

Name	Type	Description
readAccessGranted	Boolean	Specifies whether read access has been granted
reasonTrackingStopped	Integer	Reason tracking stopped (see the second of the following tables)
reconnectInterval	Integer	Frequency (in seconds) at which a secondary database server attempts to reconnect to its primary server when a primary server is not available (the value of the **ReconnectInterval** parameter in the [SyncDbService] section of the JADE initialization file)
recoveryRequired	Boolean	Specifies whether recovery is required
reorgStatus	Integer	Reorganization status (see the third of the following tables)
rpsStorageMode	Integer	Storage mode represented by one of the **RpsStorageMode_Full** (0), **RpsStorageMode_MappedExtent** (1), or **RpsStorageMode_WorkingSet** (2) **JadeDatabaseAdmin** class constants
rpsTransitionHaltCode	Integer	RPS transition halt code (see the fourth of the following tables)
rpsWorkers	Integer	Number of RPS worker threads
state	Integer	State of the secondary or RPS node in relation to the primary (see the fifth of the following tables)
subrole	Integer	Database role (see the sixth of the following tables)
syncMode	Integer	Mode of journal synchronization, specified in the **SyncMode** parameter in the [SyncDbService] section of the JADE initialization file (see the fifth of the following tables)
tracking	Boolean	Contains **true** when tracking is active or it contains **false** if tracking is stopped for any reason
trackingDisabled	Boolean	Specifies whether database tracking (journal replay) process is disabled (the value of the **TrackingDisabled** parameter in the [SyncDbService] section of the JADE initialization file)

The values of the **connectionState** attribute are represented by one of the **SDSConnectionState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_Connected	2
SDS_Connecting	3
SDS_ConnectionFailed	4
SDS_Disconnected	1

The values of the **reasonTrackingStopped** attribute are represented by one of the **SDSStopTrackingCodes** category global constants listed in the following table.

Global Constant	Integer Value	Description
SDS_ReasonErrorHalt	8	Tracking halted due to an error condition (the error code is saved in the **SDSSecondary** or **SDSSecondaryProxy** dynamic **lastErrorCode** attribute)
SDS_ReasonRpsReorgHalt	9	Tracking stopped at transition
SDS_ReasonTakeover	7	Tracking stopped during a takeover operation

The values of the **reorgStatus** attribute are represented by one of the **SDSReorgState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_ReorgStateNotReorging	1
SDS_ReorgStateOfflinePhase	5
SDS_ReorgStateReorgingFiles	4
SDS_ReorgStateRestarting	6
SDS_ReorgStateStarting	3
SDS_ReorgStateSeekingApproval	2

The values of the **rpsTransitionHaltCode** attribute are represented by one of the **RPSTransitionHaltCode** category global constants listed in the following table.

Global Constant	Integer Value	Description
RPS_HaltAutoScript	1	An automatic initiate alter script was generated (will be automatically loaded by the data pump application if configured to automatically restart)
RPS_HaltManualScript	2	A manual alter script was generated (requires administration user intervention to apply changes to RDB before tracking can be resumed)
RPS_HaltMappingDeleted	3	The RPS mapping was deleted on the primary database, rendering the RPS node and associated RDB defunct
RPS_HaltNoScript	0	Changes do not affect RDB, so no script was generated

The values of the **state** attribute are represented by one of the **SDSSecondaryState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_StateCatchingUp	1
SDS_StateDisconnected	0
SDS_StateReorging	5
SDS_StateSynchronized	2
SDS_StateTrackingHalted	4
SDS_StateTransferHalted	3

The values of the **subrole** attribute are represented by one of the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value
SDS_SubroleNative (native JADE Object Manager database)	1
SDS_SubroleRelational (relational database)	2

The values of the **syncMode** attribute are represented by one of the **SDSSecondaryState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_BlockWrite	2
SDS_JournalSwitch	1

Note When a secondary server or RPS node is restarted in an interrupted mode, the **recoveryRequired** attribute is set and the active and interrupted transaction counts are not valid until the first journal has been replayed. The **recoveryRequired** attribute is reset when the outstanding interrupted transactions complete.

For the block write synchronization mode only, the dynamic attributes for the secondary or RPS object type are listed in the following table.

Name	Type	Description
latestReplayedAuditTimestamp	TimeStamp	Timestamp of the latest replayed audit record
latestStableAuditTimestamp	TimeStamp	Timestamp of the latest stable replayed audit record
maxCommittedTranID	Decimal	Transaction id of the maximum committed audit record
maxReplayedTranID	Decimal	Transaction id of the maximum replayed audit record
maxStableTranID	Decimal	Transaction id of the maximum stable audit record

To obtain the latest committed timestamp for a secondary or RPS node in block write synchronization mode, compare the **latestReplayedAuditTimestamp** attribute with the **latestStableAuditTimestamp** value, by a single call to the **sdsGetMyServerInfo** method from a secondary or RPS node (or the **sdsGetSecondaryInfo** method from the primary). The difference between these attribute values should be very small and under normal conditions, they will differ only by network latency plus the journal disk write time on the secondary or RPS node.

All audit timestamps are UTC values, which are converted to local time for dynamic attributes. The timestamps of both the primary and secondary or RPS proxy are both derived from the UTC audit timestamp recorded by the primary and converted to local time on the secondary or RPS proxy (catering for primary and secondary or RPS proxy time zones that differ).

sdsGetSecondaryInfo

Signature `sdsGetSecondaryInfo(name: String; attributes: JadeDynamicObject input);`

The **sdsGetSecondaryInfo** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, retrieves the attributes of the secondary system or RPS node specified in the **name** parameter, converts the **attributes** input parameter into an **SDSSecondary** dynamic object, and populates the **JadeDynamicObject** instance with the attributes and values retrieved from the secondary server or RPS node.

The caller is responsible for creation and deletion of the secondary or RPS dynamic object parameter. As it is not valid to execute this method on a secondary database or RPS node, use the **sdsGetMyServerInfo** method instead. For details about **SDSSecondary** dynamic object attributes, see the **sdsGetMyServerInfo** method.

sdsGetSecondaryProxies

Signature `sdsGetSecondaryProxies(proxies: JadeDynamicObjectArray input);`

The **sdsGetSecondaryProxies** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, creates and populates an **SDSSecondaryProxy** object for each secondary system or RPS node that is currently registered on the executing primary system and returns these secondary or RPS proxy dynamic objects in the **proxies** parameter.

The **SDSSecondaryProxy** dynamic object has a **JadeDynamicObject** class **name** attribute value of **SDSSecondaryProxy** and **type** attribute value of **SDS_SecondaryProxyType** (2).

The dynamic attributes that are returned are listed in the following table.

Name	Type	Description
connectionCheckInterval	Integer	Number of seconds at which the secondary database or RPS node polls the primary to determine reachability via the communication paths, specified in the **ConnectionPollInterval** parameter in the [SyncDbService] section of the JADE initialization file
connectionState	Integer	State of the connection to the primary
hostName	String	Computer name of the secondary or RPS proxy host on the primary
lastErrorCode	Integer	Number of the last error that occurred
myName	String	Name of the secondary or RPS proxy on the primary, specified in the **MyName** parameter in the [SyncDbService] section of the JADE initialization file
nextJournalNumber	Integer	Next journal the primary sends if the secondary or RPS node is catching up or the next write journal when the secondary or RPS node is mirroring writes from the current journal. It remains valid when the secondary or RPS node is disconnected.
primaryServerName	String	Name of the primary, specified in the **PrimaryServerName** parameter in the [SyncDbService] section of the JADE initialization file

Name	Type	Description
subrole	Integer	Database role
syncMode	Integer	Mode of journal synchronization, specified in the **SyncMode** parameter in the [SyncDbService] section of the JADE initialization file
totalSends	Interger64	Count of messages sent to the secondary
totalBlocksSent	Interger64	Count of journal blocks sent to the secondary (there can be from 1 through 16 blocks per message)
totalBytesSent	Interger64	Count of bytes sent to the secondary; that is, the total size of all messages sent
totalUncompressedBytes	Interger64	Count of bytes sent to the secondary if compression was disabled
lastRecordSentJournal	Interger64	Journal number of the last journal block sent
lastRecordSentOffset	Interger64	Byte offset of the last journal block sent

The values of the **subrole** attribute are represented by one of the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value
SDS_SubroleNative (native JADE Object Manager database)	1
SDS_SubroleRelational (relational database)	2

The values of the **syncMode** attribute are represented by one of the **SDSSecondaryState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_BlockWrite	2
SDS_JournalSwitch	1

The caller is responsible for deletion of these transient dynamic objects. Deletion is best achieved by purging the array when the entries have been processed.

sdsGetSecondaryProxy

Signature `sdsGetSecondaryProxy(name: String;`
`proxy: JadeDynamicObject input);`

The **sdsGetSecondaryProxy** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, creates and populates an **SDSSecondaryProxy** object for the secondary system or RPS node specified in the **name** parameter and returns the secondary or RPS proxy dynamic object in the **proxy** parameter.

The **SDSSecondaryProxy** object returned by the **sdsGetSecondaryProxy** method contains a subset of the **SDSSecondary** dynamic object attributes, including **nextJournalNumber**.

The secondary or RPS proxy object:

- Enables you to obtain summary information about a secondary or RPS node without having to *go* to the secondary or RPS node.
- Can be called when the secondary or RPS node is disconnected.

Note You should call this method to first get a secondary or RPS proxy object and check the **connectionState** attribute value before calling the **sdsGetSecondaryInfo** method.

Use this method in conjunction with the **JadeDatabaseAdmin** class **getCurrentJournalOffset** method to determine the amount of journal data that has not been sent to the secondary.

The **SDSSecondaryProxy** dynamic object has a **JadeDynamicObject** class **name** attribute value of **SDSSecondaryProxy** and **type** attribute value of **SDS_SecondaryProxyType** (2).

The dynamic attributes that are returned are listed in the following table.

Name	Type	Description
connectionCheckInterval	Integer	Number of seconds at which the secondary database or RPS node polls the primary to determine reachability via the communication paths, specified in the **ConnectionPollInterval** parameter in the [SyncDbService] section of the JADE initialization file
connectionState	Integer	State of the connection to the primary
hostName	String	Computer name of the secondary or RPS proxy host on the primary
lastErrorCode	Integer	Number of the last error that occurred
myName	String	Name of the secondary or RPS proxy on the primary, specified in the **MyName** parameter in the [SyncDbService] section of the JADE initialization file
nextJournalNumber	Integer	Next journal the primary sends if the secondary or RPS node is catching up or the next write journal when the secondary or RPS node is mirroring writes from the current journal. It remains valid when the secondary or RPS node is disconnected.
primaryServerName	String	Name of the primary, specified in the **PrimaryServerName** parameter in the [SyncDbService] section of the JADE initialization file
subrole	Integer	Database role
syncMode	Integer	Mode of journal synchronization, specified in the **SyncMode** parameter in the [SyncDbService] section of the JADE initialization file
totalSends	Integer64	Count of messages sent to the secondary
totalBlocksSent	Integer64	Count of journal blocks sent to the secondary (there can be from 1 through 16 blocks per message)
totalBytesSent	Integer64	Count of bytes sent to the secondary; that is, the total size of all messages sent
totalUncompressedBytes	Integer64	Count of bytes sent to the secondary if compression was disabled
lastRecordSentJournal	Integer64	Journal number of the last journal record sent
lastRecordSentOffset	Integer64	Byte offset of the last journal record sent

The values of the **subrole** attribute are represented by one of the **SDSDatabaseRoles** category global constants listed in the following table.

Global Constant	Integer Value	Description
SDS_SubroleNative	1	Native JADE Object Manager database
SDS_SubroleRelational	2	Relational database

The values of the **syncMode** attribute are represented by one of the **SDSSecondaryState** category global constants listed in the following table.

Global Constant	Integer Value
SDS_BlockWrite	2
SDS_JournalSwitch	1

The caller is responsible for deletion of these transient dynamic objects. Deletion is best achieved by purging the object when the entry has been processed.

sdsGetTransactions

Signature `sdsGetTransactions(transactions: JadeDynamicObjectArray input);`

The **sdsGetTransactions** method of the **JadeDatabaseAdmin** class, valid only at secondary database systems, creates and populates an **SDSTransaction** dynamic object for each active transaction that is currently being replayed or isolated on the executing secondary system and returns these transaction dynamic objects in the transactions input array.

The **SDSTransaction** instances returned in the transactions array have a **name** attribute value of **SDSTransaction** and **type** attribute value of **SDS_TransactionType** (4).

The transaction dynamic attributes are listed in the following table.

Name	Type	Description
startTime	TimeStamp	Time at which the transaction started
status	Integer	Transaction status (see the following table)
statusText	String	Descriptive text that is displayed for the transaction status
transactionID	Decimal	Identifier of the transaction
userName	String	Name of the user

The values of the transaction status attribute represented by one of the **SDSTransactionStates** category global constants are listed in the following table.

Global Constant	Integer Value
SDS_TranNormal	1
SDS_TranInterrupted	2
SDS_TranDeferred	3
SDS_TranWaitingAuditCommit	4
SDS_TranReadyToCommit	5
SDS_TranPrepareToCommit	6
SDS_TranReadyToAbort	7
SDS_TranInDoubt	8

The only value of interest to most users is **SDS_TranInterrupted**. When there are active transactions in an interrupted state, read-access to persistent objects is not permitted. See also the **sdsEnableReadAccess** and **sdsEnableReadAccessAt** methods, earlier in this chapter.

The caller is responsible for deletion of these transient dynamic objects. Deletion is best achieved by purging the array when the entries have been processed.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsGetTransactionsAt

Signature
```
sdsGetTransactionsAt(secondaryName: String;
                     transactions:  JadeDynamicObjectArray input);
```

The **sdsGetTransactionsAt** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, creates and populates an **SDSTransaction** dynamic object for each active transaction that is currently being replayed or isolated on the secondary database system specified in the **secondaryName** parameter and returns these transaction dynamic objects in the transactions input array. For details about the transaction object values, see the **sdsGetTransactions** method.

The caller is responsible for deletion of these transient dynamic objects. Deletion is best achieved by purging the array when the entries have been processed.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

sdsInitiateHostileTakeover

Signature
```
sdsInitiateHostileTakeover();
```

The **sdsInitiateHostileTakeover** method of the **JadeDatabaseAdmin** class initiates a hostile take-over of the primary database by the executing secondary system without involving the primary system. This method should be invoked only within an SDS secondary system. Invoking this method from within a primary system raises an exception.

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node unless the node is running in **Full** database replication database mode.

sdsInitiateTakeover

Signature
```
sdsInitiateTakeover(takeoverMode:      Integer;
                    nextPrimaryServer: String);
```

The **sdsInitiateTakeover** method of the **JadeDatabaseAdmin** class initiates a negotiated take-over of the primary database by the secondary server specified in the **nextPrimaryServer** parameter.

The **takeoverMode** parameter specifies how the take-over operation will deal with potential conflicts between executing reader processes and database state that is being isolated, which will become visible when the secondary server assumes the role of a primary database. The values for the take-over mode are represented by one of the **SDSTakeoverState** category global constants listed in the following table.

Global Constant	Integer Value	Take-over is …
SDS_TakeoverConditional	1	Conditional on there being no conflicts
SDS_TakeoverForced	2	Forced if conflicts exist

Note A runtime exception is raised if this method is called for a Relational Population Service (RPS) node.

This method should be invoked only within an SDS primary system. Invoking this method from within a secondary system raises an exception.

For more details about take-over operations, see "SDS Takeover Operations", in Chapter 1 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

sdsIsInitialized

Signature `sdsIsInitialized(): Boolean;`

The **sdsIsInitialized** method of the **JadeDatabaseAdmin** class, valid at the primary and secondary systems or RPS nodes, returns **true** if the Synchronized Database Service (SDS) environment is initialized for the JADE system.

If SDS is not initialized or the JADE system is not SDS-capable, this method returns **false**.

sdsIsRunning

Signature `sdsIsRunning(): Boolean;`

The **sdsIsRunning** method of the **JadeDatabaseAdmin** class, valid at the primary and secondary systems or RPS nodes, returns **true** if SDS is running (that is, initialized and active) on the JADE system.

On a primary database, you can stop the SDS service by calling the **sdsStopService** method at the primary database server (or by using the SDS Administration application **Stop Service** command from the Primary menu, documented under "Stopping a Synchronized Database Service", in Chapter 1 of the *JADE Synchronized Database Service (SDS) Administration Guide*).

When the SDS service is stopped, the **sdsIsRunning** method returns **false**.

Note Although the **sdsIsInitialized** method returns **false** if the system is not SDS-capable, the **sdsIsRunning** method also takes that into account, so you do not need to call both the **sdsIsInitialized** and **sdsIsRunning** methods.

sdsReconnectNow

Signature `sdsReconnectNow();`

The **sdsReconnectNow** method of the **JadeDatabaseAdmin** class causes the secondary server or RPS node invoking the method to attempt to reconnect to its configured primary server rather than waiting for the time specified in the **ReconnectInterval** parameter in the [SyncDbService] section of the JADE initialization file to expire.

You can use this method to attempt to establish a connection to the primary system that was disabled by a **sdsDisablePrimaryConnection** or **sdsDisablePrimaryConnectionAt** method call; for example, re-establishing a disrupted network link caused by a problem in the network path.

sdsReplayNextJournal

Signature `sdsReplayNextJournal();`

The **sdsReplayNextJournal** method of the **JadeDatabaseAdmin** class, valid only at the secondary system or RPS node, initiates a replay of the next ready journal on a secondary database or RPS node when journal replay is disabled.

If you want to initiate the replaying of the next ready journal on the secondary database server or RPS node when tracking is enabled (for example, when tracking was halted due to a missing journal), use the **sdsResume** method.

sdsReplayNextJournalAt

Signature `sdsReplayNextJournalAt(secondaryName: String);`

The **sdsReplayNextJournalAt** method of the **JadeDatabaseAdmin** class, valid only at the primary system, initiates a replay of the next ready journal on the secondary database or RPS node specified in the **secondaryName** parameter when journal replay is disabled.

If you want to initiate the replaying of the next ready journal on the secondary database server or RPS node when tracking is enabled (for example, when tracking was halted due to a missing journal), use the **sdsResumeAt** method.

sdsResume

Signature `sdsResume();`

The **sdsResume** method of the **JadeDatabaseAdmin** class, valid only at secondary systems or RPS nodes, resumes the automatic replay and shipping of journals to a secondary system or RPS node when shipping and replay have been temporarily halted due to a missing journal.

When journal replay is resumed, the following actions are performed on the secondary server or RPS node.

1. Scans its current journal directory for new arrivals and updates the latest ready journal information.
2. Starts replaying from the next replay journal if it is resident on the secondary server or RPS node.
3. Requests the next required journal from the primary database server, if connected.

If you want to initiate the replaying of the next ready journal when tracking is disabled, use the **sdsReplayNextJournal** method.

sdsResumeAt

Signature `sdsResumeAt(secondaryName: String);`

The **sdsResumeAt** method of the **JadeDatabaseAdmin** class, valid only at the primary system, sends a resume command to the secondary database or RPS node specified in the **secondaryName** parameter to resume the automatic replay and shipping of journals to that secondary system or RPS node when shipping and replay have been temporarily halted due to a missing journal.

When journal replay is resumed, the following actions are performed on the secondary serve or RPS node.

1. Scans its current journal directory for new arrivals and updates the latest ready journal information.
2. Starts replaying from the next replay journal if it is resident on the secondary server or RPS node.

3. Requests the next required journal from the primary database server, if connected.

If you want to initiate the replaying of the next ready journal on the secondary database server or RPS node when tracking is disabled, use the **sdsReplayNextJournalAt** method.

sdsStartService

Signature `sdsStartService();`

The **sdsStartService** method of the **JadeDatabaseAdmin** class, valid only at the primary database server, starts a synchronized database service on the JADE system if it is currently stopped. If the service has already started, this request is ignored.

sdsStartTracking

Signature `sdsStartTracking();`

The **sdsStartTracking** method of the **JadeDatabaseAdmin** class, valid only at secondary systems and RPS nodes, resumes the tracking process on the secondary server or RPS node on which this method is executed when tracking is inactive (that is, disabled). For details, see "Delayed Replay" under "Synchronized Database Service Functionality", in Chapter 1 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

sdsStartTrackingAt

Signature `sdsStartTrackingAt(secondaryName: String);`

The **sdsStartTrackingAt** method of the **JadeDatabaseAdmin** class, valid only at the primary system, resumes the tracking process on the secondary server or RPS node specified in the **secondaryName** parameter when tracking is inactive (that is, disabled).

sdsStopService

Signature `sdsStopService();`

The **sdsStopService** method of the **JadeDatabaseAdmin** class, valid only at the primary database server, closes down the SDS environment on the database server that executes this method, while allowing all other database operations to continue. As this method is not valid on secondary database servers and RPS nodes, use the **sdsDisablePrimaryConnection** method to take a secondary server offline.

sdsStopTracking

Signature `sdsStopTracking();`

The **sdsStopTracking** method of the **JadeDatabaseAdmin** class, valid only at secondary database servers and RPS nodes, requests termination of the tracking process on the current secondary server or RPS node that calls this method.

If the tracker process is currently replaying a journal, it continues replaying until it reaches the end of the current replay journal.

sdsStopTrackingAt

Signature `sdsStopTrackingAt(secondaryName: String);`

The **sdsStopTrackingAt** method of the **JadeDatabaseAdmin** class, valid only at the primary database system, requests the termination of the tracking process on the secondary system or RPS node specified in the **secondaryName** parameter.

If the tracker process is currently replaying a journal, it continues replaying until it reaches the end of the current replay journal.

verifyJournal

Signature
```
verifyJournal(number:    Integer;
              sourceDir: String): Integer;
```

The **verifyJournal** method of the **JadeDatabaseAdmin** class verifies the transaction journal specified in the **number** parameter and located in the directory specified in the **sourceDir** parameter. If the value of the **sourceDir** parameter is **null**, the current transaction journal directory is used.

Note The **verifyJournal** method is executed on the database server even if the method initiating it is executed from a client node, so the **sourceDir** parameter must correctly identify the directory on the server.

This method returns the number of errors that were detected when verifying the transaction journal. Details of the verify operation are recorded in the ***journal-file-name*.scan** file, located in the default database directory (for example, **d:\jade\logs\current\db0000076894.scan**).

JadeDatabaseAdmin Class Event Notifications

A number of automatic events are notified, to allow JADE applications to monitor and report on the operation and progress of file backups.

The **FileBackupStartEvent** and **FileBackupCompleteEvent** events are caused by a **JadeDatabaseAdmin** object notifying the start and completion of each file backup when using a multiple file backup method (for example, **backupAllDbFiles** or **backupDbFiles**). In addition, if operation and progress event notifications are enabled (by using the **enableProgressEvents** method), a progress event is notified by each **DbFile** instance whenever the nominated percentage increment of the file has been copied and an operation event notifies which backup operation is being performed on the file. (For details, see "DbFile Class Event Notifications", earlier in this chapter.)

The event types are enumerated by the **JadeDatabaseAdmin** class constants listed in the following table.

Constant	Integer Value	Description
BackupAbortedEvent	4000	Multiple file backup terminated by the user
BackupCancelledEvent	8000	Multiple file backup has been canceled by the user
BackupCompleteEvent	3000	Multiple file backup has completed normally
BackupFailedEvent	9000	Multiple file backup has failed
FileBackupStartEvent	1000	File backup has commenced
FileBackupCompleteEvent	2000	File backup has finished
JournalTransferEvent	6000	Recovery journal file has been transferred

The **BackupAbortedEvent** event is caused by a **JadeDatabaseAdmin** class instance when a multiple file backup operation is terminated abnormally due to a user-requested termination. The **userInfo** parameter of your notification callback for this event is **null**.

The **BackupFailedEvent** event is caused by a **JadeDatabaseAdmin** class instance when a multiple file backup operation is terminated abnormally due to a fatal exception. The **userInfo** parameter of your notification callback for this event is **null**.

The **BackupCompleteEvent** event is caused by a **JadeDatabaseAdmin** class instance when a multiple file backup has completed; that is, all files have been successfully backed up. The **userInfo** parameter of your user notification method for this event is **null**.

The **FileBackupCompleteEvent** event is caused by a **JadeDatabaseAdmin** class instance when using one of the multiple file backup methods (for example, **backupAllDbFiles** or **backupDbFiles**) as each file backup is completed. The **userInfo** parameter of your user notification method contains a **DbFile** reference that represents the file for which a backup has commenced.

The **FileBackupStartEvent** event is caused by a **JadeDatabaseAdmin** class instance when using one of the multiple file backup methods (for example, **backupAllDbFiles** or **backupDbFiles**) as each file backup is commenced. The **userInfo** parameter of your user notification method contains a **DbFile** reference that represents the file for which a backup has commenced.

The **JournalTransferEvent** event is caused by the singleton persistent instance of the **System** class (denoted by the **system** JADE system variable) when a transaction journal is transferred. The **userInfo** parameter of your user notification method contains the number of the transaction journal that is transferred.

This event is intended to notify the backup application when an active journal becomes offline so that it can be backed up. The notification of this event is automatically enabled when the first instance of the **JadeDatabaseAdmin** class is created and it is disabled when the last instance is deleted.

Subscribing to JadeDatabaseAdmin Events

The following examples show the **Object** class **beginNotification** method signature used to subscribe to **JadeDatabaseAdmin** events.

```
beginNotification(self.dba, JadeDatabaseAdmin.BackupAbortedEvent,
                  Response_Continuous, tag);

beginNotification(self.dba, JadeDatabaseAdmin.BackupCompleteEvent,
                  Response_Continuous, tag);

beginNotification(self.dba, JadeDatabaseAdmin.FileBackupCompleteEvent,
                  Response_Continuous, tag);

beginNotification(self.dba, JadeDatabaseAdmin.FileBackupStartEvent,
                  Response_Continuous, tag);
```

The following example shows the **Object** class **beginNotification** method signature used to subscribe to log transfer events.

```
beginNotification(system, JadeDatabaseAdmin.JournalTransferEvent,
                  Response_Continuous, tag);
```

You could use a user notification method signature of the following specific form for objects that are interested only in backup start or complete event notifications.

```
user-notification-method(eventType: Integer; obj: JadeDatabaseAdmin;
                         eventTag: Integer; file: DbFile) updating;
```

In this specific form, the caused-by parameter is of type **JadeDatabaseAdmin** and the **userinfo** parameter is named **file** and is of type **DbFile**.

You could use a user notification method signature of the following specific form for objects that are interested only in journal transfer event notifications.

```
user-notification-method(eventType: Integer; obj: System;
                         eventTag: Integer; journalNumber: Integer) updating;
```

In this specific form, the caused-by parameter is of type **JadeDatabaseAdmin** and the **userinfo** parameter is named **journalNumber**.

Objects that need to handle several notification types must use the more generic user notification method signature, as follows.

```
user-notification-method(eventType: Integer; obj: theObject;
                         eventTag: Integer; userInfo: Any) updating;
```

In this generic form, the caused-by parameter is of type **Object** and the **userinfo** parameter is of type **Any**.

In the notation in these callback signatures, *user-notification-method* is **userNotification** for non-form objects or **userNotify** for notifications registered by form objects.

JadeDbFilePartition Class

The **JadeDbFilePartition** class is the transient class that provides an administrative Application Programming Interface (API) for manipulating and querying the state of database partitions.

Use the **DbFile** class to partition database files and to iterate partitions.

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

Inherits From: Object

Inherited By: (None)

JadeDbFilePartition Properties

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

Property	Contains…
dbFile	A reference to the parent **DbFile** instance
partitionID	The partition identifier assigned to the partition

dbFile

Type: DbFile

The read-only **dbFile** property of the **JadeDbFilePartition** class contains a reference to the parent **DbFile** instance.

partitionID

Type: Integer64

The read-only **partitionID** property of the **JadeDbFilePartition** class contains the partition identifier assigned to the partition.

The value uniquely identifies a partition within the set of partitions that make up a partitioned file.

JadeDbFilePartition Methods

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

Method	Description
allInstances	Populates an object array with references to objects stored in a database partition
backupFilePartition	Backs up a single partition of a physical database file
certifyFile	Initiates the certification of a single database partition
disableAuditing	Disables auditing associated with object operations performed for the partition
display	Returns details about a single database partition
drop	Removes the partition and marks it as deleted

Method	Description
enableAuditing	Re-enables the auditing associated with object operations
freeze	Converts a partition to read-only mode, after which no object update, delete, or create is permitted
getBackupTimestamp	Returns a timestamp containing the date and time the database partition was last backed up
getCreationTimestamp	Returns a timestamp containing the date and time the database partition was created
getFileLength	Returns the size of a single physical database partition
getFileStatus	Returns the status of a single physical database partition during the backup process
getFreeSpace	Evaluates the available free space in a single database partition
getFullBackupTimestamp	Returns a timestamp containing the date and time the database file was last backed up
getLabel	Returns the logical label associated with the database partition
getLocation	Returns the file system path assigned to the database partition
getModifiedTimestamp	Returns a timestamp containing the date and time the database partition was last updated
getName	Returns the name of the database partition
getStatistics	Returns statistics on reads of single database partition activity
getTotalFileLength64	Returns the total bytes occupied by the database partition file
isAuditing	Returns **true** if auditing associated with object operations is enabled
isFrozen	Returns **true** if the associated partition is frozen
isOffline	Returns **true** if the associated partition is offline
markOffline	Marks a partition as officially absent so that it can be taken offline
markOnline	Marks a partition as present after it has been brought back online
move	Changes the location attribute and moves the partition to the specified destination
setLabel	Changes the logical label associated with the database partition
setLocation	Changes the default or designated physical location of a database partition
thaw	Restores the database partition to its default active state

allInstances

Signature
```
allInstances(objArray:     ObjectArray input;
             maxInstances: Integer64);
```

The **allInstances** method of the **JadeDbFilePartition** class populates the object array specified in the **objArray** parameter with references to objects stored in the partition associated with the receiver. The object array is not cleared before instances are added.

The **maxInstances** parameter specifies the maximum number of instances that will be added to the object array.

Note The **allInstances** method can used to obtain a list of instances stored in an offline partition without bringing it online.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

backupFilePartition

Signature

```
backupFilePartition(backupDir:       String;
                    verifyChecksums: Boolean;
                    compress:        Boolean;
                    overwriteDest:   Boolean);
```

The **backupFilePartition** method of the **JadeDbFilePartition** class initiates a backup of a single partition of a database **backupDir** parameter. (The backup directory is specified relative to the database server node and must be a valid directory.) This method executes on the database server node, and is implemented and executed by the database engine.

The backup process performs various consistency checks similar to a database certify, to ensure the integrity of the backup.

Set the **verifyChecksums** parameter to **true** if you want checksums verified in the backed up file partition. Checksum verification is performed in a separate pass of the backed up file partition immediately after the copy phase. A checksum analysis of your backed up database partition verifies that the file partition has not been corrupted by a hardware or environmental problem during the backup process. You should perform a separate checksum analysis of any backup that has been moved across media, especially if transferred across a network.

Set the **compress** parameter to **true** if you want to compress backed up data. You can compress data in a checked or an unchecked backup.

Set the **overwriteDest** parameter to **true** if you want to allow file partition backups to overwrite existing files in the destination backup directory. When this parameter is **false**, an exception is raised if an existing file partition is detected.

Note Before partitions of a database file can be backed up using the **backupFilePartition** method, call the **beginPartitionedFileBackup** method for the corresponding database file instance.

Similarly, call the **endPartitionedFileBackup** method when the required partitions of a database file have been backed up.

Separate JADE processes can initiate concurrent file partition backups. This allows multiple file partitions to be copied concurrently, which can reduce elapsed backup time when the source and destination volumes are on different physical devices.

Caution Because of increased disk contention and disk head movement, concurrent backup operations run slower if the backup is sent to a single disk drive.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

You can use the **JadeDatabaseAdmin** class **enableProgressEvents** method to optionally notify operation and progress notifications for file partition backups. You must both enable and subscribe to this event if you want file partition backup operation and progress notification. For more details, see "DbFile Class Event Notifications".

certifyFile

Signature `certifyFile(): Integer;`

The **certifyFile** method of the **JadeDbFilePartition** class initiates an online certification of a single physical database partition; that is, it checks the database integrity. The parent map file must first be converted to read-only.

This method returns the number of errors that were detected when certifying the database partition.

The **certifyFile** method executes on a persistent server node, and is implemented and executed by the physical database engine. For details, see "Using the Certify Files Command", in Chapter 1 of the *JADE Database Administration Guide*.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

disableAuditing

Signature `disableAuditing(maxWaitForQuietpoint: Integer);`

The **disableAuditing** method of the **JadeDbFilePartition** class disables the auditing associated with object operations performed against the partition.

Auditing is disabled within a quiet point after a checkpoint has successfully completed to move the database recovery point. If the **maxWaitForQuietpoint** parameter has a non-zero value, it specifies the maximum time in seconds that the operation will wait for there to be no transaction activity, and overrides the configured or default database value specified by the **MaxWaitForQuietPoint** parameter in the [PersistentDb] section of the JADE initialization file.

If a quiet point cannot be established, exception 3077 (*Maximum time to wait for quiet point was exceeded*) is raised.

display

Signature `display(): String;`

The **display** method of the **JadeDbFilePartition** class returns a string containing details about a single database partition (that is, the name, location, whether the partition is frozen and whether it is marked as absent, and so on).

The following code fragment shows the use and of the **display** method to display information about the first partition of the **Order** file.

```
write Order.getDbFile.getPartition(1).display;
```

The following output is produced.

```
---JadeDbFilePartition/16766.1---
dbFile = DbFile/1306.4 : 4
partitionID = 1
name = part0000000001
label =
location =
offline = false
frozen = false
created @ 26 February 2009, 13:55:08
modified @ 26 February 2009, 14:20:11
```

```
stable in backup @ 00:00:00
stable in full backup @ 00:00:00
```

drop

Signature `drop();`

The **drop** method of the **JadeDbFilePartition** class removes the partition and marks it as deleted.

enableAuditing

Signature `enableAuditing(options: Integer);`

The **enableAuditing** method of the **JadeDbFilePartition** class re-enables the auditing associated with object operations performed against the file.

File operations are blocked and the partition is made stable. A copy of the partition is then inserted into the audit stream.

By default, the partition is compressed before being written into the journal. If you know that the data in the partition compresses poorly, you can disable compression by specifying the value **EnableAudit_NoCompress** a **DbFile** class constant) for the **options** parameter.

File compression and decompression operations use the directory specified by the **ReorgWorkDirectory** parameter in the [JadeReorg] section of the JADE initialization file.

freeze

Signature `freeze() updating;`

The **freeze** method of the class converts a database partition to read-only mode after which all object update, delete, or create operations are not permitted. (See also the **thaw** method.)

Note All objects in a frozen partition are automatically frozen, overriding individual volatility state.

An exception is raised if the database partition was not located, there was an error accessing a database partition control file, an attempt was made to access an offline partition, the partition is locked for administrative purposes, the partition is required for object creation, the database is locked for reorganization, or you are attempting the freeze operation when the database is in backup state.

getBackupTimestamp

Signature `getBackupTimestamp(): TimeStamp;`

The **getBackupTimestamp** method of the **JadeDbFilePartition** class returns a timestamp containing the date and time the file partition was *stable* in a backup where *stable* means the partition was not updated during the backup and therefore does not require recovery.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getCreationTimestamp

Signature `getCreationTimestamp(): TimeStamp;`

The **getCreationTimestamp** method of the **JadeDbFilePartition** class returns a timestamp containing the date and time the database partition was created.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getFileLength

Signature `getFileLength(): Integer64;`

The **getFileLength** method of the **JadeDbFilePartition** class returns the size of a single physical database partition in bytes as an **Integer64** value. This method executes on a persistent server node, and is implemented and executed by the physical database engine.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getFileStatus

Signature `getFileStatus(): Integer;`

The **getFileStatus** method of the **JadeDbFilePartition** class returns the status of physical database partition. This method executes on a persistent server node, and is implemented and executed by the physical database engine.

The status of the database partition is represented by **DbFile** class constants listed in the following table.

Constant	Description	Integer Value
Status_DelEncrypted	Encrypted file deleted from the control file	9
Status_Deleted	Partition deleted from control file	6
Status_InvalidPath	Invalid database partition path in control file	7
Status_Missing	Partition exists but the file is missing	3
Status_NotAssigned	Partition not defined in control file	1
Status_NotCreated	Partition deleted or not yet created	2
Status_Offline	Partition or file is offline	8
Status_Resident	Partition is resident on disk	4
Status_Unmapped	Partition is not mapped	5

You can use this method in backup applications to determine the status of database partitions prior to commencing the backup.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getFreeSpace

Signature `getFreeSpace(freeSpace: Integer64 output): Integer;`

The **getFreeSpace** method of the **JadeDbFilePartition** class evaluates the total amount of free space in a single physical database partition and returns the amount as an **Integer64** value. This method returns the number of errors encountered, if any, while performing the evaluation operation.

You can initiate the free space evaluation operation while the database is open with **update** usage; however, if the database mode is not **exclusive** and is not **archive**, the file access mode must be **read-only**.

Restrictions for evaluating free space are those that apply when compacting a database partition. For details, see "Compacting Files", in Chapter 3 of the *JADE Database Administration Guide*.

This method executes on a persistent server node, and is implemented and executed by the physical database engine. For details, see "Evaluating Free Space", in Chapter 3 of the *JADE Database Administration Guide*.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getFullBackupTimestamp

Signature `getFullBackupTimestamp(): TimeStamp;`

The **getFullBackupTimestamp** method of the **JadeDbFilePartition** class returns a timestamp containing the date and time the file partition was *stable* in a full backup where *stable* means the partition was not updated during the backup and therefore does not require recovery.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getLabel

Signature `getLabel(): String;`

The **getLabel** method of the **JadeDbFilePartition** class returns a string containing the label associated with the database partition.

When a partition is first created, the label is blank. You can change this label to something meaningful for the application. For details about setting the label, see the **setLabel** method.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getLocation

Signature `getLocation(): String;`

The **getLocation** method of the **JadeDbFilePartition** class returns a string containing the file system path assigned to the database partition. For details about setting the location, see the **setLocation** method.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getModifiedTimestamp

Signature `getModifiedTimestamp(): TimeStamp;`

The **getModifiedTimestamp** method of the **JadeDbFilePartition** class returns a timestamp containing the date and time the database partition was last updated.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

getName

Signature `getName(): String;`

The **getName** method of the **JadeDbFilePartition** class returns the logical name assigned to the database partition.

When a partition is first created, it is assigned a name in the following format.

```
part<partition-ID>
```

The name cannot be changed.

getStatistics

Signature `getStatistics(jdo: JadeDynamicObject input);`

The **getStatistics** method of the **JadeDbFilePartition** class returns statistics relating to read and write operations on the persistent database partition represented by the **JadeDbFilePartition** instance used as 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 properties returned in the **JadeDynamicObject** are listed in the following table.

Property	Description
logicalReads	The total number of read requests
logicalWrites	The total number of write requests
logicalReadBytes	The total accumulated size for all read requests
logicalWriteBytes	The total accumulated size for all write requests
physicalReads	The actual number of file partition read operations
physicalWrites	The actual number of file partition write operations
physicalReadBytes	The actual accumulated size for all file partition read operations
physicalWriteBytes	The actual accumulated size for all file partition write operations

The logical counts record the number and size of requests that can be serviced in cache, whereas the physical counts record actual disk activity.

The returned values include cumulative counters, which are not reset during the lifetime of the database server node. You need to compare values from one execution of the **getStatistics** method with the 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 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 already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with appropriate properties. The method is most efficient when the properties match those to be returned.

The following example shows the use of the **getStatistics** method.

```
showAllPartitionStats();
//display file partition statistics for all user files vars
   dbfile : DbFile;
   dbfiles : DbFileArray;
   dbpart : JadeDbFilePartition;
   dbpartitions : JadeDbFilePartitionArray;
   dba : JadeDatabaseAdmin;
   jdo : JadeDynamicObject;
begin
   create dba transient;
   create dbfiles transient;
   dba.getDbFiles(DbFile.Kind_User_Data, dbfiles);
   create dbpartitions transient;
   create jdo transient;
   foreach dbfile in dbfiles do
      if dbfile.isPartitioned then
         dbpartitions.clear;
         dbfile.getPartitions(dbpartitions,0);
         foreach dbpart in dbpartitions do
            dbpart.getStatistics(jdo);
            write dbfile.name & ":" & dbpart.getName & ":" & jdo.display;
         endforeach;
      endif;
   endforeach;
epilog
   delete dbfiles;
   delete dbpartitions;
   delete dba;
   delete jdo;
end;
```

The output from the **getStatistics** method shown in the previous example is as follows.

```
order:part0000000001:---DatabaseFileStatistics(108)---
logicalReads = 0
logicalWrites = 0
logicalReadBytes = 0
logicalWriteBytes = 0
gettotalfilelength64jadedbfilepartitition
```

getTotalFileLength64

Signature `getTotalFileLength64(selector: Integer): Integer64;`

The **getTotalFileLength64** method of the **JadeDbFilePartition** class returns the total bytes occupied by a database map file partition, including Unstructured Data Resource (UDR) files.

The value of the **selector** parameter is a bitmask that defines which subfile types to include in the bytes total. One or more of the first four of the following **DbFile** class constants values can be added together to give a subtotal.

DbFile Class Constant	Integer Value	Partition
GetTotLen_Base	1	*X*_part*NNNNNNNNNN*.dat
GetTotLen_Partitions	2	0
GetTotLen_SharedFileUDRs	4	*X*_part*NNNNNNNNNN*_udr.dat
GetTotLen_SingleFileUDRs (8)	8	Sum(*X*_part*NNNNNNNNNN*_udr[<oid>].dat)
GetTotLen_Everything (255)	255	Sum of all subfiles

The *X* value represents the map file name and the *NNNNNNNNNN* value represents a partition number.

isAuditing

Signature `isAuditing(): Boolean;`

The **isAuditing** method of the **JadeDbFilePartition** class returns **true** if auditing associated with object operations performed against the partition is enabled; otherwise it returns **false**.

isFrozen

Signature `isFrozen(): Boolean;`

The **isFrozen** method of the **JadeDbFilePartition** class returns **true** if the associated database partition has been frozen; otherwise it returns **false**. (See also the **freeze** and **thaw** methods.)

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

isOffline

Signature `isOffline(): Boolean;`

The **isOffline** method of the **JadeDbFilePartition** class returns **true** if the associated database partition has been marked offline; otherwise it returns **false**. (See also the **markOffline** and **markOnline** methods.)

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

markOffline

Signature `markOffline() updating;`

The **markOffline** method of the **JadeDbFilePartition** class marks a database partition as officially absent so that it can be taken offline. This enables you to distinguish attempted accesses to objects in the partition from accesses to objects in a missing partition.

An offline partition must be frozen and is not available for restart recovery or transaction abort. The action of marking a partition offline achieves a database quiet point (no active transactions), executes a checkpoint to establish a new restart recovery point, and if necessary, freezes the partition. Transaction processing is re-enabled once the mark offline operation has completed or aborted.

You cannot mark a partition offline that is required for object creation (that is, any partition in the creation window).

An exception is raised if the database file or partition is not created, the partition is required for object creation, the database file is locked for reorganization, or you are attempting the operation when the database is in backup state.

markOnline

Signature `markOnline() updating;`

The **markOnline** method of the **JadeDbFilePartition** class marks a database partition as present after it has been brought back online.

Notes Marking a partition *online* (by using the JADE Database Administration utility (**jdbadmin**) or by using the **markOnline** method does not open the partition.

Operator actions may be required to ensure that partitions stored on archival media are actually online.

An exception is raised if there was an error accessing a database partition control file, the database file is locked for reorganization, or the database file being opened is required but was not found.

move

Signature `move(destination: String) updating;`

The **move** method of the **JadeDbFilePartition** class changes the location attribute and moves the database partition using intrinsic database backup routines to the location specified by the **destination** parameter. The database path is relative to the database server.

If the destination of the move is on the same physical device as the source, the move is accomplished via a file system move or rename action. In this scenario, access to the file is blocked for the duration of the rename and metadata updates.

If the destination is on a different physical device, the move is accomplished using a backup copy and verify, followed by a remove of the source file. When a non-frozen partition is moved to a different physical device, the move partition operation blocks new transactions, acquires a database quiet point, and freezes the partition for the duration of the operation; transactions are unblocked after the partition has transitioned to a frozen state. The quiet point is required to ensure there are no updates to the partition in the pipeline that would require undo should a transaction abort.

You cannot move a partition required for object creation (that is, any partition in the creation window).

An exception is raised if there was an error accessing a database partition control file, the database file was not located, the database file is locked for reorganization or administrative purposes, you are attempting the operation on an offline partition or a partition that is required for object creation, or you are attempting the operation when the database is in backup state.

setLabel

Signature `setLabel(location: String) updating;`

The **setLabel** method of the **JadeDbFilePartition** class changes the logical label associated with the database partition. This will not affect the external file name and can be changed at any time.

An exception is raised if the database partition was not located or there was an error accessing a database partition control file.

setLocation

Signature `setLocation(location: String) updating;`

The **setLocation** method of the **JadeDbFilePartition** class changes the default or designated physical location of a database partition in its control record.

The **location** parameter specifies the file system directory in which the offline partition will be located when it is marked online. The value of the location attribute is stored in the partition control file and is similar in function to the path attribute for map files, which is stored in the global database control file.

This method causes the database to open the file in that location and requires that someone or something has put the file in that location before it is marked online. A typical usage for this would be to allow offline partitions that have been burned to optical media to be mounted in a different file system directory (for example, a different drive). In order to bring such a partition online, a tool or application would call the **setLocation** method to specify the location and then call the **markOnline** method, to enable access to the partition.

Note Calling the **setLocation** method alters the location attribute without moving the partition file to the destination.

An exception is raised if there was an error accessing a database partition control file, the database file was not located, you are attempting the operation on an online partition or when the database is in backup state.

thaw

Signature `thaw() updating;`

The **thaw** method of the **JadeDbFilePartition** class restores the database partition to its default active state, bringing the volatility of individual objects back into effect, allowing non-frozen objects to be updated or deleted. (See also the **freeze** method.)

An exception is raised if there was an error accessing a database partition control file, the database file was not located, you are attempting the operation on an offline partition or when the database is in backup state, or the database file is locked for reorganization.

JadeDotNetInvokeException Class

The **JadeDotNetInvokeException** class is the transient class that defines behavior for exceptions that are raised when an exception is detected in a .NET component during access of a property or the calling of a method.

A .NET exception is raised only if an internal exception occurs during a property access or method call of the .NET object. All other .NET errors are reported as user interface exceptions.

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

Inherits From: UserInterfaceException

Inherited By: (None)

JadeDotNetInvokeException Properties

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

Property	Description
dotNetExceptionObject	.NET exception object that was created by this .NET exception
helpLink	Link to the help file associated with this .NET exception
innerException	.NET exception that caused this JADE exception
message	Message that describes the .NET exception
source	.NET name of application or object that caused the .NET exception
target	.NET method that threw the current exception

dotNetExceptionObject

Type: JadeDotNetType

The **dotNetExceptionObject** property of the **JadeDotNetInvokeException** class is populated with the .NET **Exception** object when an exception of type **JadeDotNetInvokeException** is generated and the class of the .NET exception object class was imported from the .NET assembly.

helpLink

Type: String

The **helpLink** property of the **JadeDotNetInvokeException** class contains the link to the help file associated with this .NET exception.

innerException

Type: String

The **innerException** property of the **JadeDotNetInvokeException** class contains information about the .NET error that caused the JADE exception to be raised.

message

Type: String

The **message** property of the **JadeDotNetInvokeException** class contains the message that describes the .NET error that caused the JADE exception to be raised.

source

Type: String

The **source** property of the **JadeDotNetInvokeException** class contains the name of the .NET name of the object or application that caused the .NET exception.

target

Type: String

The **target** property of the **JadeDotNetInvokeException** class is the name of the .NET method that threw the current exception.

JadeDotNetType Class

The transient **JadeDotNetType** class is the superclass for classes that are proxies for non-GUI .NET classes. Before you can access the .NET object you must first create an instance of the JADE proxy class.

The JADE proxy object is used to create the corresponding .NET object by calling the **createDotNetObject** method, which invokes the default .NET constructor, or an equivalent **createDotNetObject_*n*** method, which invokes an alternative constructor with parameters. You can then use methods on the JADE proxy object, which were generated by the import wizard, to access corresponding members on the actual .NET object.

Before creating the .NET object, you can set the **usePresentationClient** property to specify whether the component runs on the presentation client or the application server. By default, all components run on the presentation client.

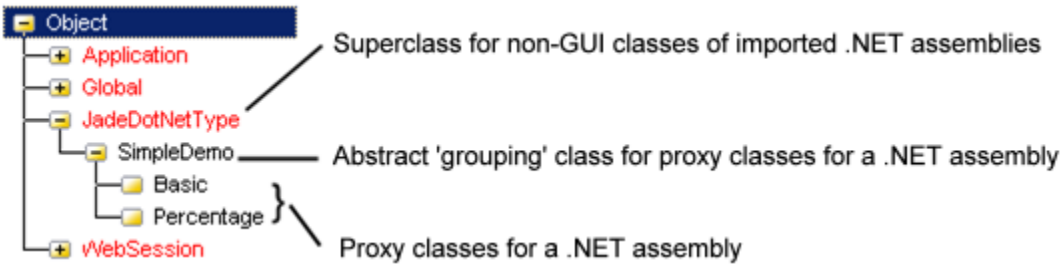
The following method shows how the generated classes and methods are used.

```
vars
    b, b2 : Basic;                          // Subclass of JadeDotNetType
begin
    // Create a JADE instance of the class representing the .NET object
    create b transient;
    // Optionally, specify where the actual .NET object is created
    b.usePresentationClient := false;
    // Request the creation of the corresponding .NET object
    b.createDotNetObject;
    // Use methods and properties on the JADE instance
    // to access members of the .NET object
    write b.integer;
    b2 := b.anotherOne;
epilog
    delete b;
    delete b2;
end;
```

See also "Updating .NET Properties on Value Types", in Chapter 4 of the *JADE External Interface Developer's Reference*.

Event handling on components is a little more complex. By default, component events are not passed onto JADE. To start receiving such events, the **beginEventNotification** method must be called.

When a .NET assembly is imported, the proxy classes for non-GUI .NET types are created as subclasses of the **JadeDotNetType** class, as shown in the following image.



For details about importing .NET assemblies, see "Importing .NET External Component Libraries", in Chapter 16 of the *JADE Development Environment User's Guide*. For details about the property and methods defined in the **JadeDotNetType** class, see "JadeDotNetType Property" and "JadeDotNetType Methods", in the following subsections.

For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeDotNetType Property

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

Property	Description
usePresentationClient	Specifies whether the non-GUI .NET component runs on the presentation client or the application server

usePresentationClient

Type: Boolean

The **usePresentationClient** property of the **JadeDotNetType** class specifies whether the .NET non-GUI component is run on the presentation client.

By default, .NET non-GUI components are run on the presentation client; that is, this value is set to **true**.

To run the non-GUI .NET component on the application server, set this property to **false**.

Note This property is ignored when the application is running on a standard client.

JadeDotNetType Methods

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

Method	Description
beginEventNotification	Causes events on a .NET component to be passed through to JADE for handling
createBoxedPrimitive	Creates a .NET object containing the JADE primitive value
createColor	Causes the .NET runtime to create a .NET **Color** object for which the receiver acts as a JADE proxy
createDotNetObject	Causes the .NET runtime to create the component .NET object for which the receiver acts as a JADE proxy
createEventNameMap	Defines a mapping between .NET events and JADE methods to be invoked
createFont	Causes the .NET runtime to create a .NET **Font** object for which the receiver acts as a JADE proxy
createPicture	Causes the .NET runtime to create a .NET **Image** object for which the receiver acts as a JADE proxy

Method	Description
enableEvent	Provides control at run time of JADE logic execution of an event associated with a component
endEventNotification	Stops events on a .NET component from being passed through to JADE for handling
getBoxedPrimitive	Returns a JADE primitive value for a .NET object that contains a type that maps to a JADE primitive type
getColor	Retrieves color information from a .NET **Color** object for which the receiver is a JADE proxy
getFont	Retrieves font information from a .NET **Font** object for which the receiver is a JADE proxy
getPicture	Retrieves binary image information from a .NET **Image** object for which the receiver is a JADE proxy

beginEventNotification

Signature `beginEventNotification(receiver: Object;`
`ref: Feature) updating;`

The **beginEventNotification** method of the **JadeDotNetType** class causes events on a .NET component to be passed through to JADE for handling.

The **receiver** parameter is the object to which events are delivered. The **ref** parameter is a reference property on the **receiver** of the type that contains the events.

Events are implemented in the JADE development environment by clicking on the **ref** property in the class of the **receiver**, which displays the events available in the method pane of the Class Browser. These events are shown grayed out (disabled) until you add code to that event. This is identical to defining an event on a control in a form by selecting the form, selecting the control, selecting a method (for example, the method) to implement, and then adding your event logic.

createBoxedPrimitive

Signature `createBoxedPrimitive(value: Any);`

The **createBoxedPrimitive** method of the **JadeDotNetType** class creates a .NET object containing the JADE primitive value. This enables a JADE primitive value to be used in .NET as if it were a .NET object.

The **value** parameter must be a JADE primitive type; for example, **Integer**, **Decimal**, **Boolean**, **TimeStamp**, **String** or **StringUtf8**. However, subclasses of the **Object** class and the primitive array classes are not allowed. An exception (14577 - Failed to create .NET object) is raised if the **value** parameter is not an allowed type.

The following example shows the use of the **createBoxedPrimitive** method.

```
vars
    a : Any;
    x : JadeDotNetType;
    t : TimeStamp;
begin
    create x;
    // Create .NET object from primitive
    // (similar to using createDotNetObject(), to create the .NET object)
    x.createBoxedPrimitive(t);
```

```
    // 'x' can now be used where a JadeDotNetObject is used
    // Get back a JADE primitive from previously created .NET object
    a := x.getBoxedPrimitive();
    write a;
epilog
    delete x;
end;
```

createColor

Signature `createColor(color: Integer);`

The **createColor** method of the **JadeDotNetType** class is a *helper* method that enables you to create a .NET **Color** object from the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if an assembly contains a class in .NET called **Basic** and the class has a **setColor** method that requires a **Color** object as a parameter, a **Basic** class with a **setColor** method is created when the assembly is imported into JADE. However, the **Color** type is not imported because it is not specific to the assembly.

The parameter to the **setColor** method in JADE is of the generic **JadeDotNetType** type.

The following method shows the use of the **createColor** method.

```
vars
    b : Basic;                          // Subclass of JadeDotNetType
    c : JadeDotNetType;                 // To represent a Color object
begin
    // Create a JADE instance of the class representing the .NET object
    create b transient;
    b.createDotNetObject;
    // Use the helper method to make a red Color object
    create c transient;
    c.createColor(Red);                 // Red is a global JADE color constant
    // Use the color object as a parameter to the setColor method
    b.setColor(c);
epilog
    delete b;
    delete c;
end;
```

createDotNetObject

Signature `createDotNetObject(): Any;`

The **createDotNetObject** method of the **JadeDotNetType** class causes the .NET run time to create the .NET object for which the receiver will act as a proxy.

A transient instance of the **JadeDotNetType** subclass is created and then this JADE object is used to create the .NET object by executing the **createDotNetObject** method. The following method shows the use of the **createDotNetObject** method.

```
vars
    b : Basic;                          // Subclass of JadeDotNetType
begin
    // Create an instance of the JadeDotNetType subclass
    create b transient;
    // Create the corresponding .NET object
```

```
    b.createDotNetObject;
    // Use the JADE proxy to access members of the .NET object
    write b.getDotNetTypeName;       // Outputs SimpleDemo.Basic
epilog
    delete b;
end;
```

The **createDotNetObject** method creates the .NET object using the default constructor, which has no parameters. Other constructors for the .NET component, which do have parameters, are imported and the corresponding JADE method name is **createDotNetObject_*n***, where ***n*** is a number to make the method name unique. For example, if the **createDotNetObject_1** method has a signature with a **StringUtf8** and an **Integer** parameter, you could use it to create a .NET object as follows:

```
vars
    b : Basic;                      // Subclass of JadeDotNetType
begin
    // Create an instance of the JadeDotNetType subclass
    create b transient;
    // Create the corresponding .NET object
    b.createDotNetObject_1("Meaning of Life", 42);
    // Use the JADE proxy to access members of the .NET object
    write b.getDotNetTypeName;     // Outputs SimpleDemo.Basic
epilog
    delete b;
end;
```

createEventNameMap

Signature `createEventNameMap() updating;`

The **createEventNameMap** method of the **JadeDotNetType** class is reimplemented in subclasses of **JadeDotNetType** by the .NET import wizard, to establish a mapping between the names of .NET events and the names of the JADE methods to be invoked.

You would not normally need to change the code generated for this method.

createFont

Signature

```
createFont(fontName:                          StringUtf8;
           pointSize:                         Real;
           bold, italic, strikeout, underline: Boolean);
```

The **createFont** method of the **JadeDotNetType** class is a *helper* method that enables you to create a .NET **Font** object from the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if an assembly contains a class in .NET called **Basic** and the class has a **setFont** method that requires a **Font** object as a parameter, a **Basic** class with a **setFont** method is created when the assembly is imported into JADE. However, the **Font** type is not imported because it is not specific to the assembly.

The parameter to the **setFont** method in JADE is of the generic **JadeDotNetType** type.

The following method shows the use of the **createFont** method.

```
vars
    b : Basic;                      // Subclass of JadeDotNetType
    f : JadeDotNetType;             // To represent a Font object
begin
```

```
    // Create a JADE instance of the class representing the .NET object
    create b transient;
    b.createDotNetObject;
    // Use the helper method to make a Font object (10 pt Arial bold)
    create f transient;
    f.createFont("Arial", 10, true, false, false, false);
    // Use the Font object as a parameter to the setFont method
    b.setFont(f);
epilog
    delete b;
    delete f;
end;
```

createPicture

Signature `createPicture(pic: Binary);`

The **createPicture** method of the **JadeDotNetType** class is a *helper* method that enables you to create a .NET **Image** object from the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if an assembly contains a class in .NET called **Basic** and the class has a **setImage** method that requires an **Image** object as a parameter, a **Basic** class with a **setImage** method is created when the assembly is imported into JADE. However, the **Image** type is not imported because it is not specific to the assembly.

The parameter to the **setImage** method in JADE is of the generic **JadeDotNetType** type.

The following method shows the use of the **createPicture** method.

```
vars
    b : Basic;                      // Subclass of JadeDotNetType
    i : JadeDotNetType;             // To represent an Image object
begin
    // Create a JADE instance of the class representing the .NET object
    create b transient;
    b.createDotNetObject;
    // Use the helper method to make an Image object
    create i transient;
    i.createPicture(app.loadPicture("c:\jade.bmp"));
    // Use the Image object as a parameter to the setImage method
    b.setImage(i);
epilog
    delete b;
    delete i;
end;
```

enableEvent

Signature
```
enableEvent(mth:     Method;
            enabled: Boolean);
```

The **enableEvent** method of the **JadeDotNetType** class enables you to control whether JADE logic associated with an event for a component is executed at run time.

Use the **mth** parameter to specify the name of the event that is to be disabled.

Set the **enable** parameter to **false** if you want to disable the event specified in the name parameter. All events are enabled by default.

An exception is raised if the event name specified in the **mth** parameter is not valid.

Notes Enabling or disabling an event has no impact if there is no logic associated with that event.

Event methods can be enabled or disabled in both standard client mode and in thin client mode.

endEventNotification

Signature `endEventNotification() updating;`

The **endEventNotification** method of the **JadeDotNetType** class stops events on a .NET component from being passed through to JADE for handling. This method ends notifications that were subscribed to by using the **beginEventNotification** method.

getBoxedPrimitive

Signature `getBoxedPrimitive(): Any;`

The **getBoxedPrimitive** method of the **JadeDotNetType** class returns a JADE primitive value for a .NET object that contains a type that maps to a JADE primitive type. An exception (14579 - Not a valid .NET object for this method) is raised if the .NET object is not of a type that can be mapped to a JADE primitive.

The following example shows the use of the **getBoxedPrimitive** method.

```
vars
    a : Any;
    x : JadeDotNetType;
    t : TimeStamp;
begin
    create x;
    // Create .NET object from primitive
    // (similar to using createDotNetObject(), to create the .NET object)
    x.createBoxedPrimitive(t);
    // 'x' can now be used where a JadeDotNetObject is used
    // Get back a JADE primitive from previously created .NET object
    a := x.getBoxedPrimitive();
    write a;
epilog
    delete x;
end;
```

getColor

Signature `getColor(): Integer;`

The **getColor** method of the **JadeDotNetType** class is a *helper* method that enables you to retrieve color information from a .NET **Color** object, which is in the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if a method in a .NET assembly returns a **Color** object, the **Color** type is not imported because it is not specific to the assembly when the assembly is imported into JADE.

The returned type for the imported method is the generic **JadeDotNetType** type.

The following method shows the use of the **getColor** method.

```
vars
    c : JadeDotNetType;                    // To represent a Color object
```

```
begin
    // Use createColor to make a blue Color object
    create c transient;
    c.createColor(Blue);            // Blue is a global JADE color constant
    // Use the helper method to get color information from a Color object
    write c.getColor;               // Outputs 16711680
epilog
    delete c;
end;
```

getFont

Signature `getFont(fontName: StringUtf8 io;`
`pointSize: Real io;`
`bold, italic, strikeout, underline: Boolean io);`

The **getFont** method of the **JadeDotNetType** class is a *helper* method that enables you to retrieve font information from a .NET **Font** object, which is in the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if a method in a .NET assembly returns a **Font** object, the **Font** type is not imported because it is not specific to the assembly when the assembly is imported into JADE.

The returned type for the imported method is the generic **JadeDotNetType** type.

The following method shows the use of the **getColor** method.

```
vars
    f : JadeDotNetType;                     // To represent a Font object
    fontName : StringUtf8;
    pointSize : Real;
    bold, italic, strikeout, underline : Boolean;
begin
    // Use createFont to make a Font object (10 pt Arial bold)
    create f transient;
    f.createFont("Arial", 10, true, false, false, false);
    // Use the helper method to retrieve font information from a Font object
    f.getFont(fontName, pointSize, bold, italic, strikeout, underline);
    write fontName;                         // Outputs Arial
    write pointSize;                        // Outputs 10
    write bold;                             // Outputs true
epilog
    delete f;
end;
```

getPicture

Signature `getPicture(): Binary;`

The **getPicture** method of the **JadeDotNetType** class is a *helper* method that enables you to retrieve binary image information from a .NET **Image** object, which is in the **System.Drawing.dll** assembly if that type has not been imported into JADE; for example, if a method in a .NET assembly returns an **Image** object, the **Image** type is not imported because it is not specific to the assembly when the assembly is imported into JADE.

The returned type for the imported method is the generic **JadeDotNetType** type.

The following method shows the use of the **getPicture** method.

```
vars
    i : JadeDotNetType;                    // To represent an Image object
begin
    // Use createPicture to make an Image object
    create i transient;
    i.createPicture(app.loadPicture("c:\jade.bmp"));
    // Use the helper method to retrieve binary picture information
    write i.getPicture.pictureType;        // Outputs 1 (PictureType_Bitmap)
epilog
    delete i;
end;
```

JadeDynamicObject Class

The transient **JadeDynamicObject** class implements the structure and behavior of dynamic objects. A JADE dynamic object is a self-describing object whose attributes are determined at run time.

The **JadeDynamicObject** class has two fixed attributes: **type** and **name**, which you can use to determine the runtime type of a dynamic object. The JADE inspector displays the type, name, and dynamic attribute names and values of **JadeDynamicObject** instances.

Note If the type of a property is removed by deleting the class or removing the schema and the property has been assigned a value, the value is no longer valid and attempting to use it will raise exception 1046 *(Invalid class number)*.

The method in the following example creates a dynamic object and passes this to the **Collection** class **getStatistics** method, which populates the object with various statistical attributes and their values, returning the dynamic object to the caller. The method then uses the **getPropertyName**, **getPropertyValueByIndex**, and **propertyCount** methods to display the statistical attribute name and value pairs. As the calling method created the **jdo** variable, it is also responsible for deletion, which is performed in an epilog.

```
vars
    jdo   : JadeDynamicObject;
    str   : String;
    int   : Integer;
    count : Integer;
begin
    create jdo;
    node.processes.getStatistics(jdo);
    str := '---' & jdo.getName & '(' & jdo.type.String & ')---';
    count := jdo.propertyCount;
    foreach int in 1 to count do
        str := str & CrLf & jdo.getPropertyName(int) &
                            " = " & jdo.getPropertyValueByIndex(int).String;
    endforeach;
    write str;
epilog
    delete jdo;
end;
```

The method in the following example uses the **Process** class **getRequestStatistics** method to retrieve the process ticks used to create an object.

```
getProcessTicks();
vars
    sample                 : JadeDynamicObject;
    cumulativeProcessTicks : Integer64;
    processTicks           : Integer64;
    person                 : Person;
begin
    create sample transient;
    process.getRequestStatistics(sample, 1);    // local statistics
    cumulativeProcessTicks :=
                        sample.getPropertyValue("processTicks").Integer64;
    beginTransaction;
    create person persistent;
    person.surname   := "Smith";
```

```
    person.firstName := "John";
    commitTransaction;
    process.getRequestStatistics(sample, 1);
    processTicks := sample.getPropertyValue("processTicks").Integer64 -
                                        cumulativeProcessTicks;
    delete sample;
    write "Process ticks to create an object = " & processTicks.String;
end;
```

For details about the properties and methods defined in the **JadeDynamicObject** class, see "JadeDynamicObject Properties" and "JadeDynamicObject Methods", in the following subsections. For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeDynamicObject Properties

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

Property	Contains the …
children	Array containing dynamic objects
name	Runtime name of the dynamic object instance in textual format
parent	Dynamic object reference that is the inverse of **children**
type	Type of the runtime dynamic object instance, which you can use to distinguish between dynamic object types

children

Type: JadeDynamicObjectArray

The **children** property of the **JadeDynamicObject** class is a container of dynamic child objects that can be used to create an aggregate structure.

name

Type: String[128]

The **name** property of the **JadeDynamicObject** class contains the runtime name of the **JadeDynamicObject** instance in textual format.

parent

Type: JadeDynamicObject

The **parent** property of the **JadeDynamicObject** class is the inverse of the **children** property.

type

Type: Integer[4]

The **type** property of the **JadeDynamicObject** class contains the runtime type of the **JadeDynamicObject** instance, which you can use to distinguish between dynamic object types.

For details about the dynamic object types are returned from **JadeDatabaseAdmin** class SDS query methods, see "Returning Information about SDS Entities", in Chapter 10 of the *JADE Developer's Reference*.

JadeDynamicObject Methods

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

Method	Description
addProperty	Adds the specified property at run time
clear	Clears the contents of the dynamic object at run time
clearValues	Clears the contents of values assigned to the receiver using the **setPropertyValue** method to set the value to **null**
display	Returns a list of attributes names and values as a string
getName	Returns the name of the dynamic object
getPropertyName	Returns the name of a property at the specified relative index
getPropertyType	Returns a reference to the type of a property with the specified name
getPropertyTypeByIndex	Returns a reference to the type of a property at the specified relative index
getPropertyValue	Returns the value of a property with the specified name
getPropertyValueByIndex	Returns the value of a property at the specified relative index
propertyCount	Returns the number of dynamic properties defined for the receiver
setPropertyValue	Sets the value of the property specified in the **pname** parameter to the value specified in the **value** parameter at run time
setPropertyValueByIndex	Sets the value of the property at the relative position specified in the **index** parameter to the value specified in the **value** parameter at run time

addProperty

Signature

```
addProperty(name: String;
            type: Type) updating;
```

The **addProperty** method of the **JadeDynamicObject** class adds the property with the specified **name** and **type** to the receiver at run time. An exception is raised if the value of the **name** parameter conflicts with an existing fixed or dynamic property name.

The **name** parameter cannot exceed 100 characters.

clear

Signature `clear();`

The **clear** method of the **JadeDynamicObject** class clears the contents of the receiver at run time, by deleting all user-defined properties added by calling the **addProperty** method and all values set by calling the **setPropertyValue** method.

clearValues

Signature `clearValues();`

The **clearValues** method of the **JadeDynamicObject** class clears the contents of values assigned to the receiver using the **setPropertyValue** method to set the value to **null**. All property descriptions added to the receiver using the **addProperty** method are retained. As a side effect, any dynamic memory assigned to store variable-length string or binary values is released.

display

Signature `display(): String;`

The **display** method of the **JadeDynamicObject** class returns a string containing a list of attributes names and values as a string, with the name and value (by index) of each attribute separated by carriage return/line feed (**CR/LF**) characters.

getName

Signature `getName(): String;`

The **getName** method of the **JadeDynamicObject** class returns a string containing the name of the dynamic object.

getPropertyName

Signature `getPropertyName(index: Integer): String;`

The **getPropertyName** method of the **JadeDynamicObject** class returns a string containing the name of the property at the relative index specified in the **index** parameter. If the value of the **index** parameter is outside the range **1** through **propertyCount**, an exception is raised.

getPropertyType

Signature `getPropertyType(name: String): Type;`

The **getPropertyType** method of the **JadeDynamicObject** class returns a reference to the type of the property specified in the **name** parameter.

If the receiver does not define the property name, an exception is raised.

getPropertyTypeByIndex

Signature `getPropertyTypeByIndex(index: Integer): Type;`

The **getPropertyTypeByIndex** method of the **JadeDynamicObject** class returns a reference to the type of the property at the relative position specified in the **index** parameter.

If the value of the **index** parameter is outside the range **1** through **propertyCount**, an exception is raised.

getPropertyValue

Signature `getPropertyValue(name: String): Any;`

The **getPropertyValue** method of the **JadeDynamicObject** class returns the value of the property specified in the **name** parameter.

If the receiver does not define the property name, an exception is raised.

getPropertyValueByIndex

Signature `getPropertyValueByIndex(index: Integer): Any;`

The **getPropertyValueByIndex** method of the **JadeDynamicObject** class returns the value of the property at the relative position specified in the **index** parameter.

If the value of the **index** parameter is outside the range **1** through **propertyCount**, an exception is raised.

propertyCount

Signature `propertyCount(): Integer;`

The **propertyCount** method of the **JadeDynamicObject** class returns the number of dynamically defined properties for the receiver.

setPropertyValue

Signature
```
setPropertyValue(name: String;
                 value: Any) updating;
```

The **setPropertyValue** method of the **JadeDynamicObject** class sets the value of the property specified in the **name** parameter to the value specified in the **value** parameter at run time.

setPropertyValueByIndex

Signature
```
setPropertyValueByIndex(index: Integer;
                        value: Any) updating;
```

The **setPropertyValueByIndex** method of the **JadeDynamicObject** class sets the value of the property at the relative position specified in the **index** parameter to the value specified in the **value** parameter at run time.

If the value of the **index** parameter is outside the range **1** through **propertyCount**, an exception is raised.

JadeDynamicObjectArray Class

The **JadeDynamicObjectArray** class is an array-based container whose members are dynamic objects and can be used by methods that need to return a variable number of dynamic object instances from a single method call.

The dynamic objects are referenced by their position in the collection.

The bracket (**[]**) subscript operators enable you to assign values to and receive values from a dynamic object array.

Inherits From: ObjectArray

Inherited By: (None)

JadeDynamicPropertyCluster Class

The **JadeDynamicPropertyCluster** class encapsulates the behavior required to extend a class, by storing dynamic properties.

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

For details about dynamic clusters and properties, see "Dynamic Clusters and Properties", in Chapter 4 of the *JADE Development Environment User's Guide*, and to "Runtime Dynamic Properties", in Chapter 21 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeDynamicPropertyCluster Properties

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

Property	Contains the …
name	Read-only name of the dynamic property cluster
properties	Collection of dynamic properties in the cluster
schemaType	Type of the dynamic property cluster instance

name

Type: String[100]

The **name** property of the **JadeDynamicPropertyCluster** class contains the read-only name of the dynamic property cluster.

properties

Type: PropertyNDict

The **properties** property of the **JadeDynamicPropertyCluster** class contains a read-only collection of dynamic properties in the cluster.

schemaType

Type: Type

The **schemaType** property of the **JadeDynamicPropertyCluster** class contains the read-only type of the dynamic property cluster instance; that is, the type of the class that is being extended by the dynamic properties.

JadeDynamicPropertyCluster Methods

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

Method	Description
addDynamicProperty	Adds a new dynamic property to the receiving cluster using the specified information
addExclusiveDynamicProperty	Adds a new exclusive dynamic property to the receiving cluster using the specified information
deleteDynamicProperty	Deletes the dynamic property with the specified name from the receiving cluster
findDynamicProperty	Returns the dynamic property with the specified name from the receiving cluster

addDynamicProperty

Signature

```
addDynamicProperty(propertyName: String;
                   propertyType: Type;
                   length:       Integer;
                   scaleFactor:  Byte): Property;
```

The **addDynamicProperty** method of the **JadeDynamicPropertyCluster** class adds a new runtime dynamic property with the name specified by the **propertyName** parameter name to the receiving dynamic property cluster and returns the newly created property.

If the type of the property is:

- **Binary**, **String**, or **StringUtf8**, the length must be specified in the **length** parameter and zero (**0**) must be specified for the **scaleFactor** parameter.
- **Decimal**, the precision and number of decimal places must be specified in the **length** and **scaleFactor** parameters, respectively.
- A class or collection class, zero (**0**) must be specified as the value for the **length** and **scaleFactor** parameters.
- A collection, the property is created as a shared reference.

addExclusiveDynamicProperty

Signature

```
addExclusiveDynamicProperty(propertyName: String;
                            propertyType: Type): Property;
```

The **addExclusiveDynamicProperty** method of the **JadeDynamicPropertyCluster** class adds a new exclusive runtime dynamic property (for example, an exclusive collection reference) with the name specified by the **propertyName** parameter name to the receiving dynamic property cluster and returns the newly created property.

The **propertyType** parameter must be a subclass of the **Collection** class.

deleteDynamicProperty

Signature `deleteDynamicProperty(propertyName: String);`

The **deleteDynamicProperty** method of the **JadeDynamicPropertyCluster** class deletes the runtime dynamic property with the name specified in the **propertyName** parameter from the receiving cluster.

Note If instances of the class exist, you cannot delete a dynamic property. A non-embedded blob, slob, or slobutf8 dynamic property has a size or length greater than 540.

The maximum length of an embedded **Binary** type is 540 characters, an embedded **StringUtf8** type is 540 characters, and an embedded **String** type is 539 characters.

You can delete a runtime dynamic property only if the class in which it is defined is not being used by any other process. If production mode is set, a dynamic property can be deleted in single user mode only.

An exception is raised if the runtime dynamic property is not defined in the cluster.

findDynamicProperty

Signature `findDynamicProperty(name: String): Property;`

The **findDynamicProperty** method of the **JadeDynamicPropertyCluster** class returns the runtime dynamic property with the name specified in the **name** parameter from the receiving cluster or it returns null if the specified runtime dynamic property is not defined in the cluster.

JadeGenericMessage Class

The transient **JadeGenericMessage** class encapsulates the behavior required to build the content of a message before it is sent and to examine the content and properties of a message after it is received.

The behavior of creating a message, adding a message to a queue, and retrieving a message from a queue is provided by the **createMessage**, **getMessage**, and **putMessage** methods, respectively, in the **JadeGenericQueue** class.

The message body (often referred to as the *payload*) is a variable-length binary whose content and format is user-defined. The message body can be considered as just another property of the message.

The message header is a series of properties that provide out-of-band data about the payload; for example, the creation timestamp, sender details, encoding, message type, and reply details.

For details about the use of the JADE messaging framework, see Chapter 15, "Using the Messaging Framework", in the *JADE Developer's Reference*.

For details about the constants, properties, and methods defined in the **JadeGenericMessage** class, see "JadeGenericMessage Class Constants", "JadeGenericMessage Properties", and "JadeGenericMessage Methods", in the following subsections. For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeGenericMessage Class Constants

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

Constant	Integer Value
Feedback_Appl_First	65536
Feedback_Expiration	258
Feedback_NAN	276
Feedback_None	0
Feedback_PAN	275
Report_Expiration	#200000
Report_None	0
Type_Appl_First	65536
Type_Datagram	1
Type_Reply	3
Type_Report	4
Type_Request	2
Type_Unspecified	0

JadeGenericMessage Properties

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

Property	Contains the …
body	Information content of the message (often referred to as the *payload*)
correlationID	Identifier used to associate a reply message with the corresponding request message
createdWhen	Date and time value when the message was added to the message queue
expiresWhen	Date and time value when the sender of the message considers the message to have expired
feedback	Type of a report message
format	Format of the data in the body of the message (for example, binary, text/UTF8, JadeTpls)
messageID	Unique identifier assigned to the message when it is added to a queue
replyQueueFullName	Name of the queue into which replies to this message should be put
report	Specification of the report messages that are required
retrievedWhen	Date and time value when the message was retrieved from the queue
senderID	Details of the process that added the message into the queue
tag	User-defined string value for the message
type	Integer value used by the process that retrieves the message to identify its purpose

body

Type: Binary

The **body** property of the **JadeGenericMessage** class contains the information content of the message (often referred to as the *payload*) in a binary format. Assignment to the **body** property causes any data previously appended to be discarded.

The body of a message can also be built by appending data using the **appendBinary**, **appendString**, **appendStringAsUtf8**, or **appendStringUtf8** methods of the **JadeGenericMessage** class.

The **body** property cannot be assigned to after calling the **appendBodyTuple** method of the **JadeGenericMessage** class or the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

correlationID

Type: Binary

The **correlationID** property of the **JadeGenericMessage** class contains an identifier used to associate a reply message with the corresponding request message.

When a reply message is built, the **correlationID** property of the reply message is set to the **messageID** property of the request message. When the reply is received, the **correlationID** property identifies the request message previously sent.

The **correlationID** property cannot be set after calling the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

In the following method example, a request message is sent and a reply is received.

```
request();
vars
    msg : JadeGenericMessage;
    corrID : Binary;
begin
    msg := myRequestQueue.createMessage(true);
    msg.replyQueueFullName := myReplyQueue.fullName;
    msg.appendString("What is the meaning of life?");
    myRequestQueue.putMessage(msg, null);
    corrID := msg.messageID;
    msg.initializeForGet;
    myReplyQueue.getMessageByCorrelationID(msg, "", corrID);
    write msg.body.String;                                  // writes "42"
epilog
    delete msg;
end;
```

In the following method example, a request message is received and a reply is sent.

```
reply();
vars
    get : JadeGenericMessage;
    put : JadeGenericMessage;
    replyQueue : JadeGenericQueue;
    factory : JadeMessagingFactory;
begin
    get := myQueue.createMessage(false);
    myQueue.getMessage(get, null);
    create factory transient;
    replyQueue := factory.openQueue(get.replyQueueFullName,
                                    "Access=Public;Usage=Put");
    replyQueue.inspectModal;
    put := replyQueue.createMessage(true);
    put.correlationID := get.messageID;
    put.appendString("42");
    replyQueue.putMessage(put, null);
epilog
    delete get;
    delete put;
    delete factory;
    delete replyQueue;
end;
```

Note As some transports such as WebSphere have **correlationID** values with trailing null characters that are part of the **correlationID**, you should not store **correlationID** values in fixed-length **Binary** variables.

createdWhen

Type: TimeStamp

The read-only **createdWhen** property of the **JadeGenericMessage** class contains the date and time at which the message was added to the message queue by using the **putMessage** method or the **beginMessage** method of the **JadeGenericQueue** class.

expiresWhen

Type: TimeStamp

The read-only **expiresWhen** property of the **JadeGenericMessage** class contains the date and time at which the sender of the message considers the message to have expired. The value of the property can be set by the sender of the message using the **setExpiryRelativeToNow** method of the **JadeGenericMessage** class, but this method cannot be called after the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

If an expiry value is not explicitly set, it is effectively set to a valid date and time in the very distant future, which can be correctly compared with a current timestamp value.

Your application can decide how to deal with messages that the sender considers to have expired.

feedback

Type: Integer

The **feedback** property of the **JadeGenericMessage** class contains additional information about the type of a report message. This property has meaning only for reports with a **type** property of **Type_Report**.

You can set the **feedback** property to one of the **JadeGenericMessage** class constants listed in the following table.

Class Constant	Type of Report Message
Feedback_Expiration	An expiry report message indicating that an application attempted to retrieve an expired message
Feedback_NAN	A negative action notification report message indicating that a request has not been successfully serviced
Feedback_None	A report message with an unspecified type
Feedback_PAN	A positive action notification report message indicating that a request has been successfully serviced

format

Type: String

The **format** property of the **JadeGenericMessage** class contains the format of the data in the body of the message (for example, binary, text/UTF8, **JadeTpls**, and so on).

The **format** property cannot be set after calling the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

Your application can decide how to deal with messages that have a particular format.

messageID

Type: Binary

The read-only **messageID** property of the **JadeGenericMessage** class contains a unique identifier in a binary format that is assigned to the message when it is added to a queue using the **putMessage** method or the **beginMessage** method of the **JadeGenericQueue** class.

The format and content of the **messageID** property depend on the message transport.

Note As some transports such as WebSphere have **messageID** values with trailing null characters that are part of the **messageID**, you should not store **messageID** values in fixed-length **Binary** variables.

replyQueueFullName

Type: String[250]

The **replyQueueFullName** property of the **JadeGenericMessage** class contains the full name of the queue into which replies to this message should be put.

The **replyQueueFullName** property would be set by the sender of a request message; that is, a message from which the sender expects to receive a reply. The process replying to the message is expected to add the reply message to the queue identified by the **replyQueueFullName** property. The transport name part of the **replyQueueFullName** must match the transport name of the message.

The **replyQueueFullName** property cannot be set after calling the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

report

Type: Integer

The **report** property of the **JadeGenericMessage** class contains the report messages that are required.

You can set the **report** property to one of the **JadeGenericMessage** class constants listed in the following table.

Class Constant	Report Messages Requested
Report_Expiration	Expiry messages when an application attempted to retrieve an expired message
Report_None	No messages are requested

If report messages are requested then the **replyQueueFullName** property must not be blank.

retrievedWhen

Type: TimeStamp

The read-only **retrievedWhen** property of the **JadeGenericMessage** class contains the date and time at which the message was retrieved from the queue by using the **getMessage** method of the **JadeGenericQueue** class.

senderID

Type: String

The read-only **senderID** property of the **JadeGenericMessage** class contains details of the process that added the message into the queue by using the **putMessage** method or the **beginMessage** method of the **JadeGenericQueue** class.

The format and content of the **senderID** property depend on the message transport.

tag

Type: String

The **tag** property of the **JadeGenericMessage** class contains a string value that is stored with the message object. Unlike other properties, the value of the **tag** property is not used by JADE and is available for you to use for any purpose in your application.

Note Although the **tag** property is a property in the **JadeGenericMessage** object, it is not considered part of the message. Consequently it is not part of the data that is put in the queue and is not available for retrieval.

By default, the **tag** property is set to a null string (**""**).

type

Type: Integer

The **type** property of the **JadeGenericMessage** class contains an integer value that can be used by the process that retrieves the message, to identify the purpose of the message.

You can set the **type** property to one of the **JadeGenericMessage** class constants listed in the following table.

Class Constant	Purpose of Message
Type_Datagram	A simple message for which no reply is expected
Type_Request	A message for which a reply is expected
Type_Reply	A reply to a request message
Type_Report	A message that describes an event such as the occurrence of an error

You can use other values, but some transports have reserved ranges (for example, WebSphere recommends the range **MQMT_APPL_FIRST** (65536) through **MQMT_APPL_LAST** (999,999,999)).

JadeGenericMessage Methods

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

Method	Description
appendBinary	Adds the specified binary information to the end of the message body
appendBodyTuple	Adds a tuple (name, type, and value) to the end of the message body
appendString	Adds the specified text to the end of the message body
appendStringAsUtf8	Encodes the specified text as UTF8 and adds to the end of the message body
appendStringUtf8	Adds the specified UTF8 text to the end of the message body
getBodyLength	Returns the length of the message body in bytes
getBodyTuple	Scans the text in the message body for a tuple with the specified name
getMessageProperty	Returns the value of the specified property
getTransportName	Returns the name of the transport associated with the message

Method	Description
getUtf8bodyAsString	Converts the UTF8 encoded text in the message body to a JADE string and returns that value
initializeForGet	Initializes an existing message object before retrieving a message from a queue
initializeForPut	Initializes an existing message object before building the message and adding it to a queue
setExpiryRelativeToNow	Sets a message expiry date and time in the future from a specified value
setMessageProperty	Sets a specified property to a specified value

appendBinary

Signature `appendBinary(data: Binary);`

The **appendBinary** method of the **JadeGenericMessage** class adds the binary information specified by the **data** parameter to the end of the message body. A message can be constructed by calling the **appendBinary** method a number of times. You can call the **appendBinary** method before or after calling the **appendString**, **appendStringAsUtf8**, and **appendStringUtf8** methods or directly assigning a value to the **body** property. If you call the **appendBinary** method before or after calling the **appendBodyTuple** method, an exception is raised.

appendBodyTuple

Signature
```
appendBodyTuple(tuplename: String;
                value:     Any);
```

The **appendBodyTuple** method of the **JadeGenericMessage** class adds a tuple to the end of the message body. A tuple has a name, a type, and a single primitive value. The **tuplename** parameter, which is case-sensitive, must start with a letter. The subsequent characters to a maximum of 89 characters can be letters, digits, underscores, or periods.

The **body** property of the message must be empty before the **appendBodyTuple** method is called for the first time. A partially built message can be discarded by calling the **initializeForPut** method.

In the following example of sending of tuple data, a message containing the name and the shoe size of a person is constructed.

```
msg.appendBodyTuple("name", "wilbur");
msg.appendBodyTuple("shoe_size", 6.5);
```

The first tuple that is appended has a value for the **tuplename** parameter of **"name"**, a value for the **value** parameter of **"wilbur"**, and the type of data is **String**. The second tuple has a value for the **tuplename** parameter of **"shoe_size"**, a value for the **value** parameter of **6.5**, and the type of data is **Real**.

No check is made to determine if a tuple is already present with the same name. When the message is read using the **getBodyTuple** method, only the first tuple in the message with a matching name is returned; all other duplicates are ignored.

A message can be constructed by calling the **appendBodyTuple** method a number of times. If you call the **appendBodyTuple** method before or after calling and **appendBinary**, **appendStringUtf8**, **appendString**, or **appendStringAsUtf8** methods or directly assigning a value to the **body** property, an exception is raised.

The **format** property of the message is automatically set to **"JadeTpls"**.

Note Take care when using this mechanism to send an object reference to another process.

appendString

Signature `appendString(data: String);`

The **appendString** method of the **JadeGenericMessage** class adds the text in the **data** parameter to the end of the message body.

A message can be constructed by calling the **appendString** method a number of times. You can call the **appendString** method before or after calling the **appendBinary**, **appendStringAsUtf8**, and **appendStringUtf8** methods or directly assigning a value to the **body** property. If you call the **appendString** method before or after calling the **appendBodyTuple** method, an exception is raised.

appendStringAsUtf8

Signature `appendStringAsUtf8(data: String);`

The **appendStringAsUtf8** method of the **JadeGenericMessage** class encodes the content of the string in the **data** property as a UTF8 string that is added to the end of the message body.

In an ANSI environment, the **currentLocale** property of the application determines the code page that is used to encode characters with decimal values 128 through 255.

A message can be constructed by calling the **appendStringAsUtf8** method a number of times. You can call the **appendStringAsUtf8** method before or after calling the **appendBinary**, **appendString**, and **appendStringUtf8** methods or directly assigning a value to the **body** property. If you call the **appendStringAsUtf8** method before or after calling the **appendBodyTuple** method, an exception is raised.

appendStringUtf8

Signature `appendStringUtf8(data: StringUtf8);`

The **appendStringUtf8** method of the **JadeGenericMessage** class adds the UTF8 text in the **data** parameter to the end of the message body.

A message can be constructed by calling the **appendStringUtf8** method a number of times. You can call the **appendStringUtf8** method before or after calling the **appendBinary**, **appendString**, and **appendStringAsUtf8** methods or directly assigning a value to the **body** property. If you call the **appendStringUtf8** method before or after calling the **appendBodyTuple** method, an exception is raised.

getBodyLength

Signature `getBodyLength(): Integer;`

The **getBodyLength** method of the **JadeGenericMessage** class returns the length of the message body in bytes.

getBodyTuple

Signature
```
getBodyTuple(tuplename: String;
             value:     Any output): Boolean;
```

The **getBodyTuple** method of the **JadeGenericMessage** class scans the text in the message body for a tuple with the name specified by the **tuplename** parameter and places the corresponding value in the variable passed to the method as the **value** parameter. The method returns **true** if the tuple is found.

An exception is raised if the type of the variable passed as the **value** parameter does not match the type of the tuple. A variable of type **Any** is compatible with all tuple values.

An exception is raised if the **format** property of the message is not "**JadeTpls**", or the message body does not match the format generated by the **appendBodyTuple** method.

In the following code fragment, the value of a tuple called **"name"** is retrieved by using the **getBodyTuple** method and stored in a string variable.

```
msg.getBodyTuple("name", str);
txtName.text := str;
```

getMessageProperty

Signature `getMessageProperty(propname: String;`
`value: Any output): Boolean;`

The **getMessageProperty** method of the **JadeGenericMessage** class returns the value of the *user* property specified in the **propname** parameter. User properties are currently only supported on **JadeMQ** messages. They are primitive values that are bundled with the message but are not part of the body. The name must begin with an uppercase **X** or a lowercase **x** character.

The result is returned through the **value** output parameter, which is of type **Any**. If the property name is invalid, an exception is raised.

An exception is raised if the type of the variable passed as the **value** parameter does not match the type of the property. A variable of type **Any** is compatible with all property values.

getTransportName

Signature `getTransportName(): String;`

The **getTransportName** method of the **JadeGenericMessage** class returns the name of the transport associated with the message, which must be the same as the transport that created the message object.

getUtf8bodyAsString

Signature `getUtf8bodyAsString(): String;`

The **getUtf8bodyAsString** method of the **JadeGenericMessage** class converts the UTF8 encoded text in the message body to a standard JADE string, which is then returned.

An exception is raised if the body is not a valid UTF8 string or it contains Unicode characters that cannot be represented in an ANSI environment.

initializeForGet

Signature `initializeForGet();`

The **initializeForGet** method of the **JadeGenericMessage** class initializes an existing message object so that a message can be retrieved from a queue. The initialization sets the values of message properties to default values, sets the values of user properties to null, and sets the message body to empty.

This method must be called before calling the **getMessage** method of the **JadeGenericQueue** class.

The **initializeForGet** method is implicitly called when a message is created by calling the **createMessage** method of the **JadeGenericQueue** class with the **forPUT** parameter set to **false**.

initializeForPut

Signature `initializeForPut();`

The **initializeForPut** method of the **JadeGenericMessage** class initializes an existing message object so that a new message can be built and then added to a queue. The initialization sets the values of message properties to default values, sets the values of user properties to null, and sets the message body to empty.

When this method is called, an incomplete message, including any segments that have already been sent, is discarded. This method must be called before calling the **putMessage** method of the **JadeGenericQueue** class.

The **initializeForPut** method is implicitly called when a message is created by calling the **createMessage** method of the **JadeGenericQueue** class with the **forPUT** parameter set to **true**.

setExpiryRelativeToNow

Signature `setExpiryRelativeToNow(lifeTimeSeconds: Integer);`

The **setExpiryRelativeToNow** method of the **JadeGenericMessage** class sets the **expiresWhen** property of the message to a date and time value resulting from adding the number of seconds specified by the value of the **lifeTimeSeconds** parameter to the current date and time. This method cannot be called after the **beginMessage** or **putMessage** method of the **JadeGenericQueue** class.

setMessageProperty

Signature
```
setMessageProperty(propname: String;
                   value:    Any);
```

The **setMessageProperty** method of the **JadeGenericMessage** class sets the *user* property of the receiver specified in the **propname** parameter to the value specified in the **value** parameter. User properties are currently only supported on **JadeMQ** messages. They are primitive type values that are bundled with the message but are not part of the body.

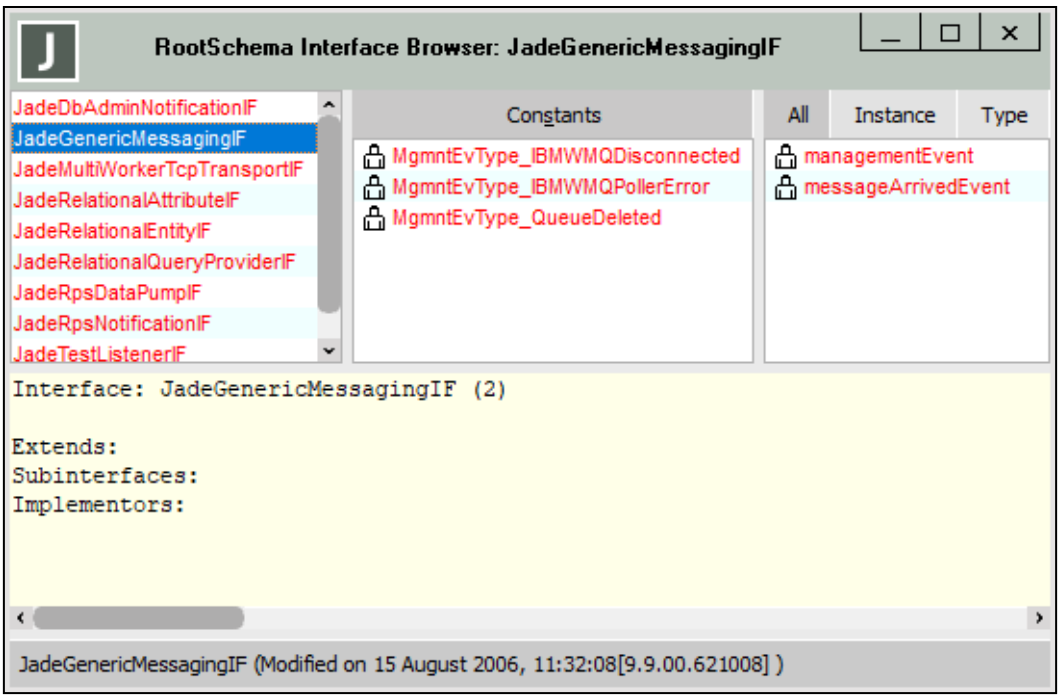
The name must begin with an uppercase **X** or a lowercase **x** character.

If the property specified in the **propname** parameter is invalid, an exception is raised.

JadeGenericMessagingIF Interface

The **JadeGenericMessagingIF** interface, defined in the **RootSchema**, provides the definition of the event callback methods that you can implement in your user schema classes for events such as a message arriving in a queue or a message queue being deleted.

You can view the **JadeGenericMessagingIF** interface and its constant and 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 **JadeGenericMessagingIF** 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*.

The automatically generated stub methods in classes that implement the interface contain no body logic.

Note It is your responsibility to provide the source that meets your application requirements for each stub method.

For details about the **JadeGenericMessagingIF** interface constants and methods, see "JadeGenericMessagingIF Interface Constants" and "JadeGenericMessagingIF Interface Method Callback Signatures", in the following subsections.

JadeGenericMessagingIF Interface Constants

The constants provided by the **JadeGenericMessagingIF** interface are listed in the following table.

Constant	Integer Value
MgmntEvType_IBMWMQDisconnected	2
MgmntEvType_IBMWMQPollerError	3
MgmntEvType_QueueDeleted	1

JadeGenericMessagingIF Interface Callback Method Signatures

The signatures of callback methods provided the **JadeGenericMessagingIF** interface are summarized in the following table.

Method	Callback method for the …
managementEvent	Management event
messageArrivedEvent	Message arrived in a queue event

managementEvent

Signature
```
managementEvent(queue: JadeGenericQueue input;
                type:  Integer;
                info:  Any);
```

When a management event occurs, the object that implements the **JadeGenericMessagingIF** interface is notified and executes its implementation of the **managementEvent** interface method.

Management events relate to changes to the messaging environment. Currently the only management event that has been defined is the deletion of the message queue specified by the **queue** parameter. The value of the **type** parameter in this case is provided by the interface constant **MgmntEvType_QueueDeleted**, which is zero (**0**).

messageArrivedEvent

Signature
```
messageArrivedEvent(queue: JadeGenericQueue input);
```

When the event corresponding to the arrival of a message in a queue occurs, the object that implements the **JadeGenericMessagingIF** interface is notified and executes its implementation of the **messageArrivedEvent** interface method.

The **queue** parameter specifies the **JadeGenericQueue** object into which a message had been put.

Notes In your implementation of the **messageArrivedEvent**, you must retrieve a message from the queue by using the **getMessage** method of the **queue** parameter. If you do not do this, the **messageArrivedEvent** is triggered again and the processing loops.

The **getMessage** method should be called with the **NoWait** option, and your application should handle a *no message found* situation if another process has already retrieved the message.

JadeGenericQueue Class

The **JadeGenericQueue** class encapsulates a destination to which messages can be sent and from which messages can be retrieved. A transient instance of this class is created by invoking the **openQueue** method on a transient instance of the **JadeMessagingFactory** class. The method returns a queue for the specified transport, which is ready to be accessed and used in the specified ways (sending messages, retrieving messages, or inquiring on message properties).

The following example shows how a queue is created.

```
vars
    factory : JadeMessagingFactory;
begin
    create factory transient;
    myQueue := factory.openQueue("JadeMQ://localnode/TestQ",
                                 "Access=Public; MustExist; Usage=Inq");
epilog
    delete factory;
end;
```

For details about the use of the JADE messaging framework, see Chapter 15, "Using the Messaging Framework", in the *JADE Developer's Reference*.

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

For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeGenericQueue Class Constants

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

Constant	Integer Value
Notify_Continuous	0
Notify_OneShot	1

JadeGenericQueue Properties

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

Property	Description
defaultGetMessageOptions	Enables you to specify a set of default options for subsequent **getMessage** method calls
defaultPutMessageOptions	Enables you to specify a set of default options for subsequent **putMessage** method calls

Property	Description
fullName	Contains the complete name of the queue including the transport type and queue manager name
maxMemoryInuse	Specifies the maximum amount of memory that is available for messages in the queue
maxMessageCount	Specifies the maximum number of messages that can be present in the queue
maxMessageLength	Specifies the length of the longest message that can be put into the queue
name	Specifies the name of the queue
tag	User-defined string value for the queue

defaultGetMessageOptions

Type: String

The **defaultGetMessageOptions** property of the **JadeGenericQueue** class contains a set of default options that are used by subsequent calls to the **getMessage** method of the **JadeGenericQueue** class.

Although you can specify these options each time by using the **options** parameter of the **getMessage** method, it is more efficient to use the **defaultGetMessageOptions** property to provide a pre-compiled set of options, which avoids the need to parse a full list of options on each call.

Options passed to the **getMessage** method through the **options** parameter override those specified in the **defaultGetMessageOptions** property.

defaultPutMessageOptions

Type: String

The **defaultPutMessageOptions** property of the **JadeGenericQueue** class contains a set of default options that are used by subsequent calls to the **putMessage** method of the **JadeGenericQueue** class.

Although you can specify these options each time by using the **options** parameter of the **putMessage** method, it is more efficient to use the **defaultPutMessageOptions** property to provide a pre-compiled set of options avoiding the need to parse a full list of options on each call.

Options passed to the **putMessage** method through the **options** parameter override those specified in the **defaultPutMessageOptions** property.

fullName

Type: String[250]

The read-only **fullName** property of the **JadeGenericQueue** class contains the name of the queue qualified by the names of the transport type and the queue manager. The format of the **fullName** parameter in the format *transportName* :// *queueManagerName* / *queueName*, as shown in the following example.

```
JadeMQ://edf25885-eaf1-db11-9b68-b50f1e595343/TestQ
```

maxMemoryInuse

Type: Integer

The **maxMemoryInuse** property of the **JadeGenericQueue** class contains the maximum amount of memory that is available for messages in the queue. The behavior depends on the transport. In the **JadeMQ** transport, a call to the **putMessage** method is blocked until the memory in use falls below this limit. (You can specify a **Timeout** value as part of the **options** parameter to the **putMessage** method, to limit the delay.)

In the **JadeMQ** transport, the default value of the **maxMemoryInuse** property is 10M bytes and the maximum value is 10M bytes. In the WebSphere MQ transport, the value of the **maxMemoryInuse** property is the maximum integer value (**2,147,483,647**), and attempts to change this value are ignored.

maxMessageCount

Type: Integer

The **maxMessageCount** property of the **JadeGenericQueue** class contains the maximum number of messages that can be present in the queue.

The behavior depends on the transport. In the **JadeMQ** transport, a call to the **putMessage** method is blocked until the number of messages in the queue falls below this limit. (You can specify a **Timeout** value as part of the **options** parameter to the **putMessage** method, to limit the delay.)

In the **JadeMQ** transport, the default value of the **maxMessageCount** property is **100** and the maximum value is **1,000**. In the WebSphere MQ transport, the value of the **maxMessageCount** property is **999,999,999**, and attempts to change this value are ignored.

maxMessageLength

Type: Integer

The **maxMessageLength** property of the **JadeGenericQueue** class contains the length of the longest message that can be put into this queue. An exception is raised if a call to the **putMessage** method attempts to put a message in the queue that exceeds this value.

Some transports include properties in addition to the body of the message when calculating the message length. You can disable the automatic segmentation of a message, by setting the value of the **maxMessageLength** property to the value returned by the **getMaxSegmentLength** method.

In the **JadeMQ** transport, the default value of the **maxMessageLength** property is 47M bytes, which is also the maximum value. In the WebSphere MQ transport, the default value of the **maxMessageLength** property is 50M bytes and the maximum value is 95M bytes.

name

Type: String[120]

The read-only **name** property of the **JadeGenericQueue** class contains the name of the queue. Unlike the **fullName** property, it is not qualified by the transport type and the queue manager name.

tag

Type: String

The **tag** property of the **JadeGenericQueue** class contains a value that is stored with the message object. Unlike other properties, the value of the **tag** property is not used by JADE and is available for you to use for any purpose in your application.

By default, the **tag** property is set to a null string (**""**).

JadeGenericQueue Methods

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

Method	Description
beginMessage	Prepares a specified message for segmented transmission and links it with a destination queue
cancelNotify	Deregisters the object that received callback notifications for the arrival of messages in a queue
close	Unlinks the receiver from the actual message queue so that the queue is no longer accessible
countQueuedMessages	Returns the number of messages present in the queue
countReceivers	Returns the number of processes that currently have the queue opened for retrieval of messages
createMessage	Returns a message object that can be added to or retrieved from a queue
getMaxSegmentLength	Returns the maximum length of an individual segment of a message
getMessage	Retrieves a message from the queue
getMessageByCorrelationID	Retrieves a message from the queue with a specified **correlationID** value
getQueueManager	Returns a reference to the queue manager object that manages the queue
getQueueProperty	Returns the value of the specified property of the queue object
getTransportName	Returns the name of the transport associated with the queue
isOpen	Returns **true** if the queue has not been closed
isRemote	Returns **true** if the queue is owned by another queue manager
notifyEventsAsync	Registers an object for callback notifications for the arrival of messages in a queue
purge	Discards messages that have not been retrieved and returns the number discarded
putMessage	Adds the specified message into the queue
setQueueProperty	Sets the specified property of the queue object to the specified value

beginMessage

Signature `beginMessage(msg: JadeGenericMessage input; options: String);`

The **beginMessage** method of the **JadeGenericQueue** class prepares a message specified by the value of the **msg** parameter for segmented transmission and links it with the destination queue.

The **initializeForPut** method must be invoked for the **msg** parameter either explicitly, or implicitly by calling the **createMessage** method with the value of the **forPUT** parameter set to **true**.

After the **beginMessage** method is called, data can be added to the body of the message by using the **appendBodyTuple** method only, or by repeatedly using any of the **appendBinary**, **appendString**, **appendStringAsUtf8**, or **appendStringUtf8** methods. The message is completed by calling the **putMessage** method, after which calls to an **append** method results in an exception being raised.

The message can begin transmitting before the **putMessage** method is invoked, because the **beginMessage** method has previously associated the message with the destination queue. This allows some overlap between message assembly and message transmission for large messages, although this behavior depends on the transport. After the **beginMessage** method is executed, you cannot update any properties of the message.

The **putMessage** method must be called on the same queue as the **beginMessage** method, to allow the message to become visible to receiving applications.

If the **JadeGenericMessage** object referenced by the **msg** parameter is deleted or is initialized by invoking the **initializeForGet** or **initializeForPut** method, the incomplete message is discarded, including any segments already transmitted.

If the **options** parameter is a null string, transport-specific default actions are taken. Options passed to this method override any specified in the **defaultPutMessageOptions** property. Components that can be included in the **options** parameter string for the **JadeMQ** transport are listed in the following table. (These options are ignored for the WebSphere MQ transport.)

Component	Result
NoWait	The method returns immediately if the queue is full
Timeout=*milliseconds*	The method waits for the specified number of milliseconds until there is enough room in the queue for a message to be sent
Timeout=0	The method waits indefinitely (the default implied option)

The following example shows the use of the **beginMessage** method.

```
vars
    msg : JadeGenericMessage;
begin
    msg := myQueue.createMessage(true);
    myQueue.beginMessage(msg, null);
    msg.appendString("first entry");
    // build the rest of the message
    msg.appendString("last entry");
    myQueue.putMessage(msg, null);
end;
```

cancelNotify

Signature `cancelNotify();`

The **cancelNotify** method of the **JadeGenericQueue** class deregisters the object that was registered to receive the following callback notifications by the **notifyEventsAsync** method.

- A message arrives in the receiver queue (the event callback method is **messageArrivedEvent**)
- The receiver queue is deleted (the event callback method is **managementEvent**)

close

Signature `close(options: String);`

The **close** method of the **JadeGenericQueue** class unlinks the receiver (the queue object) from the actual message queue so that the actual message queue is no longer accessible. The queue should then be deleted. A queue is closed implicitly when it is deleted.

The actual message queue may still exist after it is closed, depending on the transport. If the actual message queue still exists, the **options** parameter enables you to specify the fate of these messages.

countQueuedMessages

Signature `countQueuedMessages(): Integer;`

The **countQueuedMessages** method of the **JadeGenericQueue** class returns the number of messages present in the queue.

The queue must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Inq** or **Usage=All** included in the **options** parameter.

countReceivers

Signature `countReceivers(): Integer;`

The **countReceivers** method of the **JadeGenericQueue** class returns the number of processes that currently have the queue opened for retrieval of messages.

The queue must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Inq** or **Usage=All** included in the **options** parameter.

createMessage

Signature `createMessage(forPUT: Boolean): JadeGenericMessage;`

The **createMessage** method of the **JadeGenericQueue** class returns a **JadeGenericMessage** object that can be used for working with a message that is added to or retrieved from a queue. If the value of the **forPUT** parameter is **true**, the message object is initialized for constructing a message to be added to a queue. If the value of the **forPUT** parameter is **false**, the message object is initialized for retrieving a message from a queue.

The following example shows the use of the **createMessage** method to retrieve a message.

```
vars
    factory : JadeMessagingFactory;
    msg : JadeGenericMessage;
    queue : JadeGenericQueue;
```

```
begin
    create factory transient;
    queue := factory.openQueue("JadeMQ://localnode/TestQ", "Usage=All");
    msg := queue.createMessage(false);
    queue.getMessage(msg, null);
    write msg.body.String;
epilog
    delete factory;
    delete msg;
    delete queue;
end;
```

getMaxSegmentLength

Signature `getMaxSegmentLength(): Integer;`

The **getMaxSegmentLength** method of the **JadeGenericQueue** class returns the maximum length of an individual segment of a message. In the **JadeMQ** transport, the maximum segment size is **1,000,000** bytes. In the WebSphere MQ transport, the maximum segment size that JADE supports is **4,194,128** bytes.

getMessage

Signature
```
getMessage(msg:     JadeGenericMessage input;
           options: String): Boolean;
```

The **getMessage** method of the **JadeGenericQueue** class retrieves a message from the queue and copies its contents into the message object passed into the method and specified by the **msg** parameter. If there are no messages in the queue, the method returns **false**.

The **JadeGenericQueue** instance that created the **msg** message object (using the **createMessage** method) need not be the same instance that retrieves the message with the **getMessage** method, but both **JadeGenericQueue** instances must use the same transport.

If the **msg** object has previously been used to put an object in a queue by using the **putMessage** method or to retrieve a message from a queue by using the **getMessage** or **getMessageByCorrelationID** method, it must be initialized by using the **initializeForGet** method defined in the **JadeGenericMessage** class.

If the **options** parameter is a null string, transport-specific default actions are taken. Options passed to this method override any specified in the **defaultPutMessageOptions** property.

Components that can be included in the **options** parameter string for the **JadeMQ** transport are listed in the following table. (These options are ignored for the WebSphere MQ transport.)

Component	Result
NoWait	The method returns immediately if no message is available
Timeout=*milliseconds*	The method waits for the specified number of milliseconds for a message to arrive
Timeout=0	The method waits indefinitely (the default implied option)

The queue must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Get** or **Usage=All** included in the **options** parameter.

The following example shows the use of the **getMessage** method to retrieve a message from a queue.

```
vars
    factory : JadeMessagingFactory;
    msg : JadeGenericMessage;
    queue : JadeGenericQueue;
begin
    create factory transient;
    queue := factory.openQueue("JadeMQ://localnode/TestQ", "Usage=All");
    msg := queue.createMessage(false);
    queue.getMessage(msg, "Timeout=5000");
    write msg.body.String;
epilog
    delete factory;
    delete msg;
    delete queue;
end;
```

getMessageByCorrelationID

Signature `getMessageByCorrelationID(msg: JadeGenericMessage input; options: String; correlIdList: Any): Boolean`

The **getMessageByCorrelationID** method of the **JadeGenericQueue** class retrieves a message from the queue when the value of the **correlationID** property of the message matches one of the elements of the **correlIdList** parameter.

The contents of the message are copied into the message object passed into the method and specified by the **msg** parameter. If there are no messages with the specified **correlationID** value in the queue, the method returns **false**.

The value of the **correlIdList** parameter is a single **Binary** value or a **BinaryArray** value. In WebSphere MQ, the **correlIdList** parameter can be a single **Binary** value only, which should be stored in a maximum length variable or property, as trailing null characters are significant. If an empty **Binary** or **null** is passed, the method behaves identically to the **getMessage** method and the first available message is returned. A **BinaryArray** list must not contain more than 64 entries.

Note Do not use the **getMessageByCorrelationID** method if you have registered a receiver for messages by using the **notifyEventsAsync** method.

When a message arrives, you must handle it in the **messageArrivedEvent** event method by using the **getMessage** method and not the **getMessageByCorrelationID** method. If you do not do this, the **messageArrivedEvent** is triggered again, because the queue is not empty, and the processing loops.

The **JadeGenericQueue** instance that created the **msg** message object (using the **createMessage** method) need not be the same instance that retrieves the message with the **getMessageByCorrelationID** method, but both **JadeGenericQueue** instances must use the same transport. If the **msg** object has previously been used to put an object in a queue by using the **putMessage** method or to retrieve a message from a queue by using the **getMessage** or **getMessageByCorrelationID** method, it must be initialized by using the **initializeForGet** method defined in the **JadeGenericMessage** class.

If the **options** parameter is an empty string, transport-specific default actions are taken. Options passed to this method override any specified in the **defaultPutMessageOptions** property. Components that can be included in the **options** parameter string for the **JadeMQ** transport are listed in the following table. (These options are ignored for the WebSphere MQ transport.)

Component	Result
NoWait	The method returns immediately if no message is available
Timeout=*milliseconds*	The method waits for the specified number of milliseconds for a message to arrive
Timeout=0	The method waits indefinitely (the default implied option)

The queue must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Get** or **Usage=All** included in the **options** parameter.

The following example shows the use of the **getMessageByCorrelationID** method to retrieve a message from a queue.

```
vars
    factory : JadeMessagingFactory;
    msg : JadeGenericMessage;
    requestQ : JadeGenericQueue;
    replyQ : JadeGenericQueue;
    id : Binary;
begin
    create factory transient;
    requestQ := factory.openQueue("JadeMQ://localnode/RequestQ",
                                                          "Usage=Put");
    replyQ := factory.openQueue("JadeMQ://localnode/ReplyQ", "Usage=Get");
    msg := requestQ.createMessage(true);
    msg.replyQueueFullName := "JadeMQ://localnode/ReplyQ";
    msg.type := JadeGenericMessage.Type_Request;
    msg.appendString("Please reply to this request");
    requestQ.putMessage(msg, null);
    id := msg.messageID;     // To be used as the correlationID in the reply
    msg.initializeForGet;
    replyQ.getMessageByCorrelationID(msg, null, id);
    write msg.body.String;
epilog
    delete factory;
    delete msg;
    delete requestQ;
    delete replyQ;
end;
```

getQueueManager

Signature `getQueueManager(): JadeGenericQueueManager;`

The **getQueueManager** method of the **JadeGenericQueue** class returns a reference to the **JadeGenericQueueManager** object that manages the receiver queue.

getQueueProperty

Signature `getQueueProperty(propname: String; value: Any output): Boolean;`

The **getQueueProperty** method of the **JadeGenericQueue** class returns the value of the property specified in the **propname** parameter. The result is returned to the variable passed as the **value** parameter and must be compatible with the type of that variable. A variable of type **Any** is compatible with all property types.

If the value of the **propname** parameter is not a valid property name, an exception is raised.

Note As the **getQueueProperty** method does not perform type checking when the method is compiled, you should directly access the property value where possible.

getTransportName

Signature `getTransportName(): String;`

The **getTransportName** method of the **JadeGenericQueue** class returns the name of the transport associated with the queue.

isOpen

Signature `isOpen(): Boolean;`

The **isOpen** method of the **JadeGenericQueue** class returns **true** if the queue has not been closed.

isRemote

Signature `isRemote(): Boolean;`

The **isRemote** method of the **JadeGenericQueue** class returns **true** if the queue is owned by a queue manager other than the one to which the application is connected. It returns **false** if the queue is owned by the queue manager to which the application is connected.

An application accesses a remote queue through a local queue manager and may have a limited set of actions; for example, messages cannot be retrieved from a remote queue.

notifyEventsAsync

Signature `notifyEventsAsync(recvr: JadeGenericMessagingIF; option: Integer);`

The **notifyEventsAsync** method of the **JadeGenericQueue** class registers an object specified by the **recvr** parameter, which must implement the **JadeGenericMessagingIF** interface, for notifications through callback methods for the following events.

- A message arrives in the receiver queue (the event callback method is **messageArrivedEvent**)
- A management event occurs on the queue (the event callback method is **managementEvent**)

The parameters for the **notifyEventsAsync** method are listed in the following table.

Parameter	Description
recvr	Object that is to receive the callback notifications
option	The number of callback notifications, which can be specified using one of the following class constants. ■ **Notify_OneShot** (only the first event results in a callback notification, after which callbacks are canceled) ■ **Notify_Continuous** (every event results in a callback notification)

A process can register only one object to receive events for each queue. If the **notifyEventsAsync** method is called again, the new **recvr** object replaces the existing receiver of callback notifications.

To receive events for a queue, it must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Get or Usage=All** included in the **options** parameter.

The following example shows the use of the **notifyEventsAsync** method to register an object for callbacks.

```
vars
    factory : JadeMessagingFactory;
    queue : JadeGenericQueue;
begin
    create factory transient;
    queue := factory.openQueue("JadeMQ://localnode/TestQ", "Usage=All");
    queue.notifyEventsAsync(receiver, JadeGenenericQueue.Notify_Continuous);
epilog
    delete factory;
end;
```

purge

Signature `purge(options: String): Integer;`

The **purge** method of the **JadeGenericQueue** class discards all messages in the queue that have not been retrieved and returns the number of discarded messages.

Use this method to purge an existing queue of reply messages before issuing another request.

For the **options** parameter, no generic or transport-specific options have been defined, so you should pass a null string, as shown in the following code fragment.

```
myQueue.purge(null);
```

putMessage

Signature
```
putMessage(msg:     JadeGenericMessage input;
           options: String);
```

The **putMessage** method of the **JadeGenericQueue** class adds the message specified by the **msg** parameter to the queue. After executing the **putMessage** method, you can delete the **msg** object or initialize it for sending or receiving another message from the queue.

The method updates properties of the **msg** object such as the **messageID** and **createdWhen** properties.

The **JadeGenericQueue** instance that created the **msg** message object (using the **createMessage** method) need not be the same instance that sends the message with the **putMessage** method, but both **JadeGenericQueue** instances must use the same transport. If the **msg** object has previously been used to add a message to a queue by using the **putMessage** method or to retrieve a message from a queue by using the **getMessage** or **getMessageByCorrelationID** method, it must be initialized using the **initializeForPut** method defined in the **JadeGenericMessage** class.

If the **options** parameter is a null string, transport-specific default actions are taken. Options passed to this method override any specified in the **defaultPutMessageOptions** property.

Components that can be included in the **options** parameter string for the **JadeMQ** transport are listed in the following table. (These options are ignored for the WebSphere MQ transport.)

Component	Result
NoWait	The method returns immediately if the queue is full
Timeout=*milliseconds*	The method waits for the specified number of milliseconds until there is enough room in the queue for a message to be sent
Timeout=0	The method waits indefinitely (the default implied option)

The queue must be opened by the **openQueue** method defined in the **JadeMessagingFactory** class, with **Usage=Put** or **Usage=All** included in the **options** parameter.

The following example shows the use of the **putMessage** method to send a message to a queue.

```
vars
    factory : JadeMessagingFactory;
    msg : JadeGenericMessage;
    queue : JadeGenericQueue;
begin
    create factory transient;
    queue := factory.openQueue("JadeMQ://localnode/TestQ", "Usage=Put");
    msg := queue.createMessage(true);
    queue.getMessage(msg, "Timeout=5000");
epilog
    delete factory;
    delete msg;
    delete queue;
end;
```

setQueueProperty

Signature `setQueueProperty(propname: String;`
`value: Any);`

The **setQueueProperty** method of the **JadeGenericQueue** class sets a user-defined property of the receiver specified in the **propname** parameter to the value specified in the **value** parameter.

If the property specified in the **propname** parameter is invalid, an exception is raised.

Note You should not use the **setQueueProperty** 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.

JadeGenericQueueManager Class

The transient **JadeGenericQueueManager** class is part of the JADE messaging framework. It encapsulates the management of a single messaging queue.

A queue manager object is automatically created and associated with a queue when the **openQueue** method of the **JadeMessagingFactory** class is executed. The **getQueueManager** method of the **JadeGenericQueue** class returns a reference to the queue manager object for a queue. The queue manager object is automatically deleted when you delete the associated queue object. Unless you need to access a property or method of a queue manager object, you can ignore it in your coding.

For details about the use of the JADE messaging framework, see Chapter 15, "Using the Messaging Framework", in the *JADE Developer's Reference*. For details about the property and methods defined in the **JadeGenericQueueManager** class, see "JadeGenericQueueManager Property" and "JadeGenericQueueManager Methods", in the following subsections. For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the *JADE Developer's Reference*.

Inherits From: Object

Inherited By: (None)

JadeGenericQueueManager Property

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

Property	Description
name	Simple name of queue manager object

name

Type: String[60]

The read-only **name** property of the **JadeGenericQueueManager** class is the name of the queue manager. A name is automatically generated when the queue manager object is created by calling the **openQueue** method of the **JadeMessagingFactory** class.

If you decide to change the name, select a value that is unique within a machine and which is unique globally.

JadeGenericQueueManager Methods

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

Method	Description
getFullName	Returns a string containing the transport name, the queue manager name, and the network address
getQueueManagerProperty	Returns the value of the specified property
getTransportName	Returns the name of the transport associated with the queue manager
setQueueManagerProperty	Sets the property of the receiver to the specified value

getFullName

Signature `getFullName(): String;`

The **getFullName** method of the **JadeGenericQueueManager** class returns a string that includes the transport name, the name of the queue manager (that is, the value of the **name** property), and the network address.

getQueueManagerProperty

Signature
```
getQueueManagerProperty(propname: String;
                        value:    Any output): Boolean;
```

The **getQueueManagerProperty** method of the **JadeGenericQueueManager** class assigns the value of the property specified in the **propname** parameter to the **value** parameter.

The method returns **false** if the property name specified in the **propname** parameter is invalid.

Note As the **getQueueManagerProperty** method does not perform type checking when the method is compiled, you should directly access the property value, where possible.

getTransportName

Signature `getTransportName(): String;`

The **getTransportName** method of the **JadeGenericQueueManager** class returns the name of the transport associated with the queue manager.

setQueueManagerProperty

Signature
```
setQueueManagerProperty(propname: String;
                        value:    Any);
```

The **setQueueManagerProperty** method of the **JadeGenericQueueManager** class sets the property of the receiver specified in the **propname** parameter to the value specified in the **value** parameter.

If the property specified in the **propname** parameter is invalid, an exception is raised.

Note As the **setQueueManagerProperty** method does not perform type checking when the method is compiled, you should directly assign a value to a property, where possible.

JadeHTMLClass Class

The **JadeHTMLClass** class implements the behavior required to support HTML pages in your JADE applications. A subclass of the **JadeHTMLClass** class is automatically created in your schema when you create the first HTML page for the current schema by using the HTML Wizard in the JADE development environment. (For details about the support in JADE of HTML documents, see Chapter 12 of the *JADE Development Environment User's Guide*.)

If you want full control of the HTML pages in your applications, you can then reimplement the appropriate **JadeHTMLClass** methods to suit your requirements by reimplementing the **generateHTMLString**, **processRequest**, **reply**, and **updateValues** methods for every **JadeHTMLClass** subclass.

The connection to the Web browser can be a **TcpIpConnection** or a **NamedPipe** connection.

Notes Use **JadeHTMLClass** methods such as the **addOptionTag** method to assist you in the construction of correctly formatted HTML to use for substitution into the appropriate **JADE_TAG** tags within your **JadeHTMLClass** subclasses.

Although you can insert **JADE_TAG** tags within **<select>** and **</select>** tags, the value of the property associated with the **JADE_TAG** should contain only **<option>** and **</option>** tags. Inserting any other tag may cause unpredictable behavior (browser-dependent).

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

Inherits From: Object

Inherited By: (None)

JadeHTMLClass Properties

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

Property	Contains …
generateHTMLForJadeTagsOnly	Whether **JADE_TAG** tags are the only ones processed when HTML is generated
goToPage	The next page to send to the Web browser
httpString	The request string from the Web browser
securePage	The value of the page security status

generateHTMLForJadeTagsOnly

Type: Boolean

The **generateHTMLForJadeTagsOnly** property of the **JadeHTMLClass** class specifies whether the **JADE_TAG** tags are the only ones that are processed when the HTML is generated for a page. When this property is set to **true**, all other tags are not altered in any way. The default for this property is **false**, which will process all tags.

For more details, see "Generating Data for JADE_TAG Tags Only", in Chapter 12 of the *JADE Development Environment User's Guide*.

goToPage

Type: HTMLClass

The **goToPage** property of the **JadeHTMLClass** class contains the next page to send to the Web browser. Set this property to a **JadeHTMLClass** subclass.

httpString

Type: String

The **httpString** property of the **JadeHTMLClass** class contains the request string from the Web browser.

securePage

Type: String

The **securePage** property of the **JadeHTMLClass** class specifies whether the HTML page is secure. This property enables you to dynamically change the secure status of the page.

The default value is defined for the document when using the HTML Wizard. (For details, see "Using the HTML Wizard to Add an HTML Document", in Chapter 12 of the *JADE Development Environment User's Guide*.)

JadeHTMLClass Methods

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

Method	Description
addCookie	Adds cookie information on the client node for the HTML page when the HTML string is generated
addHiddenField	Dynamically adds hidden input tags to the HTML page
addOptionTag	Returns the HTML representing an option tag for a **<select>** tag
addPageBounce	Returns a string containing page bounce code
buildFormActionOnly	Returns the action attribute for a form tag
buildLink	Generates a hyperlink
clearCookies	Removes all cookies from the HTML page
generateHTMLString	Generates and returns the HTML string based on the information saved when the page was imported
getAttributes	Returns the attributes that have been set up for a specified property on the HTML page
getCookies	Returns all of the cookies set for the current session
getHttpValue	Processes the **httpString** property value and returns the value for the specified name
processRequest	Processes the input from the Web browser and sets the properties to the appropriate values
queryIncludePage	Determines if the included page is inserted into the HTML string
reply	Sends the response back to the Web browser

Method	Description
replyAsBinary	Returns the Binary message to the Web browser without modification
setAttributes	Allows additional attributes to be set on an input tag prior to the HTML generation
tableAddData	Returns a **<td>** tag with attributes and data set appropriately
tableAddInput	Returns a **<td>** tag with attributes, input type, and values set appropriately
tableAddItem	Sets up data for a table, with all data for a table row specified in the **value** string
tableBegin	Returns a string containing the opening tag for a table
tableBeginRow	Returns a string containing the HTML for the row tag
tableEnd	Returns a string containing the HTML for the end table tag
tableEndRow	Returns a string containing the tag for the end of a table row
updateValues	When reimplemented, sets up the required property values
wasPageBounced	Specifies whether the HTML page was bounced to another Web page

addCookie

Signature

```
addCookie(cookieName:  String;
          cookieValue: String;
          cookieLife:  Decimal;
          cookiePath:  String);
```

The **addCookie** method of the **JadeHTMLClass** class adds cookie information on the client node for the HTML page when the HTML string is generated.

The parameters for the **addCookie** method are listed in the following table.

Parameter	Description
cookieName	Name of the cookie.
cookieValue	Value of the cookie. If this is null (**""**), the **deleteCookie** function is generated; otherwise the **setCookie** function is generated.
cookieLife	Number of minutes that represents the lifetime of the cookie.
cookiePath	Path of the cookie.

The code fragment in the following example shows the use of the **addCookie** method.

```
addCookie("userName", "Wilbur", 120, "");
```

addHiddenField

Signature `addHiddenField(hiddenFieldName: String; hiddenFieldValue: String) updating;`

The **addHiddenField** method of the **JadeHTMLClass** class dynamically adds hidden input tags to the HTML page. The parameters for the **addHiddenField** method are listed in the following table.

Parameter	Description
hiddenFieldName	Name of the hidden field
hiddenFieldValue	Value for the hidden field, which is the value that is returned by the Web browser unless it is changed by script

The code fragment in the following example shows the use of the **addHiddenField** method.

```
addHiddenField("mySession", "2003");
```

This example adds the following to the HTML page.

```
<input type "hidden" name = "mySession" value = "2003">
```

addOptionTag

Signature `addOptionTag(opt: String; selected: Boolean): String;`

The **addOptionTag** method of the **JadeHTMLClass** class returns the HTML representing an option tag for a **<select>** tag. No validation is done to ensure that the **<select>** tag already exists. The parameters for the **addOptionTag** method are listed in the following table.

Parameter	Description
opt	String to be added to the **<select>** tag
selected	Set to **true** when this entry is the selected entry

The code fragment in the following example shows the use of the **addOptionTag** method.

```
addOptionTag("Denniston", true);
```

This example returns the following.

```
<option selected>Denniston </option>
```

addPageBounce

Signature `addPageBounce(bounceSeconds: Integer; bounceURL: String): String;`

The **addPageBounce** method of the **JadeHTMLClass** class returns a string containing page bounce code. The parameters for the **addPageBounce** method are listed in the following table.

Parameter	Description
bounceSeconds	Number of seconds to wait before bouncing
bounceURL	URL of the Web page to which to bounce if there is no user action within the specified number of seconds

The code fragment in the following example shows the use of the **addPageBounce** method.

```
addPageBounce(300, "www.jadeworld.com");
```

This example returns the following.

```
<meta http-equiv = "refresh" content = "300;url=www.jadeworld.com">
```

buildFormActionOnly

Signature `buildFormActionOnly(): String;`

The **buildFormActionOnly** method of the **JadeHTMLClass** class returns the action attribute for a form tag. The URL is built by using the secure page setting and user settings for the machine name and virtual directory.

The code fragment in the following example shows the use of the **buildFormActionOnly** method.

```
str := buildFormActionOnly() & " method = post";
```

This example returns the following.

```
http:/wilbur1a/jade/jadehttp.dll?WebApp method = post
```

In this returned value, **wilbur1a** is the machine name, **jade** is the virtual directory, **WebApp** is the application name, and the page has been defined as non-secure. (See also the **Application** class **setWebMachineName** and **setWebVirtualDirectory** methods.)

buildLink

Signature `buildLink(nextPage: Class): String;`

The **buildLink** method of the **JadeHTMLClass** class generates a hyperlink for the HTML page. The URL is built up by using the secure page setting, user settings for the machine name and virtual directory, and the session id. If the value of the **nextPage** parameter is not null, another parameter (using the value of the **goToPage** property) is generated, which sets up the next page to be displayed on the Web browser.

The code fragment in the following example shows the use of the **buildLink** method.

```
buildLink(Customer);     // Customer is a subclass of JadeHTMLClass class
```

This example returns the following.

```
http:/wilbur1a/jade/jadehttp.dll?WebApp &_jadeReferenceClass=Menu &_
jadeReferenceDocument=MenuPage &_session=123456789abcdef &_goToPage=Customer
```

The returned values in this example are listed in the following table.

Value	Description
wilbur1a	Machine name
jade	Virtual directory
WebApp	Application name (session information)
_jadeReferenceClass	Menu is **self** (session information)
_jadeReferenceDocument	MenuPage, which is the corresponding HTML document (session information)
_session	Session identifier (session information)
_goToPage (optional)	Page that is returned when this link is clicked

See also the **Application** class **setWebMachineName** and **setWebVirtualDirectory** methods.

clearCookies

Signature `clearCookies() updating;`

The **clearCookies** method of the **JadeHTMLClass** class clears all cookies that were added to the HTML page using the **addCookie** method.

generateHTMLString

Signature `generateHTMLString(): String updating;`

The **generateHTMLString** method of the **JadeHTMLClass** class generates and returns the HTML string. The original HTML string is updated with the following information before it is returned.

- Property values of the receiver
- Uniform Resource Locator (URL) changes
- HyperText Transfer Protocol (HTTP) or secure HyperText Transfer Protocol (HTTPS)
- Any attributes set by the **setAttributes** method

In the following example, the HTML for the page contained multi-byte characters that were converted to UTF-8 when the page was imported. The reimplemented **generateHTMLString** method replaces the standard HTML header to inform browsers that the body is in UTF-8 format.

```
vars
    message, header: String;
begin
    message := inheritMethod();
    header  := 'Content-Type: text/html;charset=UTF-8' & CrLf & CrLf;
    return replyAsBinary(header, message.Binary);
end;
```

getAttributes

Signature `getAttributes(jadeName: String): String;`

The **getAttributes** method of the **JadeHTMLClass** class returns the attributes that have been set up for the property specified in the **jadeName** parameter.

getCookies

Signature `getCookies(): StringArray;`

The **getCookies** method of the **JadeHTMLClass** class returns a string array containing all cookies set for the current session. The format of the strings in the array is as follows.

```
<cookie-name>,<cookie-value>
```

getHttpValue

Signature `getHttpValue(name: String): String;`

The **getHttpValue** method of the **JadeHTMLClass** class processes the **httpString** property value and returns the value of the string specified in the **name** parameter with the HTTP character encoding removed from it; that is, any **"+"** characters are changed to space characters and any hexadecimal-encoded characters (**"%xx"**) are decoded to the original characters. For example, if the value of the **httpString** property is **"...&inventor=Wilbur+%26+Orville+Wright&..."**, **getHttpValue("inventor")** returns **"Wilbur & Orville Wright"**.

processRequest

Signature `processRequest(): Boolean updating;`

When a request is received from the Web browser, a transient instance of the class corresponding to this request is created, and the **processRequest** method of the **JadeHTMLClass** class is called to process the property values of the instance from the request, setting up the properties with the appropriate values.

When a request is received by an HTML document-based application, the **processRequest** method is not called if it is the initial request where the home page is invoked (it is called when additional request is made on that document; for example, a link is clicked) or if the reference tag in the received data is **_jadeReferencePage**. If the reference tag in the received message is **_jadeReferenceClass**, the **processRequest** method is called. Alternatively, you can reimplement the **updateValues** method to set up the required property values for the HTML page. (See also "HTML Document Implementation", in Chapter 12 of the *JADE Development Environment User's Guide*.)

The **processRequest** method returns **true** if the processing was successful or it returns **false** if there was an error in the processing.

queryIncludePage

Signature `queryIncludePage(): Boolean;`

Before an included page is inserted into the HTML string, the **queryIncludePage** method of the **JadeHTMLClass** class is called to determine if it should be included. This method must return **true** if the page is to be inserted into the HTML string or **false** if the page should be ignored. By default, the include page is inserted into the string.

Reimplement this method for conditional includes.

reply

Signature `reply(): String updating;`

The default implementation of the **reply** method of the **JadeHTMLClass** class calls the **generateHTMLString**, which calls the **updateValues** method and then generates the HTML with the updated values from the transient instance. It includes the class name and the HTML document name as hidden fields in the string.

In addition, the session id of the **currentSession** system variable is also included as a hidden field. If the page was defined as secure, all URLs on this page are prefixed with **https**.

The transient instance is then deleted, and the application then goes back to waiting for input.

replyAsBinary

Signature
```
replyAsBinary(header:  String;
              message: Binary): String;
```

The **replyAsBinary** method of the **JadeHTMLClass** class returns the Binary message contained in the **message** parameter to the Web browser without modification. Use this method to send a binary reply without UTF-8 encoding to a Web message.

The message uses the String value in the **header** parameter as the HTTP headers. The headers are encoded with UTF-8, as is expected for HTTP headers.

As this method is designed for use in the **generateHTMLString** method that returns an HTML string result, the **replyAsBinary** method returns a string. However, as the reply has already been sent when the result is returned, the **replyAsBinary** method returns a null string (**""**).

The following example shows a reimplemented **generateHTMLString** method using the **replyAsBinary** method to download a PDF file to a browser.

```
vars
    file : File;
    message : Binary;
    header : String;
begin
    create file transient;
    file.fileName := "C:\documentation\JadLoad.pdf";
    file.kind := File.Kind_Binary;
    message := file.readBinary(file.fileLength);
    header := 'HTTP/1.1 200 OK' & CrLf &
        'Content-Type: application/pdf' & CrLf &
        'Content-Disposition: attachment;filename=myfile.pdf' & CrLf &
        'Content-Length: ' & message.length.String & CrLf & CrLf;
    return replyAsBinary(header, message);
epilog
    delete file;
end;
```

setAttributes

Signature
```
setAttributes(propertyName:   String;
              attributeName:  String;
              attributeValue: String): Boolean;
```

The **setAttributes** method of the **JadeHTMLClass** class allows additional attributes to be set on an input tag prior to the HTML generation.

This method returns **true** if the property specified in the **propertyName** parameter exists or it returns **false** if it does not. For example, if you want to change the color of a **SUBMIT** button (**name=okButton**) dependent on a condition, you could code the following.

```
if customer.specialCustomer then
    setAttributes('okButton', 'style', "background-color:Red";
else
    setAttributes('okButton', 'style', "background-color:Blue";
endif;
```

If the original HTML for this was **<input type=submit name="okButton" value="OK">**, the generated HTML code for this would be as follows (assuming the condition to be true).

```
<input type=submit name="okButton" value="OK" style="background-color:Red">
```

Note The **style** command can be ignored in Web browsers that do not fully support Cascading Style Sheets.

The parameters for the **setAttributes** method are listed in the following table.

Parameter	Description
propertyName	Name of the property whose attributes are to be set
attributeName	Name of the attribute to set
attributeValue	Value of the attribute

tableAddData

Signature
```
tableAddData(attr: String;
             data: String): String;
```

The **tableAddData** method of the **JadeHTMLClass** class returns a string containing the **<td>** tag attributes and data that are set by the **attr** and **data** parameters.

The parameters in this method are listed in the following table.

Parameter	Description
attr	**<td>** tag attributes
data	**<td>** tag data

The code fragment in the following example shows the use of the **tableAddData** method.

```
tableAddData("align=right", "203.30");
```

This example returns the following.

```
<td align=right> 203.30 </td>
```

tableAddInput

Signature
```
tableAddInput(inputType: String;
              attr:      String;
              name:      String;
              value:     String): String;
```

The **tableAddInput** method of the **JadeHTMLClass** class returns a string containing the **<td>** tag input type and name, attribute, and value set by the **inputType**, **attr**, **name**, and **value** parameters. The parameters in this method are listed in the following table.

Parameter	Description
inputType	Any valid **<input>** tags (for example, text)
attr	**<td>** tag attributes
name	Name of the input type
value	Initial value for the input type

As there are no equivalent properties that correspond to these input tags, you must reimplement the **processRequest** method and use the **getHttpValue** method to obtain the value of these tags when the response is returned from the Web browser.

The code fragment in the following example shows the use of the **tableAddInput** method.

```
tableAddInput("text", "align=middle", "ghost_town", "Denniston");
```

This example returns the following.

```
<td align=middle><input type=text name=ghost_town value=Denniston></td>
```

tableAddItem

Signature
```
tableAddItem(value:       String;
             columnCount: Integer;
             rowAttr:     String;
             cellAttr:    String): String;
```

The **tableAddItem** method of the **JadeHTMLClass** class returns a string containing the **<tr>** and **<td>** tags and all data for a table row specified in the **value** parameter.

The parameters in the **tableAddItem** method are listed in the following table.

Parameter	Description
value	Values for the cells in a row, with each value separated by a tab character.
columnCount	Number of columns to create. If there are insufficient values for the columns, the columns at the right of the table are set to blank.
rowAttr	Attributes for the table row.
cellAttr	Attributes for the table cells.

This method (which is a simpler means of setting up data for a table than the **tableAddData** method) generates the HTML required for a table row by using the **value** parameter to fill the table cells. Use the tab character (character code **09**) to separate multiple strings that you want inserted into each column of a newly added row.

The code fragment in the following example shows the use of the **tableAddItem** method.

```
tableAddItem("One" & Tab & "Two" & Tab & "Three", 3, "height=15",
"align=right");
```

This example returns the following.

```
<tr height =  15>
<td align = right> One </td>
<td align = right> Two </td>
<td align = right> Three </td>
</tr>
```

tableBegin

Signature `tableBegin(attr: String): String;`

The **tableBegin** method of the **JadeHTMLClass** class returns a string containing the opening tag for the table, specified in the **attr** parameter.

The code fragment in the following example shows the use of the **tableBegin** method.

```
tableBegin("width=100%");
```

This example returns the following.

```
<table width=100%>
```

tableBeginRow

Signature `tableBeginRow(attr: String): String;`

The **tableBeginRow** method of the **JadeHTMLClass** class returns a string containing the HTML of the row tag for the table, specified in the **attr** parameter.

The code fragment in the following example shows the use of the **tableBeginRow** method.

```
tableBeginRow("height=20");
```

This example returns the following.

```
<tr height=20>
```

tableEnd

Signature `tableEnd(): String;`

The **tableEnd** method of the **JadeHTMLClass** class returns a string representing the HTML for the tag for the end of the table. This method returns **</table>**.

tableEndRow

Signature `tableEndRow(): String;`

The **tableEndRow** method of the **JadeHTMLClass** class returns a string representing the tag for the end of the table row. This method returns **</tr>**.

updateValues

Signature `updateValues(): Boolean updating;`

Reimplement the **updateValues** method of the **JadeHTMLClass** class to set up the required property values for the HTML page.

wasPageBounced

Signature `wasPageBounced(): Boolean;`

The **wasPageBounced** method of the **JadeHTMLClass** class specifies whether the HTML page was bounced to another Web page if there was no user action within the number of seconds specified in the **addPageBounce** method.

JadeHTTPConnection Class

The **JadeHTTPConnection** class enables applications to access the standard Internet protocol HTTP. For ease of use, this class abstracts this protocol into a high-level interface. The underlying implementation uses the Microsoft default WinHTTP library or the WinINet library (using the respective **EnableWinHTTP** or **EnableWinINET** parameter in the [JadeEnvironment] section of the JADE initialization file).

A basic understanding of the HTTP protocol is required to use this feature. The two Requests for Comments (RFCs) that define this protocol are:

- RFC 1945, Hypertext Transfer Protocol - HTTP/1.0
- RFC 2616, Hypertext Transfer Protocol - HTTP/1.1

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

Inherits From: Object

Inherited By: (None)

JadeHTTPConnection Class Constants

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

Name	Type and Value	Description
Default_FTP	Integer = 21	Default FTP port
DefaultPort_HTTP	Integer = 80	Default HTTP port
DefaultPort_HTTPS	Integer = 443	Default HTTPS port
DefaultPort_Invalid	Integer = -1	Default invalid port
DefaultPort_Socks	Integer = 1080	Default SOCKS port
HeaderType_Client	Integer = 1	Client HTTP header type
HeaderType_Server	Integer = 2	Server HTTP header type
Header_Accept	String = "Accept"	Accept header
Header_ContentLength	String = "Content-Length"	Content-length header
Header_ContentType	String = "Content-Type"	Content-type header
Header_Host	String = "Host"	Host header
Header_KeepAlive	String = "Proxy-Connection"	Proxy-connection header
Header_Protocol	String = "Protocol"	Protocol header
Header_ReasonPhrase	String = "Reason-Phrase"	Reason-phrase header
Header_Referer	String = "Referer"	Referer header
Header_SoapAction	String = "SOAPAction"	SOAPAction header
Header_StatusCode	String = "Status-Code"	Status-code header

Name	Type and Value	Description
Header_UserAgent	String = "User-Agent"	User-agent header
ProtocolVersion_1_0	String = "HTTP/1.0"	HTTP Protocol 1.0
ProtocolVersion_1_1	String = "HTTP/1."	HTTP Protocol 1.1
ProxyConfig_Direct	Integer = 1	Resolves all host names locally
ProxyConfig_Preconfig	Integer = 0	Retrieves the proxy or direct configuration from the registry
ProxyConfig_Preconfig_NoAuto	Integer = 4	Retrieves the proxy or direct configuration from the registry and prevents the use of a start-up Microsoft JScript or **wpad.dat** file
ProxyConfig_Proxy	Integer = 3	Passes requests to the proxy
Scheme_DIRECT	String = "jadehttp.tcp"	JADE Direct scheme
Scheme_DIRECT_6_2	String = "jadehttp.tcp2"	JADE 6.2 Direct scheme
Scheme_DIRECT_6_2_TcpIpV4	String = "jadehttp.tcp2v4"	JADE 6.2 TcpIPv4 Direct scheme
Scheme_DIRECT_6_2_TcpIpV6	String = "jadehttp.tcp2v6"	JADE 6.2 TcpIPv6 Direct scheme
Scheme_FILE	String = "file"	File scheme; not supported
Scheme_FTP	String = "ftp"	FTP scheme; not supported
Scheme_HTTP	String = "http"	HTTP scheme
Scheme_HTTPS	String = "https"	HTTPS scheme
State_Failure	Integer = 8	Failed HTTP connection state (for details, see the **state** property)
State_NeedConnection	Integer = 0	Not connected
State_NeedRequest	Integer = 1	Connected and waiting for request
State_NoData	Integer = 6	No data available; not supported
Verb_CONNECT	String = "CONNECT"	The CONNECT operation; not supported
Verb_DELETE	String = "DELETE"	The DELETE operation; not supported
Verb_GET	String = "GET"	The GET operation
Verb_HEAD	String = "HEAD"	The HEAD operation; not supported
Verb_OPTIONS	String = "OPTIONS"	The OPTIONS operation; not supported
Verb_POST	String = "POST"	The POST operation
Verb_PUT	String = "PUT"	The PUT operation; not supported
Verb_TRACE	String = "TRACE"	The TRACE operation; not supported

JadeHTTPConnection Properties

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

Property	Description
abs_path	Path portion of the **url** property
connectTimeout	Timeout value, in milliseconds, to use for server connection requests
fragment	Fragment portion of the **url** property
hostname	Name of the host to which to connect
httpVersion	Version of the HTTP protocol
password	Password portion of the **url** property
port	Port number to use when the connect operation is requested
protocolFamily	Protocol used by the TCP/IP connection
proxyConfig	Required type of access
proxyHostname	Name of the proxy server or servers to use when proxy access is specified
proxyPassword	Password for the proxy server authentication, if required
proxyUser	User name for the proxy server authentication, if required
query	Query portion of the **url** property
receiveTimeout	Timeout value, in milliseconds, to receive a response to a request
responseBody	Response message
responseHeaders	HTTP headers from the response message
scheme	HTTP scheme
sendTimeout	Timeout value, in milliseconds, to use for sending requests
state	State of the connection
url	URL of the page being requested
usePresentationClient	Uses WinHTTP (or WinINet) from the machine from which the presentation client is running
user	User portion of the **url** property value

abs_path

Type: String

The **abs_path** property of the **JadeHTTPConnection** class contains the path portion of the **url** property, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

connectTimeout

Type: Integer

The **connectTimeout** property of the **JadeHTTPConnection** class contains the timeout value, in milliseconds, to use for server connection requests.

If a connection request takes longer than the specified timeout value, the request is cancelled.

The initial value is **120,000** (that is, 120 seconds).

fragment

Type: String

The **fragment** property of the **JadeHTTPConnection** class contains the fragment portion of the **url** property, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

hostname

Type: String

The **hostname** property of the **JadeHTTPConnection** class contains the name of the host to which to connect. This string is passed when the connection is requested.

httpVersion

Type: String

The **httpVersion** property of the **JadeHTTPConnection** class contains the version of the HTTP protocol. Its value can be one of the following **JadeHTTPConnection** class constants.

- **ProtocolVersion_1_0**, for HTTP Version 1.09
- **ProtocolVersion_1_1**, for HTTP Version 1.1 (the default value)

password

Type: String

The **password** property of the **JadeHTTPConnection** class contains the password portion of the **url** property, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

port

Type: Integer

The **port** property of the **JadeHTTPConnection** class contains the port number to use when the connect operation is requested.

protocolFamily

Type: Integer

The **protocolFamily** property of the **JadeHTTPConnection** class contains the protocol used by the TCP/IP connection when the **scheme** property is set to a **Direct_** value.

Its value can be one of the following **JadeHTTPConnection** class constants.

- **Scheme_DIRECT_6_2_TcpIpV4**
- **Scheme_DIRECT_6_2_TcpIpV6**

proxyConfig

Type: Integer

The read-only **proxyConfig** property of the **JadeHTTPConnection** class contains the required type of access.

Its value can be one of the following **JadeHTTPConnection** class constants.

- **ProtocolConfig_Direct**, which resolves all host names locally, and does not use a proxy
- **ProxyConfig_PreConfig**, which retrieves the proxy or direct configuration from the registry (the default value)
- **ProxyConfig_PreConfig_NoAuto**, which retrieves the proxy or direct configuration from the registry and prevents the use of a start-up Microsoft JScript or **wpad.dat** file
- **ProxyConfig_Proxy**, which passes requests to the proxy

proxyHostname

Type: String

The read-only **proxyHostname** property of the **JadeHTTPConnection** class contains the name of the proxy server or servers to use when proxy access is specified by setting the value of the **proxyConfig** property to **ProxyConfig_Proxy**.

If the value of the **proxyConfig** property is not set to **ProxyConfig_Proxy**, the value of the **proxyHostname** property will be null.

proxyPassword

Type: String

The read-only **proxyPassword** property of the **JadeHTTPConnection** class contains the password for the proxy server authentication, if required. The value is set using the **configureProxy** method.

Some proxy servers require a user name and password before connecting to the target host.

proxyUser

Type: String

The **proxyUser** property of the **JadeHTTPConnection** class contains the user name for the proxy server authentication, if required. The value set using the **configureProxy** method.

Some proxy servers require a user name and password before connecting to the target host.

query

Type: String

The **query** property of the **JadeHTTPConnection** class contains the query portion of the **url** property, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

receiveTimeout

Type: Integer

The **receiveTimeout** property of the **JadeHTTPConnection** class contains the timeout value, in milliseconds, to receive a response to a request.

If a response takes longer than the specified timeout value, the request is cancelled.

The initial value is **120,000** (that is, 120 seconds).

responseBody

Type: Binary

The read-only **responseBody** property of the **JadeHTTPConnection** class contains the response message.

responseHeaders

Type: String

The read-only **responseHeaders** property of the **JadeHTTPConnection** class contains the HTTP headers from the response message.

scheme

Type: String

The **scheme** property of the **JadeHTTPConnection** class contains the HTTP scheme.

Its value can be one of the following **JadeHTTPConnection** class constants.

- **Scheme_DIRECT**, for direct JADE to JADE Web services earlier than JADE 6.2
- **Scheme_DIRECT_6_2**, for direct JADE to JADE Web services from JADE 6.2 or later
- **Scheme_DIRECT_6_2_TcpIpV4**, for direct **TcpIPv4** JADE to JADE Web services from JADE 6.2 or later
- **Scheme_DIRECT_6_2_TcpIpV6**, for direct **TcpIPv6** JADE to JADE Web services from JADE 6.2 or later
- **Scheme_FILE**; known scheme, but not supported by JADE
- **Scheme_FTP**; known scheme, but not supported by JADE
- **Scheme_HTTP**, for the HTTP scheme (the default value)
- **Scheme_HTTPS**, for the HTTPS scheme

sendTimeout

Type: Integer

The **sendTimeout** property of the **JadeHTTPConnection** class contains the timeout value, in milliseconds, to use for sending requests.

If sending a request takes longer than the specified timeout value, the send is cancelled.

The initial value is **120,000** (that is, 120 seconds).

state

Type: Integer

The read-only **state** property of the **JadeHTTPConnection** class contains the state of the connection. Its value can be one of the following **JadeHTTPConnection** class constants.

- **State_Failure**, indicating a failed HTTP connection state; for example:
  - Open failed
  - Send request headers failed
  - Invalid verb specified for send headers
  - WinINet authentication failure
  - WinINet set option failure
  - WinINet open/connect failure
- **State_NeedConnection**, indicating that the connection is not open or needs to be opened
- **State_NeedRequest**, indicating that the connection is ready and waiting for a request
- **State_NoData**, indicating that no data is available (note that this constant is not currently supported)

url

Type: String

The **url** property of the **JadeHTTPConnection** class contains the URL of the page being requested, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

usePresentationClient

Type: Boolean

The **usePresentationClient** property of the **JadeHTTPConnection** class specifies whether the default WinHTTP (or WinINet) is used from the machine from which the presentation client is running.

The initial value is **false**.

user

Type: String

The **user** property of the **JadeHTTPConnection** class contains the user portion of the **url** property value, where the URL is made up of the following components.

```
"scheme://[user[:password]@]host[:port]/path[?query][#fragment]"
```

If the string is not null, the value of this property is passed when the connection is requested.

JadeHTTPConnection Methods

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

Method	Description
configureProxy	Sets up the proxy server configuration
getHeader	Retrieves an HTTP header
getHttpPage	Returns the requested page
getHttpPageBinary	Returns the complete requested page in binary format, with no text encoding conversion performed
open	Opens an HTTP connection
queryConnectionIsClose	Checks if the HTTP connection is closed
queryContentLength	Returns the length of the response message
queryContentType	Returns the content type of the response message
queryDate	Returns the timestamp at which the response message originated
queryInfo	Retrieves header information associated with an HTTP request
queryStatusCode	Retrieves the HTTP status code returned by the server
readBody	Retrieves the response message from an HTTP request
sendRequest	Sends the specified request to the HTTP server
setAccept	Sets Accept HTTP headers
setContentType	Sets Content-Type HTTP headers
setKeepAlive	Sets the Proxy-Connection HTTP header
setReferer	Sets the Referer HTTP header
setReload	Sets the Pragma HTTP header
setSoapAction	Sets the SOAPAction HTTP header
setUserAgent	Sets the User-Agent HTTP header

configureProxy

Signature
```
configureProxy(config:   Integer;
               host:     String;
               user:     String;
               password: String): Boolean;
```

The **configureProxy** method of the **JadeHTTPConnection** class sets up the proxy server configuration. The **config** parameter, which defines the type of access required, can be one of the **JadeHTTPConnection** class constants listed in the following table.

Value	Description
ProxyConfig_Direct	Resolves all host names locally, and does not use a proxy server.
ProxyConfig_PreConfig	Retrieves the proxy or direct configuration from the registry (the default value).
ProxyConfig_PreConfig_NoAuto	Retrieves the proxy or direct configuration from the registry and prevents the use of a start-up Microsoft JScript or **wpad.dat** file.
ProxyConfig_Proxy	Passes requests to the proxy.

The **host** parameter specifies the name of the proxy server or servers to use when proxy access is specified by setting the value of the **config** parameter to **ProxyConfig_Proxy**. If the value of the **proxyConfig** property is not set to **ProxyConfig_Proxy**, the value of the **host** parameter should be null.

If the proxy server requires authentication, the **user** parameter contains the user name provided to the proxy server. If the **proxyConfig** property is not set to **ProxyConfig_Proxy**, the value of the **user** parameter should be null.

If the proxy server requires authentication, the **password** parameter contains the password provided to the proxy server. If the **proxyConfig** property is not set to **ProxyConfig_Proxy**, the value of the **password** parameter should be null.

The **configureProxy** method returns **true** if the method call was successful in setting up the proxy server configuration; otherwise it returns **false**.

getHeader

Signature
```
getHeader(type: Integer;
          key:  String): String;
```

The **getHeader** method of the **JadeHTTPConnection** class retrieves an HTTP header.

The **type** parameter, which specifies the type of header to retrieve, can be one of the **JadeHTTPConnection** class constants listed in the following table.

Value	Description
HeaderType_Client	Client header
HeaderType_Server	Server header

The **key** parameter specifies the header key; for example, **Accept, Content-Type**.

The **getHeader** method returns the value of the key if a header with the specified type and key exists; otherwise it returns null.

getHttpPage

Signature
```
getHttpPage(pVerb:        String,
            pServerName:  String;
            pUrlAddress:  String;
            pMessage:     String;
            pContentType: String): String;
```

The **getHttpPage** method of the **JadeHTTPConnection** class returns the requested page.

The **pVerb** parameter specifies the HTTP verb to use in the request. The value can be **"GET"** (the default value), **"POST"**, or null.

The **pServerName** parameter specifies the scheme, host, and other optional parameters. This parameter value is specified as **"scheme://[user[:password]@]host[:port]"**, where:

- The **scheme** value can be **"http"** or **"https"**
- The **user** and **password** values are optional site security
- The **host** value is a host name or an IP number
- The optional **port** value defaults to **80** for HTTP and to **443** for HTTPS

If the value of the **pServerName** parameter is null, the host information portion of the **url** property is used (that is, **[user[:password]@]host[:port]**).

The **pUrlAddress** parameter specifies the relative URI from the server (that is, **[/]path[?query][#fragment]**). If this value is null, the value of the **url** property is used.

The optional **pMessage** parameter specifies the body of the message for POST requests.

The **pContentType** parameter, which specifies the content type, has a default value of **"text/xml; charset=utf-8"**.

The **getHttpPage** method returns a string representing the message response, or it returns null if the request failed.

getHttpPageBinary

Signature
```
getHttpPageBinary(pVerb:        String,
                  pServerName:  String;
                  pUrlAddress:  String;
                  pMessage:     String;
                  pContentType: String): Binary;
```

The **getHttpPageBinary** method of the **JadeHTTPConnection** class returns the complete requested page in binary format, with no text encoding conversion performed.

The **pVerb** parameter specifies the HTTP verb to use in the request. The value can be **"GET"** (the default value), **"POST"**, or null.

The **pServerName** parameter specifies the scheme, host, and other optional parameters. This parameter value is specified as **"scheme://[user[:password]@]host[:port]"**, where:

- The **scheme** value can be **"http"**, **"https"**, or **"jadedirect"**
- The **user** and **password** values are optional site security
- The **host** value is a host name or an IP number

- The **port** value is optional for HTTP (defaults to **80**) and HTTPS (defaults to **443**), and is required for the **jadedirect** scheme

If the value of the **pServerName** parameter is null, the host information portion of the **url** property is used (that is, **[user[:password]@]host[:port]**).

The **pUrlAddress** parameter specifies the relative URI from the server (that is, **[/]path[?query][#fragment]**). If this value is null, the value of the **url** property is used.

The optional **pMessage** parameter specifies the body of the message for POST requests (that is, SOAP requests, and so on).

The **pContentType** parameter specifies the content type. For SOAP 1.2, it is set to **"application/soap+xml"** as well as an optional **action** parameter. If the value of the **pContentType** parameter is null, it defaults to **"text/xml; charset=utf-8"**.

The **getHttpPageBinary** method returns the complete page, or it returns null if there is a problem. Use the **queryStatusCode** method to determine the HTTP error.

open

Signature `open(closeFirst: Boolean): Boolean;`

The **open** method of the **JadeHTTPConnection** class opens an HTTP connection.

If the HTTP connection is currently open and you want it closed before a new connection is opened, set the **closeFirst** parameter to **true**.

Note The **url** property must be set before the connection can be opened.

This method returns **true** if the open operation was successful; otherwise it returns **false**.

The value of the **state** property is set to **State_NeedRequest** if the open action was successful; otherwise it is set to **State_Failure** if it was unsuccessful.

queryConnectionIsClose

Signature `queryConnectionIsClose(): Boolean;`

The **queryConnectionIsClose** method of the **JadeHTTPConnection** class checks if the HTTP connection is closed.

This method returns **true** if the connection is closed; otherwise it returns **false**.

You can call this method to check the "Connection" header state before calling the **queryInfo** method, for example.

queryContentLength

Signature `queryContentLength(): Integer;`

The **queryContentLength** method of the **JadeHTTPConnection** class returns the value provided by the *Content-Length* header, which contains the length of the response message in bytes. The *Content-Length* header is optional for messages. If it is not present, **queryContentLength** returns zero (**0**). This can occur when the message body length is greater than zero (**0**).

queryContentType

Signature `queryContentType(): String;`

The **queryContentType** method of the **JadeHTTPConnection** class returns the content type of the response message (for example, **"text/html"**); or it returns null if the response message is empty.

You can call this method to check the "Content-Type" header before calling the **queryInfo** method, for example.

queryDate

Signature `queryDate(): TimeStamp;`

The **queryDate** method of the **JadeHTTPConnection** class returns the timestamp at which the response message originated.

This method returns a timestamp representing the date and time of origin on the response message (for example, **Mon, 27 Jun 2011 23:43:27 GMT**); or it returns null if the response message is empty.

You can call this method to check the "Date" header before calling the **queryInfo** method, for example.

queryInfo

Signature
```
queryInfo(key:   String,
          index: Integer io): String;
```

The **queryInfo** method of the **JadeHTTPConnection** class retrieves header information associated with an HTTP request.

The **key** parameter specifies the HTTP header key that is to be retrieved; for example, **Accept, Date**.

The **index** parameter enumerates multiple headers with the same key. When calling the function, this parameter is the index of the specified header to return (which is usually zero (**0**)). When the function returns, this parameter is the index of the next header. If the next index cannot be found, a null value is returned.

queryStatusCode

Signature `queryStatusCode(): Integer;`

The **queryStatusCode** method of the **JadeHTTPConnection** class retrieves the HTTP status code returned by the server, or it return null if the status code is empty.

readBody

Signature `readBody(length: Integer): Binary;`

The **readBody** method of the **JadeHTTPConnection** class retrieves the response message from an HTTP request.

The length parameter, if specified, preallocates space for the **responseBody** property. The default value is zero (**0**).

Note For improved performance with large messages, we recommend that you use the content length if it is known or can be estimated.

This method returns a **Binary** value representing the message response of an HTTP **"POST"** of the passed URL returned from the server. The **responseBody** property is set to this returned value. Successive calls to **readBody** return a null **Binary** value, as the response has already been read.

sendRequest

Signature
```
sendRequest(verb:                String,
            additionalHeaders:   String;
            optionalPostPutData: String): Boolean;
```

The **sendRequest** method of the **JadeHTTPConnection** class sends the specified request to the HTTP server and allows the client to specify additional headers to send along with the request. In addition, it allows the client to specify optional data to send to the HTTP server immediately following the request headers.

Note This feature is generally used for write operations such as PUT and POST.

The **verb** parameter specifies the HTTP verb to use in the request. The value can be "**GET"** (the default), **"POST"**, or null. A null value implies a **"GET"** request.

The **additionalHeaders** parameter specifies a string containing the additional headers to be appended to the request. If there are no additional headers to be appended, the value of this parameter can be null.

The **optionalPostPutData** parameter specifies a string containing any optional data to be sent immediately after the request headers. This parameter is generally used for POST and PUT operations. The optional data can be the resource or information being posted to the server. If there is no optional data to send, the value of this parameter can be null.

This method returns **true** if the send request was successful; otherwise it returns **false**.

setAccept

Signature
```
setAccept(value: String,
          index: Integer io);
```

The **setAccept** method of the **JadeHTTPConnection** class sets "Accept" HTTP headers.

The **value** parameter specifies the string value for the Accept key; for example, **"text/xml"**.

If you specify a value for the **index** parameter and the Accept key exists at the specified position in the list of headers, the corresponding entry is removed from the list if the value is null. If the value is non-null, the corresponding value is replaced. If there is no Accept key in the specified position, the key-value pair is added at the position.

setContentType

Signature
```
setContentType(value: String,
               index: Integer io);
```

The **setContentType** method of the **JadeHTTPConnection** class sets "Content-Type" HTTP headers.

The **value** parameter specifies the string value for the Content-Type key; for example, **"text/xml; charset=utf-8"**.

If you specify a value for the **index** parameter and the Content-Type key exists at the specified position in the list of headers, the corresponding entry is removed from the list if the value is null. If the value is non-null, the corresponding value is replaced. If there is no Accept key in the specified position, the key-value pair is added at this position.

setKeepAlive

Signature `setKeepAlive(flag: Boolean);`

The **setKeepAlive** method of the **JadeHTTPConnection** class sets the "Proxy-Connection" HTTP header.

Set the value of the **flag** parameter to **true** to generate the "Proxy-Connection : Keep-Alive" header; or set the value to **false** to remove the "Proxy-Connection : Keep-Alive" header if it exists.

setReferer

Signature `setReferer(value: String);`

The **setReferer** method of the **JadeHTTPConnection** class sets the "Referer" HTTP header.

Set the **value** parameter to the string value for the Referer key; for example:

```
"http://www.w3.org/hypertext/DataSources/Overview.html"
```

If the value of the **value** parameter is null and the Referer key exists in the list of HTTP headers, the corresponding entry is removed from the list. If the value is non-null, the corresponding value is replaced.

setReload

Signature `setReload(flag: Boolean);`

The **setReload** method of the **JadeHTTPConnection** class sets the "Pragma" HTTP header.

Set the value of the **flag** parameter to **true** to generate the "Pragma: no-cache" HTTP header; or set the value to **false** to remove the "Pragma: no-cache" header if it exists.

setSoapAction

Signature `setSoapAction(value: String);`

The **setSoapAction** method of the **JadeHTTPConnection** class sets the "SOAPAction" HTTP header.

Set the **value** parameter to the string value for the "SOAPAction" header. If the value of this parameter is null and the SOAPAction key exists in the list of headers, the corresponding entry is removed from the list. If the value is non-null, the corresponding value is replaced.

setUserAgent

Signature `setUserAgent(value: String);`

The **setUserAgent** method of the **JadeHTTPConnection** class sets the "User-Agent" HTTP header.

Set the **value** parameter to the string value for the "User-Agent" header. If the value of this parameter is null and the User-Agent key exists in the list of headers, the corresponding entry is removed from the list. If the value is non-null, the corresponding value is replaced.

JadeIdentifierArray Class

The **JadeIdentifierArray** class is an ordered collection of **String** values with a length less than or equal to 100 characters.

This class is designed to store identifiers for JADE entities (for example, schemas, classes, forms, and methods), which have a maximum length of 100 characters. The **StringArray** class, which can accommodate strings up to 62 characters only, is unsuitable for storing identifiers.

Inherits From: StringArray

Inherited By: (None)

JadeInternetTCPIPConnection Class

The **JadeInternetTCPIPConnection** class implements the interface defined by the **TcpIpConnection** class specifically for the Internet *Transmission Control Protocol / Internet Protocol* (TCP/IP) API.

Callback methods must match the signature required by the calling asynchronous **Connection** or **TcpIpConnection** class method. Only one synchronous operation can be performed at one time. Only one synchronous read operation can be performed at one time on a connection.

Note As you can create a **JadeInternetTCPIPConnection** object as a shared transient object, you can pass it to another JADE process on the same JADE node, if required. Shared transient Internet 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 **JadeInternetTCPIPConnection** object, by setting the **TcpIpConnection** class **port** property or the **Connection** class **name** property.

For details about the methods defined in the **JadeInternetTCPIPConnection** class, see "JadeInternetTCPIPConnection Methods", in the following subsection. For details about subclassing the **JadeInternetTCPIPConnection** class and reimplementing methods for the control of HTML documents, see "Reimplementing Methods to Interrupt the Processing Cycle", in Chapter 12 of the *JADE Development Environment User's Guide*.

Inherits From: TcpIpConnection

Inherited By: (None)

JadeInternetTCPIPConnection Methods

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

Method	Description
openPipeCallback	Initiates an asynchronous read of the opened TCP/IP connection
readBinary	Reads binary data from the Internet connection and returns when the specified number of bytes has been read or when a block of data is received
readDataWithLength	Reads data from the Internet TCP/IP connection and returns when the specified length of data is read
readPipeCallback	Performs Web session evaluation processing
sendReply	Sends the formatted HyperText Markup Language (HTML) page to the opened Internet TCP/IP connection
writeBinary	Writes binary data to the Internet TCP/IP connection and returns when the operation is complete

openPipeCallback

Signature `openPipeCallback(pipe: Connection) updating;`

The **openPipeCallback** method of the **JadeInternetTCPIPConnection** class is called when the **jadehttp** library file opens the Internet server end of the TCP/IP connection, to initiate an asynchronous read of the opened TCP/IP connection.

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.

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **JadeInternetTCPIPConnection** class reads binary data from the TCP/IP 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.

Note This method is used by the **readDataWithLength** method. You should normally not call it directly because of the format of messages received. (For details, see the **readDataWithLength** method.)

This method can be called only when the value of the **Connection** class **state** property is **Connected** (2) and a zero-length request calls the **readDataWithLength** method, expecting the length to be read to be the first eight bytes if the message.

Only one synchronous or asynchronous read operation can be performed at one time on a connection. See also the **Connection** class **timeout** property.

readDataWithLength

Signature `readDataWithLength(bin: Binary): Binary;`

The **readDataWithLength** method of the **JadeInternetTCPIPConnection** class reads a message from the TCP/IP connection expecting it to be formatted so that the first eight bytes contain the length of the message to be read. The **bin** parameter contains the information already read by the **readPipeCallback** method.

Data sent from TCP/IP can receive from one to many characters from the **readPipeCallback** method. This routine then calls the **readDataWithLength** method, passing the data already received. The **readDataWithLength** method then reads the rest of the message and returns the message, minus the length characters.

This method can be called only when the value of the **Connection** class **state** property is **Connected** (2).

Only one synchronous or asynchronous read operation can be performed at one time on a connection. See also the **Connection** class **timeout** property.

readPipeCallback

Signature
```
readPipeCallback(pipe: Connection;
                 msg:  Binary) updating;
```

The **readPipeCallback** method of the **JadeInternetTCPIPConnection** class is called to perform Web session evaluation processing when data is available on the TCP/IP connection. This method receives the initial data sent by the TCP/IP connection and it then calls the **readDataWithLength** method to accumulate the rest of the message.

The **readPipeCallback** method handles the file transfer read and write actions so that the message can be read in small pieces. If you reimplement this method, it must call **inheritMethod** so that any file transfers are processed.

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.

sendReply

Signature `sendReply(html: Binary) updating;`

The **sendReply** method of the **JadeInternetTCPIPConnection** class sends the formatted HyperText Markup Language (HTML) page back to the opened TCP/IP connection and starts the next read request.

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.

writeBinary

Signature `writeBinary(buffer: Binary);`

The **writeBinary** method of the **JadeInternetTCPIPConnection** class writes binary data to the connection and returns when the operation is complete.

This method appends the eight-character ANSI length of the data to the front of the message before transmission.

The **writeBinary** method can be called only when the value of the **Connection** class **state** property is **Connected** (2).

JadeJson Class

The **JadeJson** class is a transient-only **Object** subclass that provides standalone JSON functionality that is independent of the Representational State Transfer (REST) Application Programming Interface (API).

The **JadeJson** class enables you to create, load, unload, and parse JSON in the same way you can with XML.

The following table lists the C# type expected for each JADE type.

JADE Type	C# Type
Binary	Byte []
Boolean	Boolean
Byte	Byte
Character	Char
Date	DateTime
Decimal	Decimal
Integer	Int32
Integer64	Int64
JadeBytes	Byte []
HugeStringArray	String []
Point	String format <integer>, <integer>
Real	Double
String	String
StringUtf8	String
Time	DateTime
TimeStamp	DateTime
TimeStampInterval	TimeSpan
TimeStampOffset	DateTime (with UTC offset set)

For details about the constants and methods defined in the **JadeJson** class, see "JadeJson Class Constants" and "JadeJson Methods", the following subsections.

Inherits From: Object

Inherited By: (None)

JadeJson Constants

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

Class Constant	Integer Value	Description
Format_Json_Microsoft	0	The JSON format as is expected by the Microsoft **DataContractJsonSerializer** class. This format type does not support circular references or multiple references to the same object in the returned data. (An exception is generated if the situation is detected.)
Format_Json_Newton	2	The JSON format is as expected by the NewtonSoft **Json** class software. This format is different from Microsoft in the structure, tags, and the format of some primitive types. The output includes identifiers for each object and references to already included objects, and therefore supports circular and multiple references to the same object in the returned data.

JadeJson Methods

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

Method	Description
generateJson	Generates JSON from a primitive type variable or an object
generateJsonFile	Generates JSON from a primitive type variable or an object, and writes the output to a file
parse	Parses JSON text to create and populate an object and all referenced objects
parseFile	Reads and parses JSON text from a file to create and populate an object and all referenced objects
parsePrimitive	Parses JSON text for a primitive type and returns the primitive type value
parsePrimitiveFile	Parses JSON text for a primitive type from a file and returns the primitive type value

generateJson

Signature

```
generateJson(source: Any;
             format: Integer): String;
```

The **generateJson** method of the **JadeJson** class generates JSON from a primitive type variable or an object.

The **source** parameter specifies the primitive type variable or object to be generated as JSON text output. The **format** parameter specifies the format type to generate (that is, **Format_Json_Microsoft** or **Format_Json_Newton**).

The return value is the generated JSON string.

Calling this method generates the JSON text for the source type in Microsoft or NewtonSoft format. These formats are different and not compatible in some cases.

If the **source** parameter is a primitive type, the returned text is the JSON textual version of that primitive type data; for example:

```
String: a string   Json ="a string"
Date: 23 November 2015   Json= "\/Date(1448236800000)\/"
```

For an object, the JSON output contains a textual description of the source object and the entire tree of objects referenced.

The format type represented by the **Format_Json_Microsoft** class constant rejects an object where an object is referenced more than once in the object tree (Microsoft JSON handling does not handle that situation).

The format type represented by the **Format_Json_Newton** class constant assigns a unique tag to each object, allowing multiple and circular references to be handled so that each object is included once only and subsequent references use the assigned tag for identification.

The following is a Microsoft example.

```
{"__
type":"Customer","name":"smith","sex":"M","description":"","ignore":false,"image":
[]}
```

The following is a NewtonSoft example that shows a type tag and an object identifier.

```
{"$type":"Customer","$id":"Customer4294967297","name":"Smith","bin":
[],"description":"","ignore":false,"image":[]}
```

The JSON for the description of each object includes the type of the object in the text.

generateJsonFile

Signature

```
generateJsonFile(source:                Any;
                 format:                Integer;
                 fileName:              String;
                 usePresentationClient: Boolean);
```

The **generateJsonFile** method of the **JadeJson** class generates JSON from a primitive type variable or an object and writes the output to a file.

The **generateJsonFile** method parameters are:

- The **source** parameter specifies the primitive type variable or object to be generated as JSON text output.
- The **format** parameter specifies the format type to generate (that is, **Format_Json_Microsoft** or **Format_Json_Newton**).
- The **fileName** parameter specifies the name of the file to create.
- The **usePresentationClient** parameter specifies where the file is to be created. It specifies **true** if the file is to be created on the presentation client and the JADE process is a thin client; otherwise the file is created on the same node on which the logic is executing (an application server, a server, or a client for a standard (fat) client process).

Calling this method generates the JSON text for the source type in Microsoft or NewtonSoft format. A file is created and the generated JSON text is written to the file, which is given the file name specified in the **fileName** parameter. It is written to the presentation client or onto the JADE logic execution node. The file name should be an absolute path.

The **generateJsonFile** method does not create any referenced directories.

For details about creating the JSON, see the **generateJson** method.

parse

Signature

```
parse(json:           String;
      type:           Class;
      createdObjects: ObjectArray input): Object;
```

The **parse** method of the **JadeJson** class parses JSON text to create and populate an object and all referenced objects.

The **parse** method parameters are:

- The **json** parameter specifies the source string to parse in Microsoft or NewtonSoft format (parser handles both formats without needing to know the type of the JSON).
- The **type** parameter specifies the object type of the data. If the JSON text includes a type tag, the type tag must be the same as the class specified in this parameter or a subclass of the specified class. An object of that type is created and populated. If the JSON does not include a type tag, an object of the class specified in the **type** parameter is created and populated.
- The **createdObjects** parameter specifies a transient object array supplied by the caller. All objects created are added to the array (the array is not cleared by the method). It is the responsibility of the caller to delete all objects returned from the method.

The return value is the object created from the parsed JSON string together with any referenced objects (it is the first object added to the array specified in the **createdObjects** parameter).

The return value is null if the JSON contains null or is empty.

An exception is generated if the text cannot be parsed successfully.

JSON text does not necessarily include a tag indicating the type of the data. If the JSON does not include a type tag, the JSON parser must assume that the data is of the correct type. Any properties not found on the class of any object are ignored and no error is raised. If the JSON text does not match the expected type, it could be because no property values are set on the created object or that the data does not match the property type.

parseFile

Signature

```
parseFile(type:                Class;
          createdObjects:      ObjectArray input;
          fileName:            String;
          usePresentationClient: Boolean): Object;
```

The **parseFile** method of the **JadeJson** class reads and parses JSON text from a file to create and populate an object and all referenced objects.

The **parseFile** method parameters are:

- The **type** parameter specifies the object type of the data. If the JSON text includes a type tag, the type tag must be the same as the class specified in this parameter or a subclass of the specified class. An object of that type is created and populated. If the JSON does not include a type tag, an object of the class specified in the **type** parameter is created and populated.
- The **createdObjects** parameter specifies a transient object array supplied by the caller. All objects created are added to the array (the array is not cleared by the method). It is the responsibility of the caller to delete all objects returned from the method.

- The **fileName** parameter specifies the name of the file to read.
- The **usePresentationClient** parameter specifies where the file is to be read. It specifies **true** if the file is to be read on the presentation client and the JADE process is a thin client; otherwise the file will be read from the same node where the logic is executing (an application server, a server, or a client for a standard (fat) client process).

The return value is the object created from the parsed JSON string in the file together with any referenced objects. (It is the first object added to the array specified in the **createdObjects** parameter.)

The returned value is null if the JSON contains null or is empty.

An exception is generated if the text cannot be parsed successfully.

JSON text does not necessarily include a tag indicating the type of the data. If the JSON does not include a type tag, the JSON parser must assume that the data is of the correct type. Any properties not found on the class of any object are ignored and no error is raised. If the JSON text does not match the expected type, it could be because no property values are set on the created object.

parsePrimitive

Signature
```
parsePrimitive(json: String;
               type: PrimType): Any;
```

The **parsePrimitive** method of the **JadeJson** class parses JSON text for a primitive type and returns the primitive type value.

The **json** parameter specifies the source string to parse. The **type** parameter specifies the primitive type of the data.

The return value is the primitive value of the parsed JSON string.

Calling this method parses the string in terms of the passed primitive type and returns the primitive type value represented.

JSON text for a primitive type does not include the type of the data. For a primitive value, the caller must know the type of the data represented in JSON text.

An exception is generated if the text cannot be parsed successfully.

parsePrimitiveFile

Signature
```
parsePrimitiveFile(type:                 PrimType;
                   fileName:             String;
                   usePresentationClient: Boolean): Any;
```

The **parsePrimitiveFile** method of the **JadeJson** class parses JSON text for a primitive type from a file and returns the primitive type value.

The **parsePrimitiveFile** method parameters are:

- The **type** parameter specifies the primitive type of the data.
- The **fileName** parameter specifies the name of the file to read.
- The **usePresentationClient** parameter specifies where the file is to be read. It specifies **true** if the file is to be read on the presentation client and the JADE process is a thin client; otherwise the file will be read from the same node where the logic is executing (an application server, a server, or a client for a standard (fat) client process).

The return value is the primitive value of the parsed JSON string.

Calling this method reads the indicated file and parses the string in terms of the passed primitive type and returns the primitive type value represented.

An exception is generated if the text cannot be parsed successfully.

JadeLicenceInfo Class

The **JadeLicenceInfo** class encapsulates the behavior required to get licence information for your JADE database.

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

Inherits From: Object

Inherited By: (None)

JadeLicenceInfo Class Constants

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

Constant	Character Value
Restriction_Compact	#'03'
Restriction_Enterprise	#'01'
Restriction_Free	#'02'
Restriction_None	#'00'

JadeLicenceInfo Properties

The properties defined in the **JadeLicenceInfo** class following a **getLicenceInfo** method call are summarized in the following table.

Property	Contains the …
developmentLicences	Maximum number of development licences available for your system
expiryDate	Expiry date of your licence or null ("") if perpetual for your system
jadeStandardMin	Number of standard fat client runtime licences
jadeThinMin	Number of JADE thin client runtime licences
licenceName	Licence name of your organization
licenceRestriction	Licence restriction, if any, for your system
maxDBSize	Maximum database size (in gigabytes) permitted by an enterprise or free licence
nHtmlThinSessions	Number of HTML thin client session licences currently being used
nJadeDevProcesses	Number of JADE development licences currently being used
nJadeThinNonJadeDevProcesses	Number of JADE thin client process licences currently being used
nNonJadeDevProcesses	Number of JADE standard fat client process licences currently being used
nProcessesLeft	Number of process licences remaining

Property	Contains the …
processLicences	Maximum number of process licenses for your system
uuid	Unique identifier for the database

developmentLicences

Type: Integer

The read-only **developmentLicences** property of the **JadeLicenceInfo** class contains the number of development licences available for your system.

expiryDate

Type: Date

The read-only **expiryDate** property of the **JadeLicenceInfo** class contains the expiry date of your licence or null (**""**) if your licence is perpetual for your system.

jadeStandardMin

Type: Integer

The read-only **jadeStandardMin** property of the **JadeLicenceInfo** class contains the number of standard fat client runtime licences for your system.

jadeThinMin

Type: Integer

The read-only **jadeThinMin** property of the **JadeLicenceInfo** class contains the number of JADE thin client runtime licences for your organization.

licenceName

Type: String[50]

The read-only **licenceName** property of the **JadeLicenceInfo** class contains the licence name of your organization.

licenceRestriction

Type: Character

The read-only **licenceRestriction** property of the **JadeLicenceInfo** class contains the licence restriction for your system, if any, and is one of the constant values listed in the following table.

JadeLicenceInfo Class Constant	Character Value
Restriction_Compact	#'03'
Restriction_Enterprise	#'01'
Restriction_Free	#'02'
Restriction_None	#'00'

maxDBSize

Type: Integer

The read-only **maxDBSize** property of the **JadeLicenceInfo** class contains the maximum size (in gigabytes) of the database permitted by a **Restriction_Enterprise** or **Restriction_Free** licence.

nHtmlThinSessions

Type: Integer

The read-only **nHtmlThinSessions** property of the **JadeLicenceInfo** class contains the number of HTML thin client session licences currently being used.

nJadeDevProcesses

Type: Integer

The read-only **nJadeDevProcesses** property of the **JadeLicenceInfo** class contains the number of JADE development licences currently being used.

nJadeThinNonJadeDevProcesses

Type: Integer

The read-only **nJadeThinNonJadeDevProcesses** property of the **JadeLicenceInfo** class contains the number of JADE thin client process licences currently being used.

nNonJadeDevProcesses

Type: Integer

The read-only **nNonJadeDevProcesses** property of the **JadeLicenceInfo** class contains the number of JADE standard fat client process licences currently being used.

nProcessesLeft

Type: Integer

The read-only **nProcessesLeft** property of the **JadeLicenceInfo** class contains the number of process licenses currently remaining.

processLicences

Type: Integer

The read-only **processLicences** property of the **JadeLicenceInfo** class contains the maximum number of process licences for your system.

uuid

Type: String

The read-only **uuid** property of the **JadeLicenceInfo** class contains a unique identifier generated when the database is licensed. This identifier is constant for the lifetime of the database unless it is relicensed or it partakes in a hostile database of a primary server in an SDS environment.

See also the batch JADE Database utility **generateUUID** command under "Running the JADE Database Utility in Batch Mode", in Chapter 1 of the *JADE Database Administration Guide*, which enables you to explicitly control the re-generation of database unique identifiers when required.

JadeLicenceInfo Methods

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

Method	Description
display	Returns a string containing the information stored in the properties of the **JadeLicenceInfo** object following a **getLicenceInfo** method call
getLicenceInfo	Gets your JADE licence information

display

Signature `display(): String;`

After a call to the **JadeLicenceInfo** object following a **getLicenceInfo** method, the **display** method of the **JadeLicenceInfo** class returns a string of the receiver containing the JADE licence information stored in the **JadeLicenceInfo** object properties listed in the following table.

Displays ...	Property	Example
Licence name	licenceName	Licensed Demonstration Company
uuid	uuid	99z9zzz9-9999-z999-9z9z- z99999zzzz99
Maximum number of development licences available	developmentLicences	50
Maximum number of process licences available	processLicences	500
Restrictions	licenceRestriction, maxDBSize	Enterprise version, max DB size=4 GB
Expiry Date	expiryDate	perpetual
Number of licences reserved for standard processes	jadeStandardMin	0
Number of licences reserved for thin client processes	jadeThinMin	0
Number of standard process licences in use	nNonJadeDevProcesses	4

Displays …	Property	Example
Number of development licences in use	nJadeDevProcesses	1
Number of thin client processes in use	nJadeThinNonJadeDevProcesses	0
Number of HTML thin client processes in use	nHtmlThinSessions	0
Number of process licences free	nProcessesLeft	496

Use the **JadeLicenceInfo** class **getLicenceInfo** method to get the appropriate information that you require.

getLicenceInfo

Signature `getLicenceInfo() serverExecution;`

The **getLicenceInfo** method of the **JadeLicenceInfo** class gets information about your JADE licence. (For details about the information that can be returned when you call this method, see the **JadeLicenceInfo** class **display** method.)

The following example shows the use of the **getLicenceInfo** method.

```
getCompanyName(): String;
vars
    jli : JadeLicenceInfo;
begin
    create jli transient;
    jli.getLicenceInfo;
    return jli.licenceName;
end;
```

JadeLog Class

The **JadeLog** class encapsulates the behavior required to create text log files in JADE applications. (You cannot write to the default JADE log file, but you can use the features defined in this class to create a log file and output messages to that file.)

You can use this logging mechanism to output the following classes of messages.

- Log messages, which are always output to the log file. A **log** message forces any pending output to the log file by forcing a flush of the output buffer to disk.
- Trace messages, which are output to the log file only when a value has been specified for the **selector** property and the initialization of the application locates a parameter with the same name as the ***<selector-name>*** parameter value in the [JadeLog] section of the JADE initialization file and this parameter is set to a non-zero value. The value in the **selector** property and the matching parameter name are case-insensitive.

  For example, tracing is enabled if the **selector** property passed when the application starts up has a value of **mylogging** and the JADE initialization file contains the following:

  ```
  [JadeLog]
  MyLogging=1
  ```

  If a parameter with a corresponding name is located in the JADE initialization file parameter when the application starts up but it is set to zero (**0**) or it does not have a numeric value, tracing is disabled.

To output a message to the log file, create and then initialize an instance of the **JadeLog** class. You must specify a file name in the **fileName** property, and you can set other logging options. For details, see "JadeLog Class Properties", later in this section. To output log messages, send messages with the message binary or text to the log object by using the **log**, **info**, or **trace** methods or one of the methods whose name starts with **log**, **info**, or **trace** (for example, **logServer**, **infoClient**, or **traceDumpClient**).

Messages sent in the **log** and **info** methods are always output. The file buffer is flushed when a message is output with the **log** method and it is buffered when it is output using the **info** or **trace** methods. (The output of a message with the **trace** method is determined by the setting of the ***<selector-name>*** parameter in the [JadeLog] section of the JADE initialization file having a non-zero value and the same name as the **selector** property value.)

Tip As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

The **JadeLog** class supports log messages and informational (status) messages.

The file buffer is flushed when a log message is output, ensuring that the message is written to the file. The file size is checked at the same time to see if a new log file is required. As neither of these actions occurs when an informational message is logged, the file buffer is flushed by the operating system and no check is performed on the file size. Although you can output a log message to force the file size to be checked, this operation is more expensive than letting the operating system flush the buffer by checking the file size periodically when logging informational messages.

The **JadeLog** class destructor method closes the log file. When multiple instances of the **JadeLog** class use the same file, the log file is closed when all instances have been deleted.

If the value of the **filePath** property is **null** and the **fileName** property does not contain a path, log files are output to the directory specified by the **LogDirectory** parameter in the [JadeLog] section of the JADE initialization file.

If a file is versioned (the default) when the size of the file reaches the value specified in the **maxFileSize** property, a new file is created with the next available version number; for example, **mymsg1.log** is followed by **mymsg2.log**, and so on. If the **maxFileSize** property is set to zero (**0**), the value of the **MaxLogFileSize** parameter in the [JadeLog] section of the JADE initialization file is used.

The following example shows the use of properties and methods defined in the **JadeLog** class.

```
vars
    myLog : JadeLog;
begin
    create myLog;
    // Tracing will be output if the value of MyLogging is
    // non-zero in the [JadeLog] section of the Jade.ini file
    myLog.selector := 'MyLogging';
    // Output is to MyLog[n].log in the default log directory
    myLog.fileName := 'MyLog';
    // Log files are versioned at 10MB
    myLog.maxFileSize := 10000000;
    // Log a start-up (informational) message - do not flush the file
    myLog.info('starting up');
    // Log a message on the server - do not flush the file
    myLog.infoServer('tracing about to start on a client');
    // The following message is output only if MyLogging is
    // non-zero in the Jade.ini file
    myLog.trace('start time: ' & app.actualTime.String);
epilog
    // Unconditionally output a message - flush the file
    myLog.log('all done');
    // Deleting the log instance will close the log file
    delete myLog;
end;
```

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

Inherits From: Object

Inherited By: (None)

JadeLog Class Properties

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

Property	Description
bufferOutput	Specifies whether log messages are buffered
fileExtension	Contains the log file extension
fileName	Contains the log file name
filePath	Contains the log file path
formatOutput	Specifies whether the date and time are output with each message
maxFileSize	Contains the maximum file size (in bytes) if the file is versioned
selector	Contains the JADE initialization file selector that enables or disables tracing
versionFile	Specifies whether the file is versioned

bufferOutput

Type: Boolean

Availability: Read or write at any time

The **bufferOutput** property of the **JadeLog** class specifies whether log messages are buffered.

This property applies only to **log** methods, which force any pending **info** or **trace** buffered messages to be flushed to disk. By default, messages output by calling **log** methods are not buffered; that is, this property is set to **false**.

Set this property to **true** if you do not want the file buffer flushed when a log message is received; that is, **log** methods then act as **info** methods, and messages are buffered until they are flushed by a **commitLog** method call.

fileExtension

Type: String

Availability: Read or write at any time

The **fileExtension** property of the **JadeLog** class contains the extension of the log file (for example, **out**).

Note The period character (**.**), or dot, is not required before the log file extension.

By default, log files have an extension of **log**.

fileName

Type: String

Availability: Read or write at any time

The **fileName** property of the **JadeLog** class contains the name of the log file. If the specified value does not also contain a path, the file is created in the directory identified by the **filePath** property.

Notes You must specify a log file name value in this property.

Changing the name of an active log file closes the current file if the new value differs from the current value. The new file is opened the next time a message is logged.

filePath

Type: String

Availability: Read or write at any time

The **filePath** property of the **JadeLog** class contains a valid log file path.

Note If you output log files to a client node or the server node, the log file path must be valid on both the client and the server.

If the value of the **filePath** property is **null** and the **fileName** property does not contain a path, log files are output to the directory specified by the **LogDirectory** parameter in the [JadeLog] section of the JADE initialization file.

formatOutput

Type: Boolean

Availability: Read or write at any time

The **formatOutput** property of the **JadeLog** class specifies whether the date, time, process-thread identifier, and selector are output with each message to the log file.

By default, the date, time, process identifier, and thread identifier are output with each message; that is, this property is set to **true**. The date is output in *yyyy/MM/dd* format (for example, **2007/05/26**) and the time in *HH:mm:ss* format including milliseconds (for example, **15:01:31.985**).

As the date and time formats are fixed, you cannot change either format.

The following example shows messages output to a log file when this property is set to the default value of **true**.

```
2007/05/26 05:01:31.985 00f58-b3c MyLogging: starting up
2007/05/26 05:01:31.985 00f58-b3c MyLogging: tracing about to start on a client
2007/05/26 05:01:31.985 00f58-b3c MyLogging: start time: 26 May 2007, 05:01:31
2007/05/26 05:01:34.281 00f58-b3c MyLogging: all done
```

In the first line of the above example, **00f58** is the operating system process (that is, the JADE node) and **b3c** is the thread (that is, the JADE process).

The following example shows the same log file data when only messages are output; that is, this property is set to **false**.

```
starting up
tracing about to start on a client
start time: 26 May 2007, 05:01:31
all done
```

maxFileSize

Type: Integer

Availability: Read or write at any time

The **maxFileSize** property of the **JadeLog** class contains the maximum file size (in bytes) of the log file if the file is versioned (that is, if the **versionFile** property is set to the default value of **true**).

When a versioned log file exceeds the maximum size, a new log file is created with the next available version number; for example, **mymsg1.log** is followed by **mymsg2.log**, and so on.

If the **maxFileSize** property is set to zero (**0**), the value of the **MaxLogFileSize** parameter in the [JadeLog] section of the JADE initialization file is used.

selector

Type: String

Availability: Read or write at any time

The **selector** property of the **JadeLog** class contains the JADE initialization file selector that enables or disables tracing.

The selector is a string that is used by the logging mechanism to determine whether trace messages are output. If a parameter with the same name as the value of the **selector** property is located in the [JadeLog] section of the JADE initialization file and the parameter has a non-zero numeric value when the application starts up, trace messages are output to the log file. (The value of the **selector** property and the JADE initialization file the ***<selector-name>*** parameter are case-insensitive.)

Note If the **ini** parameter is not specified in the JADE command line, the logging mechanism looks for a **jadtrace.ini** file in the Windows directory (for example, the **WINNT** directory). If the **jadtrace.ini** file cannot be located, the selector key in the JADE initialization file is assumed to be zero (**0**).

versionFile

Type: Boolean

Availability: Read or write at any time

The **versionFile** property of the **JadeLog** class specifies whether the log file is versioned.

By default, log files are versioned; that is, this property is set to **true**. Set this property to **false** if you want to ignore the value of the **JadeLog** class **maxFileSize** property and continue writing to the log file (for example, **mymsg1.log**). There is no maximum size of this log and trace messages file.

For details about incrementing versioned log files when the maximum file size is exceeded, see the **JadeLog** class **maxFileSize** property.

JadeLog Class Methods

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

Method	Description
commitLog	Flushes any pending **info** and **trace** messages in the file buffer to disk
getActualFileName	Returns the path, name, and version of the current log file
getActualFileNameClient	Returns the path, name, and version of the current log file from the client node
getActualFileNameServer	Returns the path, name, and version of the current log file from the server node
info	Outputs a message without flushing the output buffer (executed on the local, or current, node)
infoClient	Outputs a message without flushing the output buffer (executed on the client node)
infoDump	Outputs a binary block without flushing the output buffer (executed on the local, or current, node)
infoDumpClient	Outputs a binary block without flushing the output buffer (executed on the client node)
infoDumpServer	Outputs a binary block without flushing the output buffer (executed on the server node)
infoServer	Outputs a message without flushing the output buffer (executed on the server node)

Method	Description
log	Outputs a message and flushes the output buffer to disk (executed on the local, or current, node)
logClient	Outputs a message and flushes the output buffer to disk (executed on the client node)
logDump	Outputs a binary block and flushes the output buffer to disk (executed on the local, or current, node)
logDumpClient	Outputs a binary block and flushes the output buffer to disk (executed on the client node)
logDumpServer	Outputs a binary block and flushes the output buffer to disk (executed on the server node)
logServer	Outputs a message and flushes the output buffer to disk (executed on the server node)
rollOverLog	Forces the creation of a new log file before the defined value of the **MaxLogFileSize** parameter is reached
rollOverLogClient	Forces the creation of a new log file on the client node before the defined value of the **MaxLogFileSize** parameter is reached
rollOverLogServer	Forces the creation of a new log file on the server node before the defined value of the **MaxLogFileSize** parameter is reached
trace	Conditionally outputs a message without flushing the output buffer (executed on the local, or current, node)
traceClient	Conditionally outputs a message without flushing the output buffer (executed on the client node)
traceDump	Conditionally outputs a binary block without flushing the output buffer (executed on the local, or current, node)
traceDumpClient	Conditionally outputs a binary block without flushing the output buffer (executed on the client node)
traceDumpServer	Conditionally outputs a binary block without flushing the output buffer (executed on the server node)
traceServer	Conditionally outputs a message without flushing the output buffer (executed on the server node)

By default, **JadeLog** class methods are executed on the local, or current, node.

commitLog

Signature `commitLog() updating;`

The **commitLog** method of the **JadeLog** class flushes the file buffer of the receiver to disk.

If you have set the **bufferOutput** property to **true** so that log methods are output to the file buffer instead of to disk, call this method to:

- Force any pending **info**, **log**, or **trace** buffered messages to be flushed to disk
- Flush any pending **info** or **trace** messages without calling a **log** method

getActualFileName

Signature `getActualFileName(): String updating, final;`

The **getActualFileName** method of the **JadeLog** class returns the path, name, and version of the current log file.

getActualFileNameClient

Signature `getActualFileNameClient(): String updating;`

The **getActualFileNameClient** method of the **JadeLog** class returns the path, name, and version of the current log file.

This method is executed on the client node.

getActualFileNameServer

Signature `getActualFileNameServer(): String updating, final;`

The **getActualFileNameServer** method of the **JadeLog** class returns the path, name, and version of the current log file.

This method is executed on the server node.

info

Signature `info(message: String) updating;`

The **info** method of the **JadeLog** class outputs the message specified in the **message** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the local, or current, node.

infoClient

Signature `infoClient(message: String) updating;`

The **infoClient** method of the **JadeLog** class outputs the message specified in the **message** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the client node.

infoDump

Signature `infoDump(block: Binary) updating;`

The **infoDump** method of the **JadeLog** class outputs the binary block specified in the **block** parameter to the file buffer of the receiver. The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the local, or current, node.

infoDumpClient

Signature `infoDumpClient(block: Binary) updating;`

The **infoDumpClient** method of the **JadeLog** class outputs the binary block specified in the **block** parameter to the file buffer of the receiver. The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the client node.

infoDumpServer

Signature `infoDumpServer(block: Binary) updating;`

The **infoDumpServer** method of the **JadeLog** class outputs the binary block specified in the **block** parameter to the file buffer of the receiver. The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the server node.

infoServer

Signature `infoServer(message: String) updating;`

The **infoServer** method of the **JadeLog** class outputs the message specified in the **message** parameter to the file buffer of the receiver. The output message is appended to the file buffer, which is not flushed when this method is called.

This method is executed on the server node.

log

Signature `log(message: String) updating;`

The **log** method of the **JadeLog** class outputs the message specified in the **message** parameter. A **log** message forces any pending **info** or **trace** methods to be output to disk by forcing a flush of the file buffer of the receiver to disk.

Tip As a lot of calls to **log** APIs increases the amount of physical disk I/O, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the local, or current, node.

logClient

Signature `logClient(message: String) updating;`

The **logClient** method of the **JadeLog** class outputs the message specified in the **message** parameter.

A **logClient** message forces any pending **info** or **trace** methods to be output to the log file by forcing a flush of the file buffer of the receiver to disk. As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the client node.

logDump

Signature `logDump(block: Binary) updating;`

The **logDump** method of the **JadeLog** class outputs the binary block specified in the **block** parameter.

A **logDump** message forces any pending **info** or **trace** methods to be output to the log file by forcing a flush of the file buffer of the receiver to disk. As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the local, or current, node.

logDumpClient

Signature `logDumpClient(block: Binary) updating;`

The **logDumpClient** method of the **JadeLog** class outputs the binary block specified in the **block** parameter.

A **logDumpClient** message forces any pending **info** or **trace** methods to be output to the log file by forcing a flush of the file buffer of the receiver to disk. As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the client node.

logDumpServer

Signature `logDumpServer(block: Binary) updating;`

The **logDumpServer** method of the **JadeLog** class outputs the binary block specified in the **block** parameter.

A **logDumpServer** message forces any pending **info** or **trace** methods to be output to the log file by forcing a flush of the file buffer of the receiver to disk. As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the server node.

logServer

Signature `logServer(message: String) updating;`

The **logServer** method of the **JadeLog** class outputs the message specified in the **message** parameter.

A **logServer** message forces any pending **info** or **trace** methods to be output to the log file by forcing a flush of the file buffer of the receiver to disk.

As a lot of calls to **log** APIs increases the amount of physical disk IO, where possible use the **info** methods to avoid impacting on the performance of the application.

This method is executed on the server node.

rollOverLog

Signature `rollOverLog() updating;`

The **rollOverLog** method of the **JadeLog** class forces the creation of a new log file before the defined value of the **MaxLogFileSize** parameter is reached.

rollOverLogClient

Signature `rollOverLogClient() updating;`

The **rollOverLogClient** method of the **JadeLog** class forces the creation of a new log file before the defined value of the **MaxLogFileSize** parameter is reached. This method is executed on the client node.

rollOverLogServer

Signature `rollOverLogServer() updating;`

The **rollOverLogServer** method of the **JadeLog** class forces the creation of a new log file before the defined value of the **MaxLogFileSize** parameter is reached. This method is executed on the server node.

trace

Signature `trace(message: String) updating;`

The **trace** method of the **JadeLog** class conditionally outputs the message specified in the **message** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the local, or current, node.

traceClient

Signature `traceClient(message: String) updating;`

The **traceClient** method of the **JadeLog** class conditionally outputs the message specified in the **message** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the client node.

traceDump

Signature `traceDump(block: Binary) updating;`

The **traceDump** method of the **JadeLog** class conditionally outputs the binary block specified in the **block** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the local, or current, node.

traceDumpClient

Signature `traceDumpClient(block: Binary) updating;`

The **traceDumpClient** method of the **JadeLog** class conditionally outputs the binary block specified in the **block** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the client node.

traceDumpServer

Signature `traceDumpServer(block: Binary) updating;`

The **traceDumpServer** method of the **JadeLog** class conditionally outputs the binary block specified in the **block** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the server node.

traceServer

Signature `traceServer(message: String) updating;`

The **traceServer** method of the **JadeLog** class conditionally outputs the message specified in the **message** parameter to the file buffer of the receiver.

The output message is appended to the file buffer, which is not flushed when this method is called. For details about enabling or disabling tracing, see the **JadeLog** class **selector** property.

This method is executed on the server node.

JadeMessagingException Class

The **JadeMessagingException** class is the transient class that defines behavior for exceptions that occur when processing messages using the JADE messaging framework.

For details about the use of the JADE messaging framework, see Chapter 15, "Using the Messaging Framework", in the *JADE Developer's Reference*. For details about properties defined in the **JadeMessagingException** class, see "JadeMessagingException Properties", later in this section.

Inherits From: NormalException

Inherited By: (None)

JadeMessagingException Properties

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

Property	Contains a reference to ...
theMessage	The message object involved in the exception
theQueue	The queue object involved in the exception
theQueueManager	The queue manager object involved in the exception

theMessage

Type: JadeGenericMessage

The **theMessage** property of the **JadeMessagingException** class contains a reference to the message object involved in the exception.

theQueue

Type: JadeGenericQueue

The **theQueue** property of the **JadeMessagingException** class contains a reference to the queue object involved in the exception.

theQueueManager

Type: JadeGenericQueueManager

The **theQueueManager** property of the **JadeMessagingException** class contains a reference to the queue manager object involved in the exception.

JadeMessagingFactory Class

The **JadeMessagingFactory** class is part of the JADE messaging framework. It encapsulates the behavior for creating and opening message queues.

Create a transient instance of this class, which can create or open a transport-specific queue by using the **openQueue** method. From information supplied to this method, a transient instance of the **JadeGenericQueueManager** class for the appropriate transport is obtained without your code having to create and manage it. The **getQueueManager** method of the **JadeGenericQueue** class returns a reference to the queue manager object for the queue.

For details about the use of the JADE messaging framework, see Chapter 15, "Using the Messaging Framework", in the *JADE Developer's Reference*. For details about the methods defined in the **JadeMessagingFactory** class see "JadeMessagingFactory Class Methods", in the following subsection.

Inherits From: Object

Inherited By: (None)

JadeMessagingFactory Class Methods

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

Method	Description
generateAccessPassword	Returns a unique password that can be associated with a queue
openQueue	A factory method for creating and opening a messaging queue

generateAccessPassword

Signature `generateAccessPassword(): String;`

The **generateAccessPassword** method of the **JadeMessagingFactory** class returns a unique password consisting of 32 random alphanumeric characters (never repeated within a single JADE node).

The password is used as part of the **options** parameter when a **JadeMQ** queue is created by using the **openQueue** method, as shown in the following example.

```
vars
    factory : JadeMessagingFactory;
    password : String;
begin
    create factory transient;
    password := factory.generateAccessPassword;
    myQueue := factory.openQueue("JadeMQ://localnode/TestQ",
                   "Access=Protected; AccessPassword=password; Usage=All");
epilog
    delete factory;
end;
```

Another process must use the same password to open the queue.

openQueue

Signature `openQueue(fullName: String; options: String): JadeGenericQueue;`

The **openQueue** method of the **JadeMessagingFactory** class is a factory method for creating and opening a messaging queue. It returns a transient instance of the **JadeGenericQueue** class from the information in the **fullName** parameter, which is a string specifying queue-related information.

The format of the **fullName** parameter is as follows.

transport-name :// *queue-manager-name* / *queue-name*

The components of the **fullName** parameter are listed in the following table.

Component	Description
transport-name	Name of the message queue transport.
queue-manager-name	Address of the queue manager. Some transports require this to be **localhost**.
queue-name	Name of the queue, formed from US-ASCII letters, digits, underscore, and the period. A trailing asterisk (*) indicates a mask from which a unique name is generated when the queue is opened.

Some transports require that the queue is created before it can be opened.

If the final character of the queue name is an asterisk (*) character, the queue manager replaces the asterisk with a system-generated string to create a unique queue name when the queue is opened.

If the **openQueue** method executes successfully, a queue is returned open for action. If the method fails (for example, if the format of the **fullName** parameter is invalid or the transport name is incorrect), an exception is raised.

The **options** parameter is a string that lists keywords with their associated values, if there are any. The keywords are separated by a semicolon and white space is ignored.

The following options are generically available.

Keyword	Number of Values	Allowed Values	Example
Access	Single	Public Protected Private	Access=Public Access=Protected Access=Private
Create	Single	Never (queue must exist) Always (queue must not exist) Missing (if queue does not exist then create it)	Create=Never Create=Always Create=Missing
MustCreate (same as Create=Always)	None		MustCreate
MustExist (same as Create=Never)	None		MustExist
Usage	Multiple	Get, Put, Inq, All	Usage="Get, Inq"
OtherUsage	Multiple	Get, Put, Inq, All	Usage="Get, Inq"

The following example shows the use of the **openQueue** method.

```
vars
    factory : JadeMessagingFactory;
begin
    create factory transient;
    myQueue := factory.openQueue("JadeMQ://localnode/TestQ",
                                 "Access=Public; MustExist; Usage=Get,Put");
epilog
    delete factory;
end;
```

The following options are specific to the WebSphere MQ transport.

Keyword	Description
Model	Model=*model-queue-name*
MQSERVER	MQSERVER=*mq-server-setting*. For details, see WebSphere MQ Clients documentation (csqzaf09.pdf); for example, **MQSERVER='CHANNEL/TCP/server.com(32001)'**

JadeMetadataAnalyzer Class

The **JadeMetadataAnalyzer** class encapsulates the behavior required to analyze JADE metadata. For details about the constants and methods defined in the **JadeMetadataAnalyzer** class, see "JadeMetadataAnalyzer Class Constants" and "JadeMetadataAnalyzer Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeMetadataAnalyzer Class Constants

The **AccessCheckFailed** constant is the compiler error code (*6327 - Validation access check failed*) returned by the **validateMethod** method when an access check callback returns a fail status. The **JadeMetadataAnalyzer** class constants listed in the following table define the names of statements that can be passed to the **canAccessStatement** method.

Constant	Integer or String Value
AccessCheckFailed	6327
Statement_AbortTransaction	abortTransaction
Statement_AbortTransientTran	abortTransientTransaction
Statement_BeginLoad	beginLoad
Statement_BeginLock	beginLock
Statement_BeginTransaction	beginTransaction
Statement_BeginTransientTran	beginTransientTransaction
Statement_Break	break
Statement_Call	call
Statement_CommitTransaction	commitTransaction
Statement_CommitTransientTran	commitTransientTransaction
Statement_Continue	continue
Statement_Create	create
Statement_Delete	delete
Statement_EndLoad	endLoad
Statement_EndLock	endLock
Statement_Foreach	foreach
Statement_If	if
Statement_ImportMethod	importMethod
Statement_InheritMethod	inheritMethod
Statement_OnException	on
Statement_RaiseException	raise
Statement_Read	read

Constant	Integer or String Value
Statement_Return	return
Statement_Terminate	terminate
Statement_While	while
Statement_Write	write

JadeMetadataAnalyzer Methods

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

Method	Description
canAccessSchemaEntity	Checks the access to each schema entity referenced in the method being validated
canAccessStatement	Checks the access to each statement referenced in the method being validated
validateMethod	Validates method source code using user-definable security checks

canAccessSchemaEntity

Signature `canAccessSchemaEntity(schemaEntity: SchemaEntity;`
`errorMessage: String output): Boolean;`

The **canAccessSchemaEntity** method of the **JadeMetadataAnalyzer** class checks the access to each schema entity (class, method, property, and so on) referenced in the method being validated. The schema entity object is specified in the **schemaEntity** parameter.

This method returns **true** if the access is allowed or it returns **false** if the access is not allowed. If the access is not allowed, an error message may be returned in the **errorMessage** parameter and this message is then returned in the **accessErrorMessage** parameter of the **validateMethod** method. The **canAccessSchemaEntity** method returns **true** by default.

To implement user-defined access checking of schema entities, reimplement this method in a user subclass of the **JadeMetadataAnalyzer** class.

The following example checks that input method source code does not reference the **Employee** class or the **deposit** method of the **Account** class.

```
canAccessSchemaEntity(se: SchemaEntity; err: String output): Boolean;
begin
    if se.isKindOf(Class) then
        if se.name = 'Employee' then
            err := 'access to class is not allowed';
            return false;
        endif;
    elseif se.isKindOf(Method) then
        if se.Method.schemaType.name = 'Account' and se.name = 'deposit' then
            err := 'access to method is not allowed';
            return false;
        endif;
    endif;
```

```
    return true;
end;
```

canAccessStatement

Signature
```
canAccessStatement(statement:    String;
                   errorMessage: String output): Boolean;
```

The **canAccessStatement** method of the **JadeMetadataAnalyzer** class is called by the **validateMethod** method to check the access to each statement referenced in the method being validated. The name of the statement is specified in the **statement** parameter. Class constants define the names of statements that can be checked. (For details, see "JadeMetadataAnalyzer Class Constants", earlier in this chapter.)

The **canAccessStatement** method returns **true** if the access is allowed or it returns **false** if the access is not allowed.

If the access is not allowed, an error message may be returned in the **errorMessage** parameter and this message is then returned in the **accessErrorMessage** parameter of the **validateMethod** method. The **canAccessStatement** method returns **true** by default.

To implement user-defined access checking of statements, reimplement this method in a user subclass of the **JadeMetadataAnalyzer** class. The following example shows the **canAccessStatement** method reimplemented in a subclass of the **JadeMetadataAnalyzer** class to check that input method source code does not reference the **beginTransaction** instruction.

```
canAccessStatement(statement: String; errorMessage: String output): Boolean;
begin
    if statement = Statement_BeginTransaction then
        errorMessage := 'transactions are not allowed';
        return false;
    endif;
    return true;
end;
```

validateMethod

Signature
```
validateMethod(source:             String;
               schemaType:         Type;
               schema:             Schema;
               errorCode:          Integer output;
               errorPosition:      Integer output;
               errorLength:        Integer output;
               accessErrorMessage: String output) final;
```

The **validateMethod** method of the **JadeMetadataAnalyzer** class validates method source code using user-definable security checks. It calls the compiler to check the syntax of the source code and invokes callbacks to allow you to check the access to each statement and schema entity referenced in the method.

The **validateMethod** method parameters are listed in the following table.

Parameter	Description
source	The method source code to be validated.
schemaType	The owner type, or owner root type, of the method (that is, the type of the method receiver)

Parameter	Description
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.
errorCode	The error code returned by the compiler. A value of zero (**0**) indicates that the method was successfully validated.
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.
accessErrorMessage	The error message returned by the callback method when the access is not valid.

The method in the following example calls the **validateMethod** method to validate method source.

```
checkMethod(source: String; schemaType: Type);
vars
    analyzer      : MyAnalyzer;
    err, pos, len : Integer;
    msg           : String;
begin
    create analyzer;
    analyzer.validateMethod(source, schemaType, currentSchema, err, pos,
                            len, msg);
    if err = 0 then
        write 'method validated successfully';
    else
        write 'method validation failed - ' & process.getErrorText(err);
        if err = JadeMetadataAnalyzer.AccessCheckFailed then
            write msg;
        endif;
    endif;
epilog
    delete analyzer;
end;
```

JadeMethodContext Class

The **JadeMethodContext** class provides a framework to enable a task to be split into subtasks, which are processed in parallel. A subtask is essentially a method call that is to be executed asynchronously by a worker application. The **JadeMethodContext** instance is configured with the method call, the receiver, any required parameters, and the name of a worker application. A pool of one or more worker applications with the specified name must be running, to handle any asynchronous method calls.

When the **invoke** method is called on the **JadeMethodContext** instance, a request is queued and the method call is executed by the first worker application that becomes available. The queuing of **JadeMethodContext** invocation requests is handled internally by the JADE messaging framework.

The other way to execute a subtask in parallel with the main process is to use the **startApplication**, **startAppMethod**, **startApplicationWithParameter**, or **startApplicationWithString** method of the **Application** class. However, the applications started in this way would have to be specifically designed to carry out the subtask and would carry a greater overhead, whereas the asynchronous worker applications can carry out *any* subtask because the details of the method call are in the **JadeMethodContext** instance.

The main advantage of the **JadeMethodContext** approach is that the application that invokes the asynchronous method calls in the worker processes can more easily *rendezvous* with its worker processes; that is, obtain the results from the worker processes and know when it is safe to continue. The **waitForMethods** method on the **Process** class enables an application to wait for the completion of the asynchronous method calls and to obtain the results without having to return to an idle state. The alternative approach using a **startApplication** method to launch specifically designed applications would require the application to enter an idle state and wait for and handle notifications that the subtasks have been completed.

The **JadeMethodContext** class is also used to implement asynchronous Web service method calls.

For details about making asynchronous method calls, see in Chapter 16, "Using Asynchronous Method Calls", of the *JADE Developer's Reference*. For details about constants, properties, and methods defined in the **JadeMethodContext** class, see "JadeMethodContext Class Constants", "JadeMethodContext Properties", and "JadeMethodContext Methods", later in this section.

Inherits From: Object

Inherited By: (None)

JadeMethodContext Class Constants

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

Constant	Integer Value
State_Broken	4
State_Completed	8
State_Initialised	1
State_Preparing	3
State_Processing	6
State_TimedOut	9
State_Unused	0

JadeMethodContext Properties

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

Property	Description
state	Progress of the invocation of the asynchronous method call
tag	Contains an **Integer** value to distinguish the receiver from other **JadeMethodContext** instances
timeout	Time in milliseconds that the application waits for the asynchronous method call to complete
workerAppName	Name of a worker application that is to handle the asynchronous method call

state

Type: Integer

The read-only **state** property of the **JadeMethodContext** class contains a value that indicates the progress of the invocation of the asynchronous method call.

The values for this property, represented by **JadeMethodContext** class constants, are listed in the following table.

Constant	Integer Value
State_Broken	4
State_Completed	8
State_Initialised	1
State_Preparing	3
State_Processing	6
State_TimedOut	9
State_Unused	0

tag

Type: Integer

The **tag** property of the **JadeMethodContext** class contains an **Integer** value that can be used to distinguish the receiver from other **JadeMethodContext** instances.

Use this property if you need to identify the **JadeMethodContext** instance that is handling a specific asynchronous method call.

timeout

Type: Integer

The **timeout** property of the **JadeMethodContext** class contains the time in milliseconds that the application that invokes the asynchronous method request waits for the method to complete and a result to be delivered.

After the timeout period, the request continues processing until it is completed but the reply is discarded.

workerAppName

Type: String

The **workerAppName** property of the **JadeMethodContext** class contains the name of a worker application that is to handle the asynchronous method call specified by the **JadeMethodContext** instance.

The asynchronous method call is queued and handled by the first worker application with a name matching the value of the **workerAppName** property that becomes available.

JadeMethodContext Methods

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

Method	Description
getErrorNumber	Returns the error number of an exception that occurs processing the asynchronous method call
getErrorText	Returns the error text of an exception that occurs processing the asynchronous method call
getReturnValue	Returns the value that was returned from the worker application executing the asynchronous method call
getTimestamps	Outputs timestamp and queue depth information relating to the asynchronous method call
initialize	Initializes the receiver so that it can be used for another asynchronous method call
invoke	Queues a request for the execution of an asynchronous method call to a worker application
isComplete	Returns **true** if the request has completed or timed out
isProcessing	Returns **true** if the request is queued or processing
isTimedOut	Returns **true** if the request times out before it completes

getErrorNumber

Signature `getErrorNumber(): Integer;`

The **getErrorNumber** method of the **JadeMethodContext** class returns the error number of the exception if one occurs during the processing of the asynchronous method call by the worker application. If an exception does not occur, the method returns zero (**0**).

getErrorText

Signature `getErrorText(): String;`

The **getErrorText** method of the **JadeMethodContext** class returns the error text of the exception if one occurs during the processing of the asynchronous method call by the worker application. If an exception does not occur, the method returns an empty string.

getReturnValue

Signature `getReturnValue(): Any;`

The **getReturnValue** method of the **JadeMethodContext** class returns the value that was returned from the worker application executing the asynchronous method call.

You can typecast the returned value.

getTimestamps

Signature
```
getTimestamps(invokeTS:  TimeStamp output;
              beginTS:   TimeStamp output;
              finishTS:  TimeStamp output;
              harvestTS: TimeStamp output;
              qdepth:    Integer output);
```

The **getTimestamps** method of the **JadeMethodContext** class outputs timestamp and queue depth information relating to the processing of the asynchronous method call.

The parameters for the **getTimestamps** method are listed in the following table.

Parameter	Description
invokeTS	The timestamp value when the asynchronous method call was invoked using the **JadeMethodContext** instance
beginTS	The timestamp value when the worker application began processing the asynchronous method call
finishTS	The timestamp value when the worker application completed processing the asynchronous method call
harvestTS	The timestamp when the receiver was returned by the **waitForMethods** method and reported as complete
qdepth	The number of requests already queued when the asynchronous method call request was added to the queue

initialize

Signature
```
initialize() updating;
```

The **initialize** method of the **JadeMethodContext** class initializes the receiver so that it can be used to invoke another asynchronous method. The values of the **tag** and **workerAppName** properties are not changed.

invoke

Signature
```
invoke(target: PseudoMethodCallType input) updating;
```

The **invoke** method of the **JadeMethodContext** class queues a request for the execution of an asynchronous method call to a worker application. The formal **target** parameter is a placeholder for the receiver object reference, the called method, and the list of parameters (if any) for the called method.

The following code fragment shows the use of the **invoke** method to make an asynchronous call to execute the **getHistory** method on an instance of the **Customer** class.

```
vars
    cust : Customer;
    date : Date;
begin
    ...
    jadeMethodContext.invoke(cust, getHistory, date);
```

The method signature of the **getHistory** method in the **Customer** class is as follows.

```
Customer::getHistory(startDate: Date): String;
```

isComplete

Signature `isComplete(): Boolean;`

The **isComplete** method of the **JadeMethodContext** class returns **true** if the request has completed or timed out; otherwise it returns **false**.

isProcessing

Signature `isProcessing(): Boolean;`

The **isProcessing** method of the **JadeMethodContext** class returns **true** if the request is queued or processing; otherwise it returns **false**.

isTimedOut

Signature `isTimedOut(): Boolean;`

The **isTimedOut** method of the **JadeMethodContext** class returns **true** if the request times out before it completes.

After the request times out, the asynchronous method call continues processing until it is completed but the reply is discarded.

JadeMultiWorkerTcpConnection Class

The **JadeMultiWorkerTcpConnection** class provides an interface to a TCP/IP connection managed by the **JadeMultiWorkerTcpTransport** class. For more details, see the **JadeMultiWorkerTcpTransport** class.

Note The **JadeMultiWorkerTcpConnection** class cannot be subclassed. It has a lifetime of transient (**transientAllowed**); that is, you cannot create persistent or shared transient instances of the class.

For details about constants, properties, and methods defined in the **JadeMultiWorkerTcpConnection** class, see "JadeMultiWorkerTcpConnection Class Constants", "JadeMultiWorkerTcpConnection Properties", and "JadeMultiWorkerTcpConnection Methods", later in this section.

Inherits From: Object

Inherited By: (None)

JadeMultiWorkerTcpConnection Class Constants

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

Constant	Integer Value
CloseRequested	7
Closing	8
Connected	3
Opening	2

JadeMultiWorkerTcpConnection Properties

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

Property	Description
connectionId	Contains the unique number assigned to the client connection when it opens
currentEventBusyElapsed	Contains the elapsed time to date (in microseconds) processing the current event
currentEventBusyWhen	Contains the timestamp when processing of the current event started
currentEventQueuedElapsed	Contains the elapsed time (in microseconds) the current event spent waiting to be processed
currentEventQueuedWhen	Contains the timestamp when the current event was queued
fillReadBuffer	Specifies whether the read buffer is filled
idleTimeout	Contains the maximum number seconds to wait for idle connections to have no input before causing an **IdleTimeout** connection event
keepAssigned	the Specifies whether the connection is kept assigned after exiting from the callback method
localAddress	Contains address of the local side of the connection

Property	Description
localPortnumber	Contains the port number of the local side of the connection
remoteAddress	Contains the address of the remote side of the connection
remotePortnumber	Contains the port number of the remote side of the connection
state	Contains the current state of the connection
timeout	Contains the number of seconds to wait for a **readBinary** or **writeBinary** method call to complete
userObject	Contains user-supplied connection-related information between event callbacks
userState	Contains a state value between event callbacks

connectionId

Type: Integer

The read-only **connectionId** property of the **JadeMultiWorkerTcpConnection** class contains the unique number assigned to the connection when it opens.

The value is unique to this transport group. Combine the value of this property and the result of a **getGroupId** method call to obtain a JADE node unique key over all transport groups.

currentEventBusyElapsed

Type: Integer

The read-only **currentEventBusyElapsed** property of the **JadeMultiWorkerTcpConnection** class contains the elapsed time to date (in microseconds) processing the current event.

currentEventBusyWhen

Type: TimeStamp

The read-only **currentEventBusyWhen** property of the **JadeMultiWorkerTcpConnection** class contains the timestamp when processing of the current event started.

currentEventQueuedElapsed

Type: Integer

The read-only **currentEventQueuedElapsed** property of the **JadeMultiWorkerTcpConnection** class contains the elapsed time (in microseconds) that the current event spent waiting to be processed.

currentEventQueuedWhen

Type: TimeStamp

The read-only **currentEventQueuedWhen** property of the **JadeMultiWorkerTcpConnection** class contains the timestamp when the current event was queued.

fillReadBuffer

Type: Boolean

Set the **fillReadBuffer** property of the **JadeMultiWorkerTcpConnection** class to **true** to specify that the **readBinary** method does not return until the requested length of the data has been received.

If the **fillReadBuffer** property is set to **false** (the default), the **readBinary** method returns as soon as any data is received for the connection. The **length** parameter of the **readBinary** method is therefore treated as a maximum buffer size.

idleTimeout

Type: Integer

The **idleTimeout** property of the **JadeMultiWorkerTcpConnection** class contains the maximum number seconds to wait for idle connections to have no input before causing an **IdleTimeout** connection event.

If you assign a value greater than zero (**0**) and less than **5**, the value is increased to **5**, making the minimum timeout used **5** seconds. The default value of zero (**0**) indicates that idle connections do not time out.

keepAssigned

Type: Boolean

The **keepAssigned** property of the **JadeMultiWorkerTcpConnection** class specifies whether the worker process can assume responsibility for deleting the temporary object and retain ownership of the connection after exiting from the callback method, by setting this property to **true**.

Set this property to **true** if you want to prevent the automatic deletion of the **JadeMultiWorkerTcpConnection** object when the current callback method exits, leaving the connection assigned to this worker. The default value is **false**.

Note No event callback methods are invoked for this connection until the temporary object is deleted and the connection is unassigned.

localAddress

Type: String

The read-only **localAddress** property of the **JadeMultiWorkerTcpConnection** class contains the local address of the connection; for example, **"172.16.0.1"**.

localPortnumber

Type: Integer

The read-only **localPortnumber** property of the **JadeMultiWorkerTcpConnection** class contains the local port number of the connection, in the range **1** through **65535**.

remoteAddress

Type: String

The read-only **remoteAddress** property of the **JadeMultiWorkerTcpConnection** class contains the remote address of the connection; for example, **"172.16.1.2"**onnection.

remotePortnumber

Type: Integer

The read-only **remotePortnumber** property of the **JadeMultiWorkerTcpConnection** class contains the remote port number of the connection.

state

Type: Integer

The read-only **state** property of the **JadeMultiWorkerTcpConnection** class contains the state of the connection.

Methods can be called only in the appropriate state and they can cause the connection state to change. The values of the **state** property are listed in the following table.

JadeMultiWorkerTcpConnection Class Constant	Integer Value
CloseRequested	7
Closing	8
Connected	3
Opening	2

timeout

Type: Integer

The **timeout** property of the **JadeMultiWorkerTcpConnection** class contains the number of milliseconds the connection waits for a **readBinary**, **readUntil**, or **writeBinary** method call to complete before raising an exception.

The initial value of the **timeout** property is obtained by converting the value of the **ReadTimeout** parameter (which is in seconds) in the [JadeOdbcServer] section of the JADE initialization file. If there is a separate XML application configuration file, the value is obtained by converting the value of the **read_timeout** parameter in that file.

The timeout value remains active for these operations until you reset the value in your application for that transient instance of the connection object. The default value of zero (**0**) indicates that the operation does not time out.

userObject

Type: Object

The **userObject** property of the **JadeMultiWorkerTcpConnection** class contains a reference to an object that you can associate with the connection between event callbacks.

You must set the value of this property to a shared transient or a persistent object, as it must be visible to other workers.

The default value is **null**.

Note To prevent an object leak, it is your responsibility to delete this object, if required, in your implementation of the **closedEvent** method in the receiver class.

userState

Type: Integer

The **userState** property of the **JadeMultiWorkerTcpConnection** class contains connection-related state Integer value between event callbacks.

The default value is zero (**0**).

JadeMultiWorkerTcpConnection Methods

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

Method	Description
bindToAssignedWorker	Binds this connection to the currently assigned worker (that is, to the current JADE process)
bindToWorkerId	Attempts to bind the connection to the worker process with the specified worker identifier
causeUserEvent	Queues a user event for the connection with the specified tag value
close	Requests that the connection be closed locally
getAssignedWorkerId	Returns the worker to which the connection is assigned
getBoundWorkerId	Returns the worker identifier (the current or another worker process) to which the connection is bound
getFullName	Returns a string containing the full name of the associated **JadeMultiWorkerTcpTransport** object
getGroupId	Returns the identifier of the transport group to which the connection belongs
getLocalHostname	Returns a string containing the generic host name associated with the **localAddress** property
getRemoteHostname	Returns a string containing the generic host name associated with the **remoteAddress** property
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
readUntil	Reads data from the connection and returns when the specified delimiter is found in the data stream
unbind	Unbinds the connection if it is currently bound
writeBinary	Writes binary data to the connection and returns when the operation is complete
writeBinaryAndFile	Writes binary data and the file to the connection, writes the specified section of the file, and then returns when the operation is complete

bindToAssignedWorker

Signature `bindToAssignedWorker();`

The **bindToAssignedWorker** method of the **JadeMultiWorkerTcpConnection** class binds this connection to the currently assigned worker (that is, to the current JADE process).

All subsequent events for the connection are delivered to the currently assigned worker.

bindToWorkerId

Signature `bindToWorkerId(workerId: Integer): Boolean;`

The **bindToWorkerId** method of the **JadeMultiWorkerTcpConnection** class attempts to bind the connection to the worker process specified in the **workerId** parameter. This method returns **true** if the bind operation is successful or it returns **false** if the worker identifier is unknown. If successful, all subsequent events for the connection are delivered to the specified worker.

causeUserEvent

Signature `causeUserEvent(tag: Integer);`

The **causeUserEvent** method of the **JadeMultiWorkerTcpConnection** class queues a **UserEvent** client connection event for the connection with the value specified in the **tag** parameter.

As there can be one unprocessed **UserEvent** event only for a connection, a subsequent call of this method overwrites an unprocessed user event.

close

Signature `close();`

The **close** method of the **JadeMultiWorkerTcpConnection** class closes a local connection to a remote application and returns when the connection is closed. This method can be called when the connection is in any state.

A connection **Closed** event is queued when the connection has been closed.

getAssignedWorkerId

Signature `getAssignedWorkerId(): Integer;`

The **getAssignedWorkerId** method of the **JadeMultiWorkerTcpConnection** class returns the identifier of the worker (that is, the current JADE process) to which the connection is assigned from the **JadeMultiWorkerTcpTransport** object.

getBoundWorkerId

Signature `getBoundWorkerId(): Integer;`

The **getBoundWorkerId** method of the **JadeMultiWorkerTcpConnection** class returns the worker identifier (the current or another worker process) to which the connection is bound.

getFullName

Signature `getFullName(): String;`

The **getFullName** method of the **JadeMultiWorkerTcpConnection** class returns a string containing the full name of the associated **JadeMultiWorkerTcpTransport** object.

getGroupId

Signature `getGroupId(): Integer;`

The **getGroupId** method of the **JadeMultiWorkerTcpConnection** class returns the identifier of the transport group to which this connection belongs.

getLocalHostname

Signature `getLocalHostname(): String;`

The **getLocalHostname** method of the **JadeMultiWorkerTcpConnection** class returns a string containing the generic host name (computer name) associated with the **localAddress** property.

This method returns a null string (**""**) if the Domain Name Server (DNS) look-up fails or the local host name cannot be resolved.

getRemoteHostname

Signature `getRemoteHostname(): String;`

The **getRemoteHostname** method of the **JadeMultiWorkerTcpConnection** class returns a string containing the generic host name (computer name) associated with the **remoteAddress** property.

This method returns a null string (**""**) if the DNS look-up fails or the remote host name cannot be resolved.

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **JadeMultiWorkerTcpConnection** class reads binary data from the connection and returns the binary value 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 **fillReadBuffer** property.

This method can be called only when the value of the **state** property is **Connected**.

If the value of the **timeout** property is not zero (**0**), the read operation can terminate with an exception and return less than the specified length.

readUntil

Signature
```
readUntil(delimiter: Binary;
          maxLength: Integer): Binary;
```

The **readUntil** method of the **JadeMultiWorkerTcpConnection** 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 **state** property is **Connected**.

If the value of the **timeout** property is not zero (**0**), the read operation can terminate with an exception and return less than the specified length.

Note The delimiter is not included in the returned data.

unbind

Signature `unbind();`

The **unbind** method of the **JadeMultiWorkerTcpConnection** class unbinds the connection if it is currently bound. This method is ignored if the connection is not currently bound. A connection can be unbound only by the worker to which it is currently bound.

writeBinary

Signature `writeBinary(buffer: Binary);`

The **writeBinary** method of the **JadeMultiWorkerTcpConnection** class writes binary data to the connection and returns when the operation is complete. This method can be called only when the value of the **state** property is **Connected**. If the value of the **timeout** property is not zero (**0**), the write operation can terminate with an exception and without writing all of the contents of the buffer.

Messages are sent in the order that the connection object receives them.

writeBinaryAndFile

Signature
```
writeBinaryAndFile(buffer: Binary;
                   fyle:   File;
                   offset: Decimal;
                   length: Integer);
```

The **writeBinaryAndFile** method of the **JadeMultiWorkerTcpConnection** class writes binary data to the connection, writes the specified section of the file specified in the **fyle** parameter, and then returns when the operation is complete. The value of the **buffer** parameter can be empty.

The file specified in the **fyle** parameter must be open for reading.

Use the **offset** parameter to specify a file offset as a decimal value and the **length** parameter to specify the number of bytes to read and send.

This method can be called only when the value of the **state** property is **Connected**.

Note Specify the file **offset** and **length** parameter values in octets.

If you want to send the entire file, specify a value of zero (**0**) for the **offset** parameter and a value of minus 1 (**-1**) for the **length** parameter.

JadeMultiWorkerTcpTransport Class

The **JadeMultiWorkerTcpTransport** class provides an interface for sharing the messages arriving on client connections associated with a single listen TCP/IP connection among a pool of server JADE applications. This enables server-style applications to share a workload generated by a large number of clients among a number of worker processes. No master or manager application is required.

Notes You cannot create persistent or shared transient instances of this class or its subclasses.

All JADE worker processes must reside in the same JADE node as the TCP/IP listen connection.

The following terms are used in a JADE multiple worker TCP/IP connection environment.

- The *transport group* encompasses the client connection pool, the worker pool, the listen connection, and so on.
- The *connection pool* encompasses the group of client connections associated with the transport group.
- The *worker pool* encompasses the group of JADE processes that are linked to the transport group.

Each transport group has a unique full name, which is a string containing the **listenHostname** property value, a colon (**:**) character, and the **listenPortnumber** property value. The listen connection uses the **listenHostname** and **listenPortnumber** property values for binding its local address and local port number, respectively. A JADE process joins a transport group by creating an instance of the **JadeMultiWorkerTcpTransport** class, setting the appropriate properties (at a minimum, the **listenPortnumber** property), and then calling the **beginListening** method.

If a JADE process attempts to join an unknown transport group, the requested transport group is created and that process joins it as the first worker. During creation, the group automatically creates the listen connection and opens it.

Any JADE process in a node can join an existing transport group, provided that it resides in the same JADE node and it knows the full name of the group (that is, the values of the **listenHostname** and **listenPortnumber** properties) and it uses exactly the same **JadeMultiWorkerTcpTransport** subclass.

When a process attempts to join a transport group, the **validateServerProcess** method is invoked. To perform additional basic security checks and reject the join attempt if necessary, reimplement this method on a **JadeMultiWorkerTcpTransport** subclass.

A worker leaves a transport group by deleting its instance of **JadeMultiWorkerTcpTransport** that it used to join the transport group. When the last worker leaves the transport group, the listen connection and any client connections are closed and the group is then deleted.

New workers are added to the end of the idle worker list and are the last to be woken.

For details about:

- Connection assignment, connection binding, event handling, and reading and writing data, see "Connection Assignment", "Connection Binding", "JADE MultiWorker Events", and "Reading and Writing Data", respectively, in the following subsections
- The class constants, properties, and methods defined in the **JadeMultiWorkerTcpTransport** class, see "JadeMultiWorkerTcpTransport Class Constants", "JadeMultiWorkerTcpTransport Properties", and "JadeMultiWorkerTcpTransport Methods", later in this chapter.

Inherits From: Object

Inherited By: (None)

Connection Assignment

A connection is assigned to a worker when that worker removes it from the connection-ready queue to handle an event for that connection.

A temporary transient **JadeMultiWorkerTcpConnection** object is created and passed as a parameter to the appropriate event callback method.

By default, when the worker has finished handling the event and it exits from the callback method, the temporary connection object is automatically deleted, which causes the client connection to be unassigned and added back to the idle connection list to await the next event.

The worker process can assume responsibility for deleting the temporary object and retain ownership of the connection after exiting from the callback method, by setting the **keepAssigned** property to **true**.

A worker can have multiple client connections concurrently assigned to it.

A client connection can be assigned only to a single worker and only that worker can manipulate that connection (for example, read data, write data, or modify property values).

While a connection is assigned to a worker, no further event callback methods are invoked for that connection but events are queued until the connection is unassigned.

If a worker process terminates holding assigned connections, those connections are closed.

If a connection callback is unwound by an unhandled exception, that connection is closed.

Connection Binding

A client connection can be bound to a single worker process. Events for a bound connection are directed to that bound worker. Events for unbound connections are handled by any worker.

Events for bound connections have higher priority than those of unbound connections.

A connection can be bound and unbound multiple times. A connection can be:

- Bound to the current process
- Bound to a worker identifier (the current or another worker process)
- Unbound from the current process

For unbound events, the most-recently idle worker is always the first woken.

If a worker process terminates holding bound connections, those connections are closed.

JADE MultiWorker Events

When data arrives on a client connection, the transport group manager adds the connection to the end of the connection-ready queue and wakes the worker that most recently went idle. When the worker wakes, it removes the connection at the front of the ready queue and invokes the appropriate event callback method.

The connection-ready queue is first-in-first-out, ensuring that a message is processed from each connection in turn. The worker-idle list is last-in-first-out, to make best use of processor and database caches.

As incoming messages from unbound connections are directed to the next idle worker process, a specific client connection can be handled by more than one worker. This means that session data for unbound connections must be held in persistent or shared transient objects (for details, see the **JadeMultiWorkerTcpConnection** class **userObject** property).

The event groups are connection events and management events. For details, see the following subsections.

Connection Events

The connection events, represented by callback methods in the **JadeMultiWorkerTcpTransportIF** interface implemented as a mapping to classes in your user schema, are as follows.

- **ReadReadyEvent**

  The connection has input data available to read.

- **OpenedEvent**

  The connection has been established. The connection state is **Opening** until the event callback method exits.

- **ClosedEvent**

  A remote or a local close has been performed on this connection. The connection state is **ClosePending** until the event callback method exits.

- **UserEvent**

  This event is queued by the **JadeMultiWorkerTcpConnection** class **causeUserEvent** method and the **JadeMultiWorkerTcpTransport** class **causeUserEventOnConnId** method.

- **ConnectionEvent**

  This multiplex callback event that delivers the **IdleTimeout** event type. This event is queued when a connection has received no input for the time specified in the **idleTimeout** property of that connection.

If a connection callback method is unwound by an unhandled exception, that connection is closed.

Management Events

The **ManagementEvent** is a multiplex callback that delivers the following event types.

- **QueueDepthExceeded**

  The queue depth exceeded management event allows additional workers to be started when the work load is exceeds the capacity of the current worker pool.

- **WorkerIdleTimeout**

  The idle worker timeout management event allows excess workers to terminate.

Use these events in conjunction with the **queueDepthLimit**, **queueDepthLimitTimeout**, and **workerIdleTimeout** properties in the **JadeMultiWorkerTcpTransport** class to dynamically start and stop worker processes to match the workload.

Reading and Writing Data

Data can be read and written only:

- On the temporary transient **JadeMultiWorkerTcpConnection** object passed to the event callback methods.
- By the currently assigned worker process.
- On connections whose **state** property value is **Connected**.

There are no asynchronous (that is, callback) read and write variants.

JadeMultiWorkerTcpTransport Class Constants

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

Constant	Integer Value
Lstate_Broken	9
Lstate_Closed	1
Lstate_Listening	3
Lstate_Unattached	0
Notify_Continuous	0
Notify_OneShot	1

JadeMultiWorkerTcpTransport Properties

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

Property	Contains the …
listenHostname	Host name of the listening connection
listenPortnumber	Port number of the listening connection
listenState	State of the listen connection
queueDepthLimit	Connection ready queue size required to trigger the **QueueDepthExceeded** management event
queueDepthLimitTimeout	Number of seconds that the connection ready queue size must remain greater than the value of the **queueDepthLimit** property before triggering the **QueueDepthExceeded** management event
statisticsLogInterval	Number of seconds at which worker and group to-date statistics are written to the **jommsg.log** file
userGroupObject	Reference to an object that you can associate with the transport group
workerId	Unique sequential number assigned on the first **beginListening** method call on this **JadeMultiWorkerTcpTransport** object
workerIdleTimeout	Number of seconds after the last event handled by this process that the **WorkerIdleTimeout** management event is queued for this process

listenHostname

Type: String[128]

The **listenHostname** property of the **JadeMultiWorkerTcpTransport** class contains the local host name or IP address of the listening connection.

The default null value (**""**) indicates a value of **"0.0.0.0"**.

This property is read-only unless the value of the **listenState** property is **Lstate_Unattached** (**0**).

listenPortnumber

Type: Integer

The **listenPortnumber** property of the **JadeMultiWorkerTcpTransport** class contains the port number of the listening connection. Valid values are in the range **1** through **65535**.

This property is read-only unless the value of the **listenState** property is **Lstate_Unattached** (**0**).

listenState

Type: Integer

The read-only **listenState** property of the **JadeMultiWorkerTcpTransport** class contains the state of the listen connection.

The values for this property, represented by **JadeMultiWorkerTcpTransport** class constants, are listed in the following table.

Class Constant	Integer Value
Lstate_Broken	9
Lstate_Closed	1
Lstate_Listening	3
Lstate_Unattached	0

queueDepthLimit

Type: Integer

The **queueDepthLimit** property of the **JadeMultiWorkerTcpTransport** class contains the connection ready queue size required to trigger a **QueueDepthExceeded** management event.

The default value of zero (**0**) indicates that a **QueueDepthExceeded** management event is never queued.

queueDepthLimitTimeout

Type: Integer

The **queueDepthLimitTimeout** property of the **JadeMultiWorkerTcpTransport** class contains the number of seconds that the connection ready queue size must remain greater than the value of the **queueDepthLimit** property before triggering a **QueueDepthExceeded** management event.

The default value of zero (**0**) indicates that a **QueueDepthExceeded** management event is never queued.

As values greater than zero (**0**) but less than 5 are increased to 5, the effective minimum timeout is 5 seconds.

statisticsLogInterval

Type: Integer

The **statisticsLogInterval** property of the **JadeMultiWorkerTcpTransport** class contains the number of seconds at which worker and group to-date statistics are written to the **jommsg.log** file.

The default value of zero (**0**) indicates that statistics are never written to the JADE message log file.

userGroupObject

Type: Object

The **userGroupObject** property of the **JadeMultiWorkerTcpTransport** class contains a reference to an object that you can associate with the transport group between event callbacks.

You must set the value of this property to a shared transient or a persistent object, as it must be visible to other workers.

The default value is **null**.

Note To prevent an object leak, it is your responsibility to delete this object, if required, in your implementation of the **closedEvent** method in the receiver class.

workerId

Type: Integer

The read-only **workerId** property of the **JadeMultiWorkerTcpTransport** class contains the unique sequential number assigned on the first **beginListening** method call on this **JadeMultiWorkerTcpTransport** object.

workerIdleTimeout

Type: Integer

The **workerIdleTimeout** property of the **JadeMultiWorkerTcpTransport** class contains the number of seconds after the last event handled by this process that a **WorkerIdleTimeout** management event is queued for this process.

The idle event is queued each time the worker is idle for more than the specified timeout period.

The default value of zero (**0**) indicates that **WorkerIdleTimeout** management events are not queued.

As values greater than zero but less than ten are increased to **10**, the effective minimum timeout is **10** seconds.

JadeMultiWorkerTcpTransport Methods

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

Method	Description
beginListening	Attaches to the transport group and creates a new group if the specified group does not exist

Method	Description
cancelNotify	Stops callbacks being queued to this process
causeUserEventOnConnId	Queues a user event for the specified connection
countAssignedConnections	Returns the number of connections currently assigned to this worker
countBoundConnections	Returns the number of connections currently bound to this worker
countIdleWorkers	Returns the number of idle worker processes sharing the pool
countQueuedConnections	Returns the number of connections that are unassigned and have queued events
countWorkers	Returns the number of worker processes sharing the pool
getFullName	Returns the full name of this transport group (that is, the values of the **listenHostname** and **listenPortnumber** properties, separated by a colon character)
getGroupId	Returns the unique number assigned to each transport group when it is created by the first worker to call the **beginListening** method with a new full name
getGroupStatistics	Populates the specified **JadeDynamicObject** with properties containing the to-date values of statistics captured by this transport group
getMyStatistics	Populates the specified **JadeDynamicObject** with properties containing the to-date values of statistics captured by the receiving worker
notifyEventsAsync	Notifies the implementing **JadeMultiWorkerTcpTransportIF** receiver of asynchronous events
stopListening	Closes the listen connection to prevent further client connections
validateServerProcess	Validates the server process

beginListening

Signature `beginListening();`

The **beginListening** method of the **JadeMultiWorkerTcpTransport** class attaches the calling process to the transport group and creates the group if it does not exist.

A worker can join an existing group only if:

- It has the same full name (that is, the values of the **localAddress** and **localPortnumber** properties)
- It uses exactly the same **JadeMultiWorkerTcpTransport** class (or subclass)
- The **validateServerProcess** method returns **true**, to ensure that this worker connection can be accepted

Subsequent calls to the **beginListening** method ensure that the listen connection is open (that is, it undoes a **stopListening** method call).

cancelNotify

Signature `cancelNotify();`

The **cancelNotify** method of the **JadeMultiWorkerTcpTransport** class stops event callbacks being queued to this process and removes this worker process from the idle worker queue.

causeUserEventOnConnId

Signature `causeUserEventOnConnId(connId: Integer; tag: Integer);`

The **causeUserEventOnConnId** method of the **JadeMultiWorkerTcpConnection** class queues a **UserEvent** client connection event for the connection specified in the **connId** parameter with the value specified in the **tag** parameter.

The request is ignored if the specified connection is unknown or closing.

countAssignedConnections

Signature `countAssignedConnections(): Integer;`

The **countAssignedConnections** method of the **JadeMultiWorkerTcpTransport** class returns the number of connections currently assigned to this worker.

countBoundConnections

Signature `countBoundConnections(): Integer;`

The **countBoundConnections** method of the **JadeMultiWorkerTcpTransport** class returns the number of connections currently bound to this worker.

countIdleWorkers

Signature `countIdleWorkers(): Integer;`

The **countIdleWorkers** method of the **JadeMultiWorkerTcpTransport** class returns the number of idle worker processes sharing the pool.

countQueuedConnections

Signature `countQueuedConnections(allBound: Integer output; allUnbound: Integer output);`

The **countQueuedConnections** method of the **JadeMultiWorkerTcpTransport** class returns the number connections that are unassigned and have queued events.

The **allBound** parameter is populated with the number of queued bound connections and the **allUnbound** parameter is populated with the number of queued unbound connections.

countWorkers

Signature `countWorkers(): Integer;`

The **countWorkers** method of the **JadeMultiWorkerTcpTransport** class returns the number of worker processes sharing the pool.

getFullName

Signature `getFullName(): String;`

The **getFullName** method of the **JadeMultiWorkerTcpTransport** class returns a string containing the full name of this transport group (that is, the values of the **listenHostname** and **listenPortnumber** properties, separated by a colon character).

getGroupId

Signature `getGroupId(): Integer;`

The **getGroupId** method of the **JadeMultiWorkerTcpTransport** class returns the unique number assigned to each transport group when it is created by the first worker to call the **beginListening** method with a new full name.

Each time a transport group is created in a node, it is given a different group identifier even if it has previously used a full name.

The **getGroupId** method returns zero (**0**) until a call to the **beginListening** method is successful.

getGroupStatistics

Signature
```
getGroupStatistics(which: Integer;
                   stats: JadeDynamicObject input);
```

The **getGroupStatistics** method of the **JadeMultiWorkerTcpTransport** class populates the **JadeDynamicObject** specified in the **stats** parameter with properties containing the to-date values of statistics captured for this transport group.

The value of the **which** parameter must be zero (**0**).

Properties with names that have a suffix of **time** represent elapsed time values measured in microseconds. You should assign these values to **Decimal[18]** (or wider) variables.

getMyStatistics

Signature
```
getMyStatistics(which: Integer;
                stats: JadeDynamicObject input);
```

The **getMyStatistics** method of the **JadeMultiWorkerTcpTransport** class populates the **JadeDynamicObject** specified in the **stats** parameter with properties containing the to-date values of statistics captured by the receiving worker.

The value of the **which** parameter must be zero (**0**).

Properties with names that have a suffix of **time** represent elapsed time values measured in microseconds. You should assign these values to **Decimal[18]** (or wider) variables.

notifyEventsAsync

Signature
```
notifyEventsAsync(recv:   JadeMultiWorkerTcpTransportIF;
                  option: Integer);
```

The **notifyEventsAsync** method of the **JadeMultiWorkerTcpTransport** class registers the receiver object on which the event callback methods will be invoked.

The value of the **recv** parameter must be an instance of a class that implements the **JadeMultiWorkerTcpTransportIF** interface.

The valid values for the option parameter are represented by the **JadeMultiWorkerTcpTransport** class constants listed in the following table.

Constant	Integer Value	Occurs…
Notify_Continuous	0	Continuously until disabled by a call to the **cancelNotify** method
Notify_OneShot	1	Once only

Use the **Notify_Continuous** value if you want to have callbacks automatically re-armed after a callback method exits. Conversely, use the **Notify_OneShot** value if you want to re-arm callbacks manually as part of each callback method. You can switch between these notification modes, if required.

stopListening

Signature `stopListening();`

The **stopListening** method of the **JadeMultiWorkerTcpTransport** class closes the listen connection to prevent further client connections.

To re-open the listen connection, call the **beginListening** method.

Established client connections are not affected by the state of the listen connection.

validateServerProcess

Signature `validateServerProcess(): Boolean;`

The **validateServerProcess** method of the **JadeMultiWorkerTcpTransport** class validates the server process.

This method is automatically called the first time the **beginListening** method is called by a process.

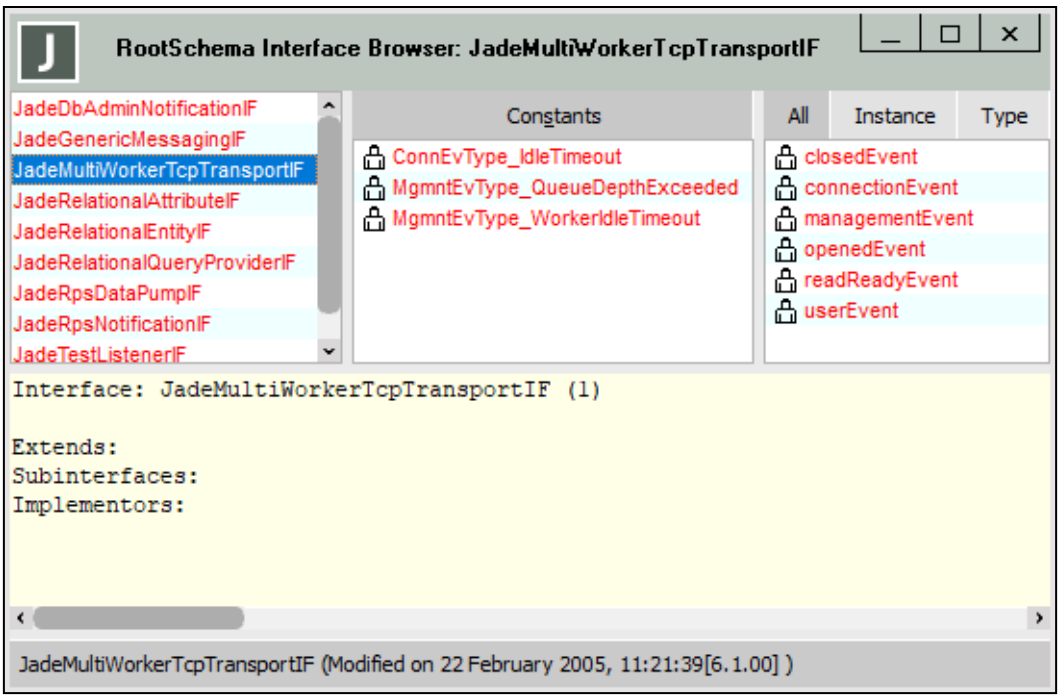
The default implementation always returns **true**. If this method returns **false**, the process calling the **beginListening** method receives an exception.

JadeMultiWorkerTcpTransportIF Interface

The **JadeMultiWorkerTcpTransportIF** interface, defined in the **RootSchema**, provides the definition of the event callback methods that you can implement in your user schema classes to handle management and connection event notifications.

For more details about connection events, see the **JadeMultiWorkerTcpConnection** class and **JadeMultiWorkerTcpTransport** class.

You can view the **JadeMultiWorkerTcpTransportIF** interface and its constants and 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 **JadeMultiWorkerTcpTransportIF** 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 **JadeMultiWorkerTcpTransportIF** interface constants and methods, see "JadeMultiWorkerTcpTransportIF Interface Constants" and "JadeMultiWorkerTcpTransportIF Interface Method Callback Signatures", in the following subsections.

JadeMultiWorkerTcpTransportIF Interface Constants

The constants provided by the **JadeMultiWorkerTcpTransportIF** interface are listed in the following table.

Constant	Integer Value
ConnEvType_IdleTimeout	1
MgmntEvType_QueueDepthExceeded	1
MgmntEvType_WorkerIdleTimeout	2

JadeMultiWorkerTcpTransportIF Interface Callback Method Signatures

The signatures of callback methods provided the **JadeMultiWorkerTcpTransportIF** interface are summarized in the following table.

Method	Callback method for the …
closedEvent	**ClosedEvent** connection event
connectionEvent	**ConnectionEvent** connection event
managementEvent	**ManagementEvent** transport event
openedEvent	**OpenedEvent** connection event
readReadyEvent	**ReadReadyEvent** connection event
userEvent	**UserEvent** connection event

closedEvent

Signature

```
closedEvent(listener: JadeMultiWorkerTcpTransport input;
            conn:     JadeMultiWorkerTcpConnection input);
```

When a connection closes, a **Closed** connection event is queued and is delivered to a worker process via the **closedEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object and the **conn** parameter with the connection that has closed.

connectionEvent

Signature

```
connectionEvent(listener: JadeMultiWorkerTcpTransport input;
                conn:     JadeMultiWorkerTcpConnection input;
                type:     Integer;
                info:     Any);
```

When a connection has been idle for the maximum number of seconds specified in the **JadeMultiWorkerTcpConnection** class **idleTimeout** property, an **IdleTimeout** connection event is queued and is delivered to a worker process via the **connectionEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object and the **conn** parameter with the connection. The **type** parameter is set to **ConnEvType_IdleTimeout** and the value of the **info** parameter is null.

managementEvent

Signature
```
managementEvent(listener: JadeMultiWorkerTcpTransport input;
                type:     Integer;
                info:     Any);
```

When a management event occurs, the **ManagementEvent** transport event is queued and is delivered to a worker process via the **managementEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object. The **type** parameter is set to **MgmntEvType_ WorkerIdleTimeout** or **MgmntEvType_QueueDepthExceeded** and the value of the **info** parameter is null.

openedEvent

Signature
```
openedEvent(listener: JadeMultiWorkerTcpTransport input;
            conn:     JadeMultiWorkerTcpConnection input);
```

When a connection opens, the **Opened** connection event is queued and is delivered to a worker process via the **openedEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object and the **conn** parameter with the connection that has opened.

readReadyEvent

Signature
```
readReadyEvent(listener: JadeMultiWorkerTcpTransport input;
               conn:     JadeMultiWorkerTcpConnection input);
```

When a ready event occurs, the **ReadReady** connection event is queued and is delivered to a worker process via the **readReadyEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object and the **conn** parameter with the connection that has the available data.

Note To avoid a continuous loop, your implementing callback method must read some data.

userEvent

Signature
```
userEvent(listener: JadeMultiWorkerTcpTransport input;
          conn:     JadeMultiWorkerTcpConnection input;
          tag:      Integer);
```

The **JadeMultiWorkerTcpConnection** class **causeUserEvent** method or the **JadeMultiWorkerTcpTransport** class **causeUserEventOnConnId** method queues a **UserEvent** connection event that is delivered via the **userEvent** callback method in your implementation of the **JadeMultiWorkerTcpTransportIF** interface.

The **listener** parameter is populated with the transport object and the **conn** parameter with the connection associated with the event.

The **tag** parameter is the **Integer** value that was passed to the **JadeMultiWorkerTcpConnection** class **causeUserEvent** method or **JadeMultiWorkerTcpTransport** class **causeUserEventOnConnId** method.

Only the most recent user event is delivered. When a new user event is queued, the previous undelivered user event is discarded.

JadePatchControlInterface Class

The **JadePatchControlInterface** class encapsulates the behavior required to dynamically access patch control information.

For details about the methods defined in the **JadePatchControlInterface** class, see "JadePatchControlInterface Methods", in the following subsection.

Inherits From: Object

Inherited By: (None)

JadePatchControlInterface Methods

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

Method	Description
closePatchNumber	Closes a specified patch number
getAllEntitiesForPatchNumber	Returns all entities updated for a specific patch number
getCheckedOutEntitiesForPatch	Returns the checked out methods for a specific patch number and delta identifier
getDeletedEntitiesForPatchNo	Returns the deleted entities for a specific patch number
getEntitiesForPatchNumber	Returns instances of subclasses of **SchemaEntity** updated for a specific patch number
getOpenPatchNumbers	Returns all open patch numbers
getPatchDetailsForPatchNumber	Returns details about a specific patch number
getResynchClasses	Returns a collection of classes that require resynchronization so that the external system that makes this method call can update its cache
reassignPatchNumbers	Reassigns entities from one patch to another
setPatchNumber	Sets a specific patch number
unsetPatchNumber	Unsets a specific patch number

closePatchNumber

Signature
```
closePatchNumber(patchNumber: Integer;
                 userName:    String): Boolean;
```

The **closePatchNumber** method of the **JadePatchControlInterface** class closes the patch number specified in the **patchNumber** parameter for the user specified in the **userName** parameter.

This method returns **true** if the patch version closes successfully or it returns **false** if it does not.

getAllEntitiesForPatchNumber

Signature
```
getAllEntitiesForPatchNumber(patchNumber: Integer): ObjectSet;
```

The **getAllEntitiesForPatchNumber** method of the **JadePatchControlInterface** class returns a reference to all entities that were updated by using the patch number specified in the **patchNumber** parameter.

Note In addition to instances of subclasses of the **SchemaEntity** class, the changed entities for a patch include RPS mappings, HTML documents, applications, and ActiveX controls. However, not all entity changes are recorded in patch control; for example, adding a Relational View is not recorded.

getCheckedOutEntitiesForPatch

Signature
```
getCheckedOutEntitiesForPatch(patchNumber: Integer;
                              deltaId:     String): MethodSet;
```

The **getCheckedOutEntitiesForPatch** method of the **JadePatchControlInterface** class returns a set of all methods that were updated by using the patch number specified in the **patchNumber** parameter and that are checked out to the delta with a delta identifier specified in the **deltaId** parameter. If a value is not specified for the **deltaId** parameter, the **getCheckedOutEntitiesForPatch** method returns a set of all checked out methods for the specified patch number.

getDeletedEntitiesForPatchNo

Signature
```
getDeletedEntitiesForPatchNo(patchNumber: Integer): HugeStringArray;
```

The **getDeletedEntitiesForPatchNo** method of the **JadePatchControlInterface** class returns an array of all entities that were deleted by using the patch version specified in the **patchNumber** parameter. The format of deleted entities is as follows.

<object-type>::<fully-qualified-entity-name>

getEntitiesForPatchNumber

Signature
```
getEntitiesForPatchNumber(patchNumber: Integer): ObjectSet;
```

The **getEntitiesForPatchNumber** method of the **JadePatchControlInterface** class returns a set containing all instances of subclasses of the **SchemaEntity** class that were updated by using the patch number specified in the **patchNumber** parameter.

getOpenPatchNumbers

Signature
```
getOpenPatchNumbers(): IntegerArray;
```

The **getOpenPatchNumbers** method of the **JadePatchControlInterface** class returns an array of all patch numbers that are currently open.

getPatchDetailsForPatchNumber

Signature
```
getPatchDetailsForPatchNumber(patchNumber:     Integer;
                              _dateTimeOpened: TimeStamp output;
                              _openedBy:       String output;
                              _description:    String output;
                              _dateTimeClosed: TimeStamp output;
                              _closedBy:       String output;
                              _count:          Integer output):
                                                          Boolean;
```

The **getPatchDetailsForPatchNumber** method of the **JadePatchControlInterface** class obtains patch details for the patch number specified in the **patchNumber** parameter and returns **true** if the specified patch number exists in the JADE database.

The parameters to which patch number details are output are listed in the following table.

Parameter	Specifies the…
_dateTimeOpened	Date and time at which the patch number was opened
_openedBy	User id of the user who opened the patch number
_description	Description specified in the Patch Number Update dialog when a patch number is added
_dateTimeClosed	Date and time at which the patch number was closed
_closedBy	User id of the user who closed the patch number
_count	Number of patch entries

This method returns **false** if the patch number specified in the **patchNumber** parameter is not found.

getResynchClasses

Signature `getResynchClasses(classColl: ClassColl input);`

The **getResynchClasses** method of the **JadePatchControlInterface** class returns a collection of system classes that require resynchronization so that the external system that makes this method call can update its cache.

reassignPatchNumbers

Signature
```
reassignPatchNumbers(schemaName:  String;
                     fromVersion: String;
                     fromPatch:   Integer;
                     toVersion:   String;
                     toPatch:     Integer;
                     entities:    HugeStringArray;
                     errorString: String output): Boolean;
```

The **reassignPatchNumbers** method of the **JadePatchControlInterface** class reassigns entities from one patch to another.

The parameters that specify information for the reassignment are listed in the following table.

Parameter	Specifies the…
schemaName	Name of schema to which the patches belong
fromVersion	System version number from which the patches are moved
fromPatch	Patch number from which the patches are moved
toVersion	System version number to which the patches are moved
toPatch	Patch number to which the patches are moved
entities	Array containing the qualified name of the entities to be moved
errorString	Description of the error if the reassignment fails

If the entities are successfully reassigned, the method returns **true** and the value of the **errorString** parameter is an empty string. If the reassignment fails, the method returns **false** and the reason for failure is output to the **errorString** parameter.

The following method shows the use of the **reassignPatchNumbers** method.

```
vars
    jpcf :     JadePatchControlInterface;
    entities : StringArray;
    error :    String;
begin
    create jpcf transient;
    create entities transient;
    entities.add('TenderSaleItem::getTypeString');
    entities.add('OrderProxy::action');
    entities.add('TransactionAgent::zSilentLockExceptionHandler');
    entities.add('InitialDataLoader::zCloseTendersAtCurrentDate');
    entities.add('AddressableEntity::getNameAndAddress');
    jpcf.reassignPatchNumbers('ErewhonInvestmentsModelSchema', '7.1.03',
                                    400, '7.1.03', 300, entities, error);
    write error;
epilog
    delete jpcf;
    delete entities;
end;
```

setPatchNumber

Signature `setPatchNumber(patchUser: String; patchNumber: Integer): Boolean updating;`

The **setPatchNumber** method of the **JadePatchControlInterface** class sets the patch number specified in the **patchNumber** parameter for the user specified in the **patchUser** parameter.

This method returns **true** if the patch version is set successfully or it returns **false** if it is not.

unsetPatchNumber

Signature `unsetPatchNumber(patchUser: String; patchNumber: Integer): Boolean updating;`

The **unsetPatchNumber** method of the **JadePatchControlInterface** class unsets the patch number specified in the **patchNumber** parameter for the user specified in the **patchUser** parameter.

This method returns **true** if the patch version is unset successfully or it returns **false** if it is not.

JadePrintData Class

The **JadePrintData** class is the abstract superclass of report output data classes that enables you to store print data or send it directly to a display device for previewing. Each page of print output is contained in a Windows metafile (in the enhanced metafile **.emf** format) by default, which contains details of all Application Programming Interfaces (APIs) calls necessary to reproduce print output in a form independent of the printer device so that it can be output directly to a display device. You can use this metafile, for example, to send electronic mail output with a message or insert it into Microsoft Word for Windows. (Note that the metafile is inserted into Word for Windows; the file is not opened from within Word.) A metafile stretches or shrinks to the orientation and size of the page with which you are dealing. For example, a page that was painted with portrait orientation and is subsequently changed to landscape orientation may be stretched in width but compacted in height to fit the new orientation.

JADE print data can be one of the following formats.

- Scalable Vector Graphics (SVG), which is the default value on all operating systems
- Windows Enhanced Meta Files (EMF)

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. For more details, see "Portable Printing" under "Printer Class".

By default, when a print preview is requested, a **JadeReport** object is created that contains an array of transient **JadePrintDirect** and **JadePrintPage** object entries. This **JadeReport** object is invisible and transparent to you.

Use the **Printer** class **setReport** method to capture this output for storage, manipulation, and printing to meet your requirements. Alternatively, use the **Printer** class **printPage** method to print the specified page of print output on the current printer.

Note If you are running JADE in thin client mode, the page of print output is constructed and previewed on the presentation client, but the binary image containing the page is sent to the application server for storage.

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 **Printer::setReport** to indicate that user logic subsequently stores or manipulates the report output, each page of output is transferred to the application server.

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.

Inherits From: Object

Inherited By: JadePrintDirect, JadePrintPage

JadePrintDirect Class

The **JadePrintDirect** class is a transient class that holds output directives that are sent directly to the *printer*. (The *printer* is a display device for previewing print output.)

These output directive objects are created in JADE by using the **=direct** option of the **TextBox** class or **Label** class **formatOut** property, which 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 that is to be dialed when printing to a fax device, as shown in the code fragment in the following example.

```
create printForm;
app.printer.printPreview     := true;
app.printer.print(printForm.frameDirect);
printForm.listBox1.addItem("add 1");
printForm.listBox1.addItem("add 2");
printForm.faxLabel.formatOut := '=direct';
printForm.faxText.formatOut  := '=direct';
printForm.faxLabel.caption   := '<ToName:Dr. Who> <ToFaxnum:993999999>
                                 <FromName:Ms. Tardis> <FromFaxnum:4321>';
printForm.faxText.caption    := '';
app.printer.print(printForm.frame1);
app.printer.print(printForm.frameDirect);
return;
```

By default, when a print preview is requested, a **JadeReport** object is created that contains an array of transient **JadePrintDirect** and **JadePrintPage** object entries. This **JadeReport** object is invisible and transparent to you.

Use the **Printer** class **setReport** method to capture this output for storage, manipulation, and printing to meet your requirements. Alternatively, use the **Printer** class **printPage** method to print the specified page of print output on the current printer.

Notes Client-side facilities only are available. Print facilities cannot be invoked from a server method.

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. 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 **Printer**::**setReport** to indicate that user logic subsequently stores or manipulates the report output, each page of output is transferred to the application server.

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.

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

Inherits From: JadePrintData

Inherited By: (None)

JadePrintDirect Properties

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

Property	Description
fontBold	Specifies whether the font style is bold
fontItalic	Specifies whether the font style is italic
fontName	Contains the font used to display text 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
left	Contains the left output position for the printed page of output
text	Contains the text that is to be output to the printer
top	Contains the top output position for the printed page of output

fontBold

Type: Boolean

Availability: Read or write at any time

The **fontBold** property of the **JadePrintDirect** class specifies whether the font style of text output directly to a printer or display device is bold.

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, either in the JADE development environment or at run time by using logic.

Notes The font uses the application font if the **fontName** property for direct printing is set to a null 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 **JadePrintDirect** class specifies whether the font style of text output directly to a printer or display device is italic. 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, either in the JADE development environment or at run time by using logic.

Notes The font defaults to the application font if the **fontName** property for direct printing is set to a null 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 **JadePrintDirect** class contains the font of text output directly to a printer or display device. 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.

Note Changing the **fontName** property to a null string (**""**) causes the direct printing to use the default font. The **fontBold** and **fontSize** properties revert to the font of the application.

fontSize

Type: Real

Availability: Read or write at any time

The **fontSize** property of the **JadePrintDirect** class contains the size of the font to be used for text output directly to a printer or display device. The default value is determined by the system.

To change the default font size, specify the size of the font in points.

Notes The font defaults to the application font if the **fontName** property is set to a null 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 **JadePrintDirect** class specifies whether the font style for text output directly to a printer or display device is strikethrough.

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

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

Notes The font defaults to the application font if the **fontName** property for direct print output is set to a null 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 **JadePrintDirect** class specifies whether the font style for text output directly to a printer or display device is underlined.

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)

Notes The font defaults to the application font if the **fontName** property for direct print output is set to a null string (**""**) at run time.

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

left

Type: Real

Availability: Read or write at any time

The **left** property of the **JadePrintDirect** class contains the left coordinate of output printed directly to a printer or display device, in pixels.

When direct printing, this property has no meaning in most cases.

text

Type: String

Availability: Read or write at any time

The **text** property of the **JadePrintDirect** class contains the text output directly to a printer or display device.

top

Type: Real

Availability: Read or write at any time

The **top** property of the **JadePrintDirect** class contains the top coordinate of output printed directly to a printer or display device, in pixels.

When direct printing, this property has no meaning in most cases.

JadePrintPage Class

The **JadePrintPage** class is the transient class that holds a page of print output. Each page of print output is contained in a Windows metafile, which contains details of all Application Programming Interfaces (APIs) calls necessary to reproduce print output in a form independent of the printer device, to enable you to preview printed output.

The JADE print preview facility creates an array of transient **JadePrintDirect** and **JadePrintPage** objects containing the output pages, which you can then manipulate and print to meet your requirements.

Use the **Printer** class **setReport** method to capture this output for storage, manipulation, and printing to meet your requirements. Alternatively, use the **Printer** class **printPage** method to print the specified page of print output on the current printer.

Notes Client-side facilities only are available. Print facilities cannot be invoked from a server method.

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. 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 **Printer**::**setReport** 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 the properties defined in the **JadePrintPage** class, see "JadePrintPage Properties", in the following subsection.

Inherits From: JadePrintData

Inherited By: (None)

JadePrintPage Properties

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

Property	Description
customPaperHeight	Contains the custom height of the page of output when the **documentType** property is set to **Print_Custom_Paper** (256)
customPaperWidth	Contains the custom width of the page of output when the **documentType** property is set to **Print_Custom_Paper** (256)
documentType	Contains the document type for the page of output
orientation	Contains the orientation of the page of output
pageImage	Contains the constructed metafile output for the page
pageNumber	Contains the number of the page to be printed
paperSource	Contains the paper source of the page of output

customPaperHeight

Type: Integer

Availability: Read or write at any time

The **customPaperHeight** property of the **JadePrintPage** class contains the height (in units of a tenth of a millimeter) of the page of print output; that is, the paper size.

This property is ignored if the value of the **JadePrintPage** class **documentType** property is not set to **Print_Custom_Paper** (256).

customPaperWidth

Type: Integer

Availability: Read or write at any time

The **customPaperWidth** property of the **JadePrintPage** class contains the width (in units of a tenth of a millimeter) of the page of print output; that is, the paper size.

This property is ignored if the value of the **JadePrintPage** class **documentType** property is not set to **Print_Custom_Paper** (256).

documentType

Type: Integer

Availability: Read or write at any time

The **documentType** property of the **JadePrintPage** class contains the form type; that is, the paper size, of the page of print output. The default value is **Print_A4**.

This property cannot be modified after printing has begun.

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

Global Constant	Integer Value	Description
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
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

Global Constant	Integer Value	Description
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
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

orientation

Type: Integer

Availability: Read or write at any time

The **orientation** property of the **JadePrintPage** class contains the orientation of your page of print output.

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 default value is **Print_Portrait** (that is, portrait orientation).

pageImage

Type: Binary

Availability: Read or write at any time

The **pageImage** property of the **JadePrintPage** class contains the constructed Windows metafile image of the print output page.

pageNumber

Type: Integer

Availability: Read or write at any time

The **pageNumber** property of the **JadePrintPage** class contains the number of the print output page. The user can modify this property at any time.

The default value is **1**.

paperSource

Type: Integer

Availability: Read or write at any time

The **paperSource** property of the **JadePrintPage** class contains the paper source of the page of print output. This property cannot be modified after printing has begun.

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.)

Use the **Printer** class **getAllPaperSources** method to access the valid paper sources of a printer.

JadeProfiler Class

The **JadeProfiler** class encapsulates the behavior required to configure what is profiled and reported by the JADE Interpreter. Statistics are captured about the execution of methods, focusing on two types of information.

- Execution times and number of times methods are called in an application
- Code coverage of methods executed during testing

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.

Information about the execution times of methods in an application and the number of times methods are called can help developers improve the performance of an application by:

- Making a method that is executed frequently run faster resulting in a large cumulative saving of time
- Improving the structure of the code by eliminating unnecessary method calls

The following example shows the use of the **JadeProfiler** class to produce a standard profiler report.

```
vars
    prof : JadeProfiler;
begin
    prof := process.profiler;
    prof.fileName := "p:\profiling\dev\myprofile.log";
    prof.methodCount := 25;
    prof.reportCacheStatistics := false;
    prof.start;
    // call user methods here ...
    prof.report;
end;
```

Code coverage is a measure used in software testing to describe the degree to which the methods in a system have been executed. It is a useful measure to assure the quality of a set of tests, as opposed to directly reflecting the quality of the system under test. Code coverage can help testers and developers to:

- Discover methods and blocks of code that are not exercised by a set of tests
- Create tests that increase code coverage
- Quantify the overall code coverage of a system, which is one measure of quality

The following example shows the use of the **JadeProfiler** class to perform code coverage.

```
vars
    prof : JadeProfiler;
begin
    prof := process.profiler;
    prof.codeCoverageFileName:= "erewhonshop_20090312_074611.ccd";
    prof.clearMethodCache;
    prof.startCodeCoverage;
    // call test methods here ...
    prof.reportCodeCoverage;
end;
```

Alternatively, you can enable code coverage programmatically by declaring a new application that initiates a special initialize method, as shown in the following example.

```
initializeCodeCoverage() updating;
begin
    create myCodeCoverage; // myCodeCoverage is a reference to JADEProfiler
                           // on app.
    myCodeCoverage.startCodeCoverage(); // start coverage
    myInitialize(); // call standard Application class initialize method
end;
```

To turn off code coverage when it is on, use the **Application** class **finalize** method, as shown in the following example.

```
myFinalize() updating;
begin
    if myCodeCoverage <> null then
        myCodeCoverage.stopCodeCoverage();
        myCodeCoverage.reportCodeCoverage();
        delete myCodeCoverage;
    endif;
    ...
end;
```

Use the **start** method to start the recording of times spent in JADE and external methods. These statistics are recorded until the **stop** method is executed, and they are output to file only when the **report** method is executed. Use this method to optimize your JADE code, by analyzing the performance of your methods. For example:

1. Call the **start** method in your initialize method and then call the **stop** method after the methods whose performance you want to analyze have been called.
2. Call the **report** method to output the recorded profile statistics to a file.
3. Run the JADE application whose performance you want to monitor.
4. Analyze the JADE Profiler report and then make the appropriate changes to your JADE methods, to optimize performance.
5. Repeat steps 3 and 4 until you have made the performance improvements that you require.

Code coverage methods are used in testing JADE methods, as follows.

1. The **startCodeCoverage** method is called in the *setup* method before the testing starts.
2. The **stopCodeCoverage** method is called in the *teardown* method after the testing ends.
3. The **reportCodeCoverage** method is called to output the code coverage statistics to a file.

4. The **resetCodeCoverage** method is called to clear all code coverage data.

For details about analyzing a code coverage file, see "Code Coverage" in Chapter 17 of the *JADE Developer's Reference*. For details about the properties and methods defined in the **JadeProfiler** class, see "JadeProfiler Properties" and "JadeProfiler Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeProfiler Properties

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

Property	Description
codeCoverageFileName	Contains the name of the file used for code coverage output
fileName	Contains the name of the file for profiler output
methodCount	Contains the number of methods to profile
profileRemoteExecutions	Specifies whether method executions on remote nodes are profiled
reportActualTime	Specifies whether method profile times are reported
reportCacheStatistics	Specifies whether cache statistics are reported
reportInCSVFormat	Specifies whether the report is output to a comma-separated values (CSV) file
reportLoadTime	Specifies whether method load times are reported
reportMethodSize	Specifies whether the size of the method in the cache is reported
reportStatistics	Specifies whether profile statistics are reported
reportTotalTime	Specifies whether the total profiling time is reported

codeCoverageFileName

Type: String

The **codeCoverageFileName** property of the **JadeProfiler** class contains the file name and optionally the path to be used to contain the code coverage output. Code coverage records the degree to which the JADE methods in a system have been tested when the **JadeProfiler** class **reportCodeCoverage** method or **viewCodeCoverage** method is called.

If the value of the **codeCoverageFileName** property does not include a path, the value of the **CodeCoverageDirectory** parameter in the [JadeProfiler] section of the JADE initialization file is used (that is, **logs\CodeCoverage**).

The default name of the code coverage file is ***application-name_YYYYMMDD_hhmmss*.ccd** (for example, **erewhonshop_20090312_074611.ccd**).

For details about recording and reporting code coverage statistics, see the methods documented under "JadeProfiler Methods", later in this chapter.

In JADE thin client mode, code coverage output is always output to the workstation that is running the JADE logic; that is, to the application server.

fileName

Type: String

The **fileName** property of the **JadeProfiler** class contains the profile file name to be used for the output of profile statistics recording times spent in JADE and external methods when the **JadeProfiler** class **report** method is called.

You can use this property to specify an optional disk path and a profile file name if you do not want the output from your method profiling output to the log or CSV file located in the path specified by the **ResultsFile** parameter in the [JadeProfiler] section of the JADE initialization file, if any.

The file name, which is dynamic, defaults to**JadeProfiler_<*application-name*>_*yyyyMMdd_hhmmss* .log** or **.csv**. A new report file is therefore generated for each call to the **JadeProfiler** class **report** method.

If a path is not specified in this **ResultsFile** parameter, the file is located in the log file directory, specified by using the **LogDirectory** parameter in the [JadeLog] section of the JADE initialization file. (For details, see your *JADE Initialization File Reference*.) For details about recording and reporting profile statistics, see the methods documented under "JadeProfiler Methods", later in this chapter.

In JADE thin client mode, method profiler output is always output to the workstation that is running the JADE logic; that is, to the application server.

Note You can specify the default value for this property in the **ResultsFile** parameter in the [JadeProfiler] section of the JADE initialization file.

methodCount

Type: Integer

The **methodCount** property of the **JadeProfiler** class contains the number of methods that are profiled. The default value is **100**.

Change the value of this property if you want more or fewer than the first 100 methods profiled.

Note You can override the default value for this property in the **MethodCount** parameter in the [JadeProfiler] section of the JADE initialization file.

profileRemoteExecutions

Type: Boolean

The **profileRemoteExecutions** property of the **JadeProfiler** class specifies whether method executions on remote nodes are profiled. The default value is **false**.

Set the value of this property to **true** if you want to profile method executions on both the client and the server nodes when running in multiuser mode.

Notes Profiling method executions on both the server and client nodes causes the profiler to incur additional overhead.

You can override the default value for this property in the **ProfileRemoteExecutions** parameter in the [JadeProfiler] section of the JADE initialization file.

reportActualTime

Type: Boolean

The **reportActualTime** property of the **JadeProfiler** class specifies whether method profile times are reported. The default value is **true**.

Note You can override the default value for this property in the **ReportActualTime** parameter in the [JadeProfiler] section of the JADE initialization file.

reportCacheStatistics

Type: Boolean

The **reportCacheStatistics** property of the **JadeProfiler** class specifies whether cache statistics are reported during your method profiling session. The default value is **true**.

Note You can override the default value for this property in the **ReportCacheStatistics** parameter in the [JadeProfiler] section of the JADE initialization file.

reportInCSVFormat

Type: Boolean

The **reportInCSVFormat** property of the **JadeProfiler** class specifies whether method profile times are output to a list of the called methods with the execution times as comma-separated values.

By default, the value is **false**; that is, the method call summary reports are directed to a log file.

Note You can override the default value for this property in the **ReportInCSVFormat** parameter in the [JadeProfiler] section of the JADE initialization file.

reportLoadTime

Type: Boolean

The **reportLoadTime** property of the **JadeProfiler** class specifies whether method load times are reported during your method profiling session. The default value is **true**.

Note You can override the default value for this property in the **ReportLoadTime** parameter in the [JadeProfiler] section of the JADE initialization file.

reportMethodSize

Type: Boolean

The **reportMethodSize** property of the **JadeProfiler** class specifies whether sizes of methods in the cache are reported during your profiling session. The default value is **true**.

Note This report is available only if the **MethodCache** parameter in the [JadeInterpreter] section of the JADE initialization file is set to the default value of **multiple**.

You can use the **clearMethodCache** method to flush all methods in the cache for the process that is currently being profiled.

Note You can override the default value for this property in the **ReportMethodSize** parameter in the [JadeProfiler] section of the JADE initialization file.

reportStatistics

Type: Boolean

The **reportStatistics** property of the **JadeProfiler** class specifies whether statistics are reported during your method profiling session. The default value is **true**.

Note You can override the default value for this property in the **ReportStatistics** parameter in the [JadeProfiler] section of the JADE initialization file.

reportTotalTime

Type: Boolean

The **reportTotalTime** property of the **JadeProfiler** class specifies whether the total method profiling time is reported. The default value is **true**.

Note You can override the default value for this property in the **ReportTotalTime** parameter in the [JadeProfiler] section of the JADE initialization file.

JadeProfiler Methods

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

Method	Description
clearMethodCache	Flushes all methods in the cache for the process that is currently being profiled
isProfiling	Specifies whether method profiling is active (that is, it returns **true** if profiling has started)
report	Creates a file containing statistics of times spent in JADE and external methods and the number of method executions
reportCodeCoverage	Creates a file containing code coverage data of JADE methods executed during testing
reset	Clears the current statistics of times spent in JADE and external methods and the number of method executions
resetCodeCoverage	Clears the current statistics of code coverage data of JADE methods executed during testing
start	Starts the capture of times spent in JADE and external methods and the number of method executions
startCodeCoverage	Starts the capture of code coverage data of JADE methods executed during testing
stop	Stops the capture of times spent in JADE and external methods and the number of method executions
stopCodeCoverage	Starts the capture of code coverage data of JADE methods executed during testing
viewCodeCoverage	Displays the Code Coverage Results Browser

clearMethodCache

Signature `clearMethodCache();`

The **clearMethodCache** method of the **JadeProfiler** class flushes all methods in the cache for the process that is currently being profiled. For details about reporting method cache sizes, see the **reportMethodSize** property.

isProfiling

Signature `isProfiling(): Boolean;`

The **isProfiling** method of the **JadeProfiler** class specifies whether method profiling is active (that is, it returns **true** if method profiling has started or it returns **false** if it has not).

report

Signature `report();`

The **report** method of the **JadeProfiler** class outputs profile statistics of times spent in JADE and external methods to the file specified in the **fileName** property. In JADE thin client mode, profiler output is always output to the workstation that is running the JADE logic; that is, to the application server.

The **JadeProfiler** class **methodCount**, **reportActualTime**, **reportCacheStatistics**, **reportLoadTime**, **reportMethodSize**, **reportStatistics**, and **reportTotalTime** properties enable you to tailor the number of methods that are profiled and the details that are reported.

If you do not specify the **fileName** property or it has a **null** value, the profile statistics are dynamically output to the **JadeProfiler_<*application-name*>_*yyyyMMdd_hhmmss* .log** or **.csv** file in the path specified by the **ResultsFile** parameter in the [JadeProfiler] section of the JADE initialization file, if any. A new report file is therefore generated for each call to the **JadeProfiler** class **report** method. If a path is not specified in this property, the file is located in the log file directory, specified by using the **LogDirectory** parameter in the [JadeLog] section of the JADE initialization file. (For details, see the *JADE Initialization File Reference*.)

Depending on the values of the **reportActualTime**, **reportCacheStatistics**, **reportLoadTime**, **reportMethodSize**, **reportStatistics**, and **reportTotalTime** properties, the report produced by this method can contain six sections, as follows.

- The "methods in actual time order" table lists all methods that are called, in the order of the *actual* time spent in each method. This information contains:
  - The number of times that the method was called
  - The minimum, maximum, and average duration of each call (in milliseconds)
  - The percentage of the profiling time

  When running the application in multiuser mode, the schema, class, and method names are followed by the method execution location if the method executes on the server node; for example, **ErewhonInvestmentsViewSchema**::**Sale::loadData(server)**. No method location is output for methods executing locally.

  Details are output for both nodes if a method executes both locally and on the server.
- The "methods in total time order" table lists the methods that were called, in the order of the *total* elapsed time that was spent in each method. The time spent in methods called from each method is included in this table.

Notes External method calls are not reported. The time spent in any external method called from a JADE method is included in the time reported for the calling (JADE) method.

When running the application in multiuser mode, the schema, class, and method names are followed by the method execution location if the method executes on the server node; for example, **ErewhonInvestmentsViewSchema**::**Sale::loadData(server)**. No method location is output for methods executing locally. Details are output for both nodes if a method executes both locally and on the server.

- The "methods in total load time order" table lists the method *load* times, in the order of time each method took to load. This table lists the total time taken to load the method, the number of times that it was loaded, the average load time (that is, total load time divided by the number of times that it was loaded), and the schema, class, and name of the method.

  Use this table to monitor the frequency of method loads so that you can increase the interpreter method cache size if methods are frequently being loaded and exceeding the method cache, by using the **MethodCacheLimit** parameter in the [JadeInterpreter] section of the JADE initialization file.

  Note Methods already loaded in the cache before profiling started are not reported in this table. Use the **clearMethodCache** method to flush the cache.

- The "methods only by size order" table lists the size (in bytes) of the method in cache and the schema, class, and name of the method. On JADE client nodes, the interpreter method cache holds the code for methods executed on that node. Use the **MethodCache** parameter in the [JadeInterpreter] section of the JADE initialization file to allocate the number of method caches. (Multiple caches result in faster load and execution of methods, especially on Symmetric Multiprocessing (SMP) nodes. However, this improved performance is achieved at the expense of an increased usage of physical memory.)

- The "cache statistics" section of profile information contains the method cache limit and the *total* maximum size to which the method cache grew during the profiling session. The cache statistics include the number of methods that were discarded from the cache to make room for new methods during the profiler run. If there were discarded methods, the total size of the methods discarded is also listed.

  If the method cache overflowed (that is, the cache size exceeds the maximum size specified and all methods in the cache were in use and could be discarded), a table lists the amount by which the cache limit was exceeded, in ascending order of ten percentage points (for example, 10%, 20%, 30%, and so on up to 100+%, in units of 10 percentage points), the size of the method cache at that level (that is, the cache limit plus the exceeded amount), and the number of times the cache was exceeded.

  Note When a method is executed, the JADE interpreter must load the method code into cache for execution. If a method is called frequently, tuning the **MethodCacheLimit** parameter in the [JadeInterpreter] section of the JADE initialization file may result in substantial time savings.

- The "system statistics" table lists the global system-wide statistics for the duration of the profile session.

  These values are those returned by the **System** class **getStatistics** or **getStatistics64** method and output in the Statistics window of the JADE Monitor. For details, see "System Class", later in this chapter.

  Note As statistics values are accumulated by the server, they include all system activity that occurred while the profiler was active.

  If the application is running in multiuser mode and other users are accessing that application or any other application (regardless of the profiling setting), the system statistics therefore include all user operations for all applications that were running for the duration of the profile activity.

The system statistics are listed in the following table.

Statistic	Example
Committed transactions	64177
Aborted transactions	20
Get objects	201594
Queued locks	3
Objects created	532
Objects deleted	449
Objects updated	86609
Objects locked	226709
Objects unlocked	40647
Begin notifications	271
End notifications	183
Delivered notifications	597
Server method executions	3

When you call the **report** method and there is an existing profile file, records are appended to existing records, indicated by start and finish times.

Commands provided by the Jade User Interrupt Profiler submenu enable you to profile methods in an application that is currently running. For details, see Chapter 1, "Running a JADE User Application", in the *JADE Runtime Application Guide*. In addition, you can profile unit tests by using the **Report** command in the File menu of the Unit Test Runner form. For details, see "Starting the Collection of Unit Test Results" "Starting the Collection of Unit Test Results", in Chapter 17 of the *JADE Developer's Reference*.

reportCodeCoverage

Signature `reportCodeCoverage();`

The **reportCodeCoverage** method of the **JadeProfiler** class outputs code coverage statistics for JADE methods executed during testing to the file specified in the **codeCoverageFileName** property.

In JADE thin client mode, code coverage output is always output to the workstation that is running the JADE logic; that is, to the application server.

Using the **startCodeCoverage** method starts the recording of the method coverage statistics, which are recorded until the **stopCodeCoverage** method is executed. The statistics are output to file only when the **reportCodeCoverage** method is executed.

When you call the **reportCodeCoverage** method and there is an existing file, records are appended to that file.

Commands provided by the Jade User Interrupt Profiler submenu enable you to capture code coverage statistics for an application that is currently running. For details, see Chapter 1, "Running a JADE User Application", in the *JADE Runtime Application Guide*.

reset

Signature `reset();`

The **reset** method of the **JadeProfiler** class clears profile statistics of the actual and total times spent in JADE and external methods. (Profile statistics are also cleared when the **report** method outputs the statistics to file.)

For details about reporting profile statistics, see the **report** method. See also the **start** and **stop** methods.

resetCodeCoverage

Signature `resetCodeCoverage();`

The **resetCodeCoverage** method of the **JadeProfiler** class clears code coverage statistics of JADE methods executed during testing. (Code coverage statistics are also cleared when the **reportCodeCoverage** method outputs the statistics to file or you output them to the Code Coverage Results Browser by calling the **viewCodeCoverage** method, or the application terminates.)

For details about reporting or viewing code coverage statistics, see the **reportCodeCoverage** method or **viewCodeCoverage** method, respectively. See also the **startCodeCoverage** and **stopCodeCoverage** methods.

start

Signature `start();`

The **start** method of the **JadeProfiler** class starts recording profile statistics of the actual and total times spent in JADE and external methods.

To stop the recording of profile statistics, call the **stop** method.

These profile statistics are accumulated until you output them to a file by calling the **report** method or you clear them by calling the **reset** method.

startCodeCoverage

Signature `startCodeCoverage();`

The **startCodeCoverage** method of the **JadeProfiler** class starts recording code coverage statistics of JADE methods executed during testing.

To stop the recording of code coverage statistics, call the **stopCodeCoverage** method.

These code coverage statistics are accumulated until you output them to a file by calling the **reportCodeCoverage** method or you output them to the Code Coverage Results Browser by calling the **viewCodeCoverage** method, or you clear them by calling the **resetCodeCoverage** method, or the application terminates.

stop

Signature `stop();`

The **stop** method of the **JadeProfiler** class stops recording profile statistics started by the **start** method of the actual and total times spent in JADE and external methods.

These profile statistics are accumulated until you output them to a file by calling the **report** method, you clear them by calling the **reset** method, or you terminate your application.

stopCodeCoverage

Signature `stopCodeCoverage();`

The **stopCodeCoverage** method of the **JadeProfiler** class stops recording code coverage statistics of JADE methods executed during testing started by the **startCodeCoverage** method.

Code coverage data is still maintained for methods in the cache. Statistics are not gathered for new methods loaded into the cache. The **startCodeCoverage** method restarts the recording of statistics.

These code coverage statistics are accumulated until you output them to a file by calling the **reportCodeCoverage** method, you output them to the Code Coverage Results Browser by calling the **viewCodeCoverage** method, you clear them by calling the **resetCodeCoverage** method, or the application terminates.

viewCodeCoverage

Signature `viewCodeCoverage();`

The **viewCodeCoverage** method of the **JadeProfiler** class stops recording code coverage statistics of JADE methods executed during testing that were started by the **startCodeCoverage** method and outputs the results to the Code Coverage Results Browser.

Note When you call the **viewCodeCoverage** method, the Code Coverage Results Browser looks for the file on the application server. When you manually request the load of a file, the Code Coverage Results Browser looks for these files on the client.

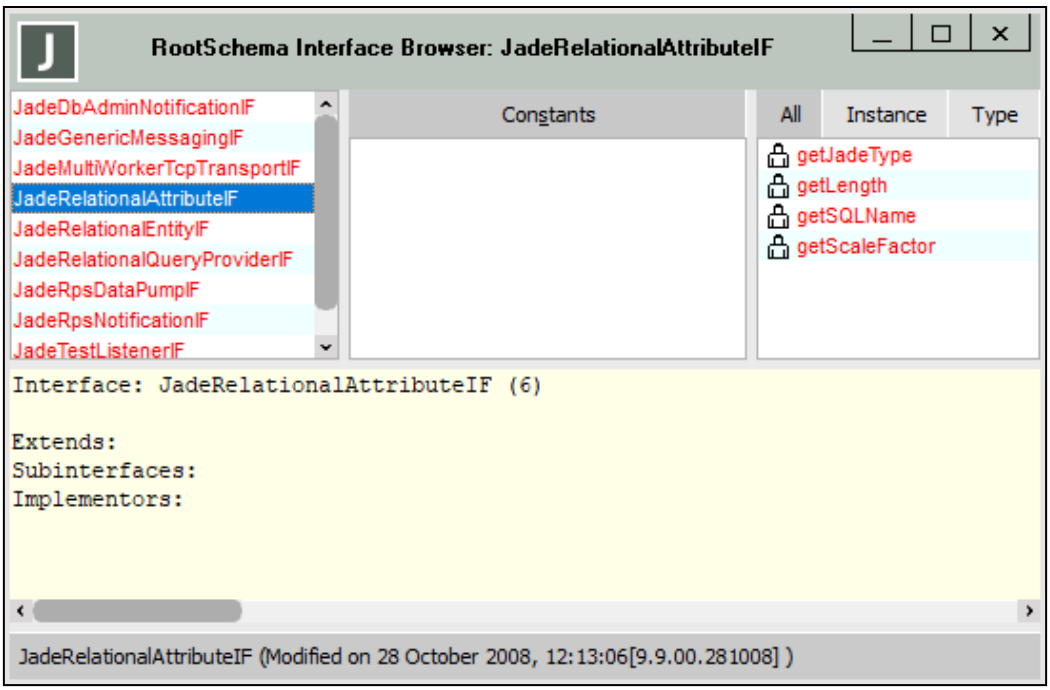
The code coverage statistics are accumulated until you output them to a file by calling the **viewCodeCoverage** or **reportCodeCoverage** method, you clear them by calling the **resetCodeCoverage** method, or you terminate your application. For details about using this browser to analyze code coverage results, see "Code Coverage", in Chapter 17 of the *JADE Developer's Reference*.

JadeRelationalAttributeIF Interface

The **JadeRelationalAttributeIF** interface, defined in the **RootSchema**, provides the definition of the methods that you can implement in your user schema classes to expose soft attributes.

The **JadeRelationalAttributeIF** interface instance is passed to the **addUserAttribute** method of the **RelationalView** class to add a soft attribute to a user-defined entity. The values returned at the time of the method call are stored in the schema meta data for the relational view. If these values change, you must remove the attribute and then add it again to update the meta data.

You can view the **JadeRelationalAttributeIF** 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 **JadeRelationalAttributeIF** 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*. The automatically generated stub methods in classes that implement the interface contain no body logic.

Note It is your responsibility to provide the source code that meets your application requirements for each stub method.

For details about the **JadeRelationalAttributeIF** interface methods, see "JadeRelationalAttributeIF Interface Method Signatures", in the following subsection.

JadeRelationalAttributeIF Interface Method Signatures

The methods provided by the **JadeRelationalAttributeIF** interface are used to analyze information after a call to the **getPropertyValue** method of the **JadeRelationalEntityIF** interface.

Method	Returns the …
getJadeType	Type of the attribute
getLength	Length of the attribute
getSQLName	Column name for the attribute
getScaleFactor	Number of decimal digits (**Decimal** attribute only)

getJadeType

Signature `getJadeType(): Type;`

The **getJadeType** method of the **JadeRelationalAttributeIF** interface returns a reference to the type returned for this attribute from the call to the **getPropertyValue** method of the **JadeRelationalEntityIF** interface.

The type must be a valid primitive type or a class reference. The **Binary**, **Boolean**, **Character**, **Date**, **Decimal**, **Integer**, **Real**, **String**, **Time**, and **TimeStamp** primitive types are supported.

getLength

Signature `getLength(): Integer;`

The **getLength** method of the **JadeRelationalAttributeIF** interface returns the length of the type returned for this attribute from the call to the **getPropertyValue** method of the **JadeRelationalEntityIF** interface. This length is used only for variable length types, specifically the **String**, **Binary**, and **Decimal** primitive types.

You can use the **getLength** method of the **Property** class to define the length if the column is mapped to a JADE property. When called for unbounded **String** or **Binary** primitive types, the **getLength** method returns the unbounded length of the attribute. For **Decimal** primitive types, it returns the precision.

getSQLName

Signature `getSQLName(): String;`

The **getSQLName** method of the **JadeRelationalAttributeIF** interface returns the column name to be used for this attribute in the relational table.

The maximum length of the name, which must be unique to the defined relational view table, is 80 characters.

getScaleFactor

Signature `getScaleFactor(): Integer;`

The **getScaleFactor** method of the **JadeRelationalAttributeIF** interface returns the scale factor (that is, the number of decimal digits) of the primitive type returned for this attribute from the call to the **getPropertyValue** method of the **JadeRelationalEntityIF** interface. The scale factor is used only for **Decimal** primitive types.

You can use the **getScaleFactor** method of the **PrimAttribute** class to define the scale factor if the column is mapped to a property that has a JADE **Decimal** type.

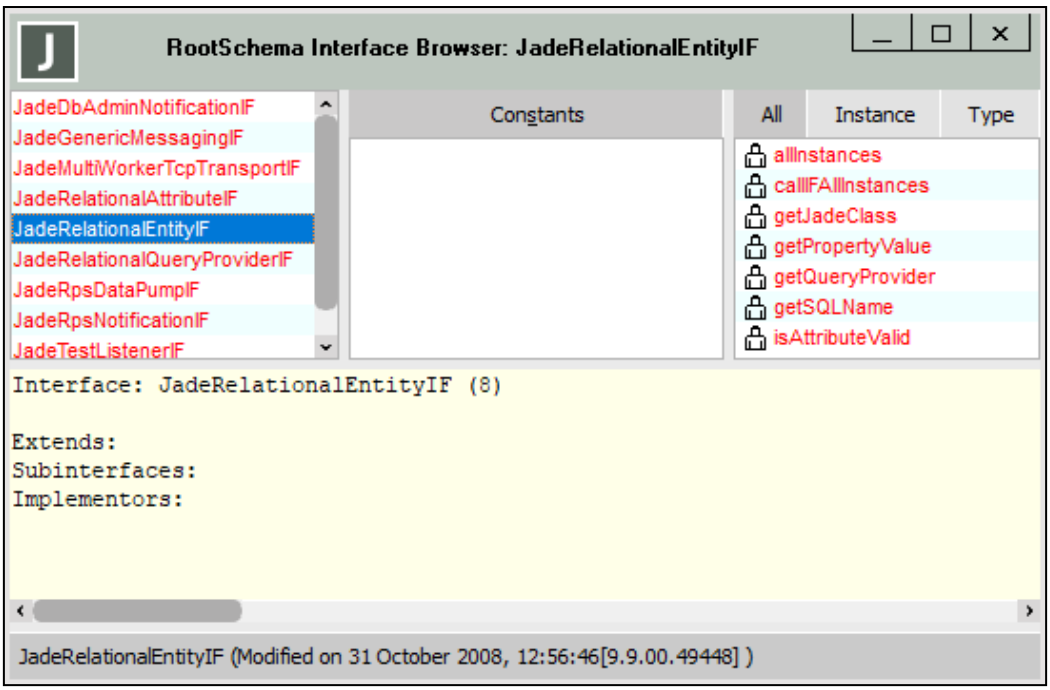
JadeRelationalEntityIF Interface

The **JadeRelationalEntityIF** interface, defined in the **RootSchema**, provides the definition of the methods that you can implement in your user schema classes to expose soft entities by mapping soft entities to tables in the relational view.

The **JadeRelationalEntityIF** interface instance is passed to the **addUserTable** method of the **RelationalView** class to add a soft entity to a relational view. The values returned for the **getJadeClass**, **getSQLName**, and **callIFAllInstances** methods at the time of the method call are stored in the schema meta data for the relational view. If these values change, you must remove the attribute and then add it again to update the meta data.

The **allInstances**, **getPropertyValue**, **getQueryProvider**, and **isAttributeValid** interface methods are called at run time to access the data in the JADE database.

You can view the **JadeRelationalEntityIF** 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 **JadeRelationalEntityIF** 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*. The automatically generated stub methods in classes that implement the interface contain no body logic.

Note It is your responsibility to provide the source code that meets your application requirements for each stub method.

For details about the **JadeRelationalEntityIF** interface methods, see "JadeRelationalEntityIF Interface Method Signatures", in the following subsection.

JadeRelationalEntityIF Interface Method Signatures

The methods provided the **JadeRelationalEntityIF** interface are summarized in the following table.

Method	Returns the …
allInstances	Collection of objects to be used in a query
callIFAllInstances	Determines how the **allInstances** method is to be run
getJadeClass	JADE class for the table
getPropertyValue	Value mapped to the column
getQueryProvider	**JadeRelationalQueryProviderIF** instance used to optimize object selection
getSQLName	Table name in the relational view
isAttributeValid	Whether the attribute is valid for the table

allInstances

Signature `allInstances(): Collection;`

The **allInstances** method of the **JadeRelationalEntityIF** interface returns a reference to a collection of objects that define all instances of the table to be used in the query.

This method is called if the table is:

- Not mapped to a JADE class using the **getJadeClass** method and there is no query provider defined for this table using the **getQueryProvider** method or there is no **WHERE** clause.
- Mapped to a JADE class using the **getJadeClass** method and the **callIFAllInstances** method returned **true**.

If the collection that is returned is a transient collection, the collection is deleted when it is no longer required.

callIFAllInstances

Signature `callIFAllInstances(): Boolean`

The **callIFAllInstances** method of the **JadeRelationalEntityIF** interface specifies how the **allInstances** method is defined when the query is run.

The **callIFAllInstances** method is called when you call the **addUserTable** method of the **RelationalView** class.

This method returns:

- **true** if the **JadeRelationalEntityIF** interface **allInstances** method is to be called for this table.
- **false** if the **allInstances** method of the **Class** class for the class specified by the **getJadeClass** interface call is to be called for this table.

  This class must be visible in the schema in which the relational view is defined. When the **allInstances** method of the **Class** class is used, the ODBC driver uses any available dictionaries and inverses to optimize WHERE clauses.

getJadeClass

Signature `getJadeClass(): Class;`

The **getJadeClass** method of the **JadeRelationalEntityIF** interface returns a reference to the JADE class for which this table is defined, if any.

This method is called when you call the **addUserTable** method of the **RelationalView** class. This class must be visible in the schema in which the relational view is defined.

If a class is associated with the table, the **callIFAllInstances** method returns **false**, and real properties are included, the ODBC driver uses included references to try to optimize any joins in the **WHERE** clause. The ODBC driver uses all instances of the class and subclasses, rather than calling the **allInstances** method of the interface.

If the **getJadeClass** method returns null, no class is associated with your relational table.

getPropertyValue

Signature
```
getPropertyValue(entity:        Object;
                 attributeDesc: JadeRelationalAttributeIF): Any;
```

The **getPropertyValue** method of the **JadeRelationalEntityIF** interface returns the value to be mapped to the column specified in the **attributeDesc** parameter for the entity.

The entity can be any object that has been returned from the **allInstances** method or from the **getResultSet** method of the **JadeRelationalQueryProviderIF** interface. The value of the **attributeDesc** parameter can be any user attribute defined for this table.

getQueryProvider

Signature `getQueryProvider(): JadeRelationalQueryProviderIF;`

The **getQueryProvider** method of the **JadeRelationalEntityIF** interface returns a reference to the implementation of the **JadeRelationalQueryProviderIF** interface that should be used to optimize object selection.

getSQLName

Signature `getSQLName(): String;`

The **getSQLName** method of the **JadeRelationalEntityIF** interface returns the name to be used for the table in the relational view.

This method is called when you call the **addUserTable** method of the **RelationalView** class. The table name must be unique in the relational view. The maximum length of the table name is 80 characters.

isAttributeValid

Signature `isAttributeValid(attributeDesc: Object): Boolean;`

The **isAttributeValid** method of the **JadeRelationalEntityIF** interface returns **true** if the attribute specified in the **attributeDesc** parameter is attribute is valid for this table.

If this message returns **false**, a message is output to the JADE messages log file (**jommsg.log**) and the value of the column is set to **null**.

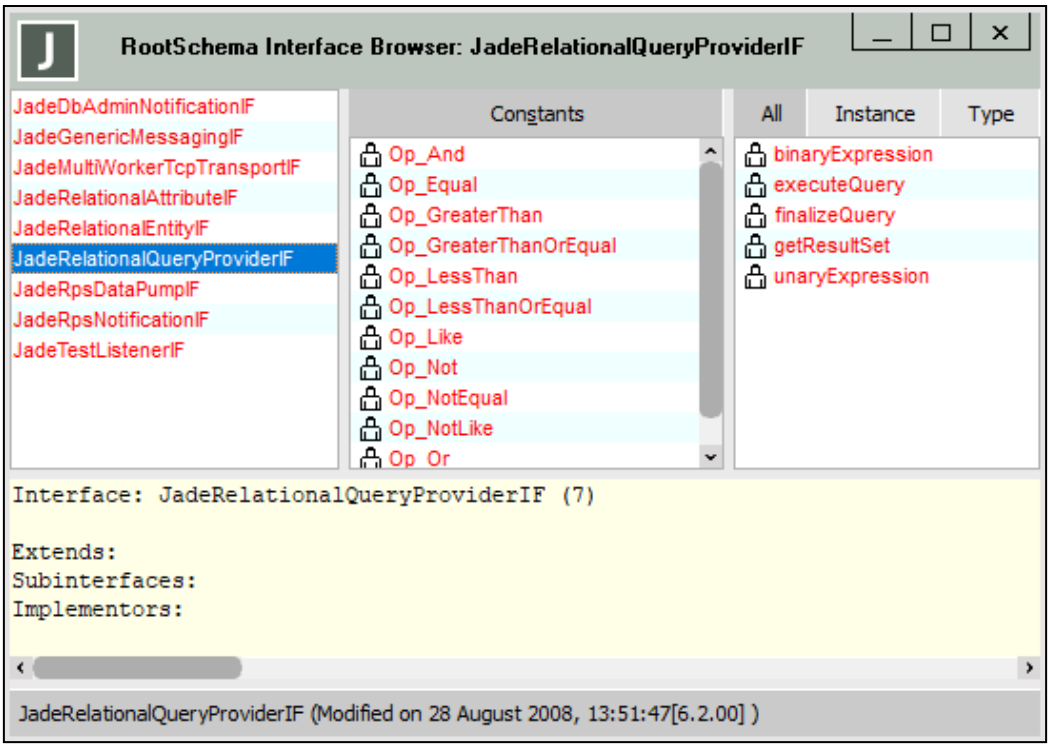
JadeRelationalQueryProviderIF Interface

The **JadeRelationalQueryProviderIF** interface, defined in the **RootSchema**, provides the definition of the methods that you can implement in your user schema classes to provide a search implementation that optimally finds and filters instances of a soft entity.

The interface implementation is used if the **JadeRelationalEntityIF** interface returns a reference to it in the **getQueryProvider** method.

The query provider interface provides methods to map the SQL search expression to an application representation of the expression. The resultant expression is passed back to the application to execute the query.

You can view the **JadeRelationalQueryProviderIF** 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 **JadeRelationalQueryProviderIF** 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*. The automatically generated stub methods in classes that implement the interface contain no body logic.

Note It is your responsibility to provide the source code that meets your application requirements for each stub method.

For details about the **JadeRelationalQueryProviderIF** interface methods, see "JadeRelationalQueryProviderIF Interface Constants" and "JadeRelationalQueryProviderIF Interface Method Callback Signatures", in the following subsections.

JadeRelationalQueryProviderIF Interface Constants

The constants provided by the **JadeRelationalQueryProviderIF** interface are listed in the following table. You can use these class constants to construct composite expressions using the **binaryExpression** and **unaryExpression** methods.

Constant	Integer Value
Op_And	9
Op_Equal	1
Op_GreaterThan	3
Op_GreaterThanOrEqual	4
Op_LessThan	5
Op_LessThanOrEqual	6
Op_Like	7
Op_Not	11
Op_NotEqual	2
Op_NotLike	8
Op_Or	10

JadeRelationalQueryProviderIF Interface Callback Method Signatures

The methods provided the **JadeRelationalQueryProviderIF** interface are summarized in the following table.

Method	Called by ODBC driver to …
binaryExpression	Map a search predicate (an expression tree) onto its own internal representation
executeQuery	Execute a query
finalizeQuery	Remove an expression object that is no longer required
getResultSet	Obtain the result set of the last executed query
unaryExpression	Map a search predicate (an expression tree) onto its own internal representation

binaryExpression

Signature

```
binaryExpression(operator:     Integer;
                 leftOperand:  Object io;
                 rightOperand: Any io;
                 level:        Integer): Object;
```

The **binaryExpression** method of the **JadeRelationalQueryProviderIF** interface is called by the ODBC driver to allow the application to map a search predicate in the form of an expression tree onto its own internal representation. For a simple logical comparison (a binary expression), this interface method is called once.

More-complex expressions that combine predicates with logical operators require multiple calls.

When this method is called to map a logical comparison predicate, the value of the **leftOperand** parameter is the object that represents an attribute (a user object that implements the **JadeRelationalAttributeIF** interface, a property, or a method) and the value of the **rightOperand** parameter is a literal value or an object that represents an attribute. For example, if the **WHERE** clause is **(name="John")**, if name is a soft attribute, the left operand is the implementor of the **JadeRelationalAttributeIF** interface.

If the **name** value is a property, the left operand is the property instance for **name**. If the **name** value is a method, the left operand is the method instance for **name**. The value of the right operand is a string with the value **"John"**.

The depth of the expression tree is indicated by the value of the **level** parameter.

When the **binaryExpression** method is called to map a logical operator (an **AND** or **OR**), the left and right operands are the object returned from a previous call to the **binaryExpression** method or the **unaryExpression** method, or an object that represents an attribute that has a **Boolean** value. For example, if the **WHERE** clause is **(name = "John") AND (surname = "Jones")**, the following calls are made.

```
binaryExpression(Op_Equal, <name attribute>, "John", 2);       // returns A

binaryExpression(Op_Equal, <surname attribute>, "Jones", 2);   // returns B

binaryExpression(Op_And, A, B, 1);                             // returns C
```

The expression represented by the object **C** is used to execute the query.

Your implementation of this method must delete any transient or persistent objects used in defining the expression. If the operand is an intermediate expression object, the implementation can delete the object before returning to the ODBC driver. If the implementation cannot represent the expression, it can return the **null** value, which instructs the ODBC driver to default to its own query execution plan.

executeQuery

Signature `executeQuery(expression: Object io): Boolean;`

The **executeQuery** method of the **JadeRelationalQueryProviderIF** interface is called by the ODBC driver to execute a query. The **expression** parameter that is passed is the value of the expression created by the final call to the application implementation of the **binaryExpression** method or the **unaryExpression** method.

Your implementation of this method must delete any transient or persistent objects used in defining the expression. The implementation can delete the object before returning to the ODBC driver.

finalizeQuery

Signature `finalizeQuery(expression: Object io);`

The **finalizeQuery** method of the **JadeRelationalQueryProviderIF** interface is called by the ODBC driver when the **expression** object is no longer required. If ODBC prepares the statement but does not execute it, this method can be used to free up any transient or persistent objects used in defining the expression.

The value of the **expression** parameter can be null if the **executeQuery** method was called and your application code deleted the expression object before returning.

getResultSet

Signature `getResultSet(): Collection;`

The **getResultSet** method of the **JadeRelationalQueryProviderIF** interface is called by the ODBC driver to obtain the result set of the last executed query.

If the returned collection is a transient collection, the collection is deleted by the ODBC driver when it is no longer required.

unaryExpression

Signature
```
unaryExpression(operator: Integer;
                operand:  Object io;
                level:    Integer): Object;
```

The **unaryExpression** method of the **JadeRelationalQueryProviderIF** interface is called by the ODBC driver to allow the application to map a search predicate in the form of an expression tree onto its own internal representation. The only unary operator is **Op_Not** applied to a **Boolean** value.

If the implementation cannot represent the expression, it can return the **null** value, which instructs the ODBC driver to default to its own query execution plan.

The depth of the expression tree is indicated by the value of the **level** parameter.

JadeReport Class

The **JadeReport** class holds the description of the report output data and enables you to access an entire printed report. This print data can be stored or sent directly to a printer or display device.

Each page of print output is contained in a Windows metafile by default, which contains details of all Application Programming Interfaces (APIs) calls necessary to reproduce print output in a form independent of the printer device, to enable you to preview printed output.

JADE print data can be one of the following formats.

- Scalable Vector Graphics (SVG), which is the default value on all operating systems
- Windows Enhanced Meta Files (EMF)

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. For more details, see "Portable Printing" under "Printer Class".

The following is an example of the code required to capture the report output data and then store it persistently.

```
vars
    report : JadeReport;
begin
    create report transient;
    app.printer.setReport(report);
    ...                  // Do some processing here
    // create print output
    ...                  // Do some processing here
epilog
    app.printer.close;
    // Clone the report output in the report variable to a persistent
    // version of the JadeReport, JadePrintPage, and JadePrintDirect
    // subclasses
    delete report;
end;
```

By default, when a print preview is requested, a **JadeReport** object is created that contains the list of **JadePrintDirect** and **JadePrintPage** entries. This **JadeReport** object is invisible and transparent to you.

Use the **Printer** class **setReport** method to capture this output for storage, manipulation, and printing to meet your requirements. Alternatively, use the **Printer** class **printReport** method to print the specified report on the current printer if the **printPreview** property is set to **false** or print the report in preview mode if the **printPreview** property is set to **true**.

Notes Client-side facilities only are available. Print facilities cannot be invoked from a server method.

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. 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 **Printer** class **setReport** 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 the properties and method defined in the **JadeReport** class, see "JadeReport Properties" and "JadeReport Method", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeReport Properties

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

Property	Description
collate	Contains the collation setting for the report output
copies	Contains the number of copies of report output to produce
description	Contains a textual description of the report
duplex	Contains the duplex setting for the report output
printArray	Contains a reference to an array of print output data
timeStamp	Contains the timestamp when the report output was created

collate

Type: Boolean

The **collate** property of the **JadeReport** class contains the collation setting for the print output; that is, whether the copies are printed in proper binding order by separating copies into groups.

By default, report output is not collated; that is, the value of this property is **false**.

copies

Type: Integer

The **copies** property of the **JadeReport** class contains the number of copies to be printed.

The default value is **1**.

description

Type: String[60]

The **description** property of the **JadeReport** class contains a textual description of the report. The description can be in the range **0** through **60** characters.

This description is for documentation only and is not automatically displayed.

duplex

Type: Integer

The **duplex** property of the **JadeReport** class contains the duplex setting for the report output; that is, the number of sides on which the paper is printed.

The default value is **Duplex_Simplex**; that is, printing is single-sided.

printArray

Type: JadePrintDataArray

The **printArray** property of the **JadeReport** class contains a reference to an array of **JadePrintData** objects (that is, transient **JadePrintDirect** and **JadePrintPage** object entries).

timeStamp

Type: TimeStamp

The **timeStamp** property of the **JadeReport** class contains a reference to the timestamp when the report object was created.

JadeReport Method

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

Method	Description
pageCount	Returns the number of pages in the report

pageCount

Signature `pageCount(): Integer;`

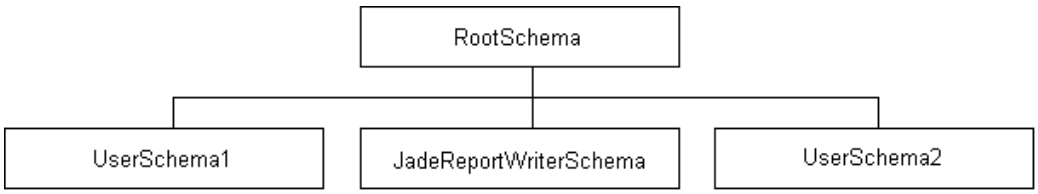
The **pageCount** method of the **JadeReport** class returns the number of pages in the report.

JadeReportWriterManager Class

The JADE Report Writer is a generic report writer that enables you to configure and design reports for all schemas in your JADE database. These reports can be tailored by a JADE developer or by a user of a deployed JADE runtime application. In addition, you can use the methods defined in the **JadeReportWriterManager**, **JadeReportWriterReport**, and **JadeReportWriterSecurity** classes to dynamically override report details at run time.

Like your user schemas, the **JadeReportWriterSchema** schema is implemented as a subschema of the **RootSchema** so that you can use it to write reports for any schema in your JADE database. You can integrate these reports with any JADE application so that they appear as extensions of that application.

The following diagram shows the schema hierarchy.



From the JADE development environment, the **JadeReportWriterSchema** schema provides the following applications that enable you to configure and design your reports.

- **ReportWriterConfiguration**, which sets up a schema view containing the classes and their properties and methods that can be reported on
- **ReportWriterDesigner**, which designs the reports based on that view

The JADE Report Writer acts on transient instances of the report definition details. As the report runs from the transient copy (which requires read-only JADE database access), the database is therefore not locked when a report is edited or printed. For details about locking objects during the query phase of the report, see the **JadeReportWriterReport** class **setQueryOptions** method, later in this chapter.

A report becomes permanent only when it is saved in the JADE Report Writer Designer.

Tip To classify and organize report definitions, create at least one folder for each user schema in your database, as all JADE Report Writer reports defined for all schemas in your database are stored in the **JadeReportWriterSchema** schema for use in applications in the schema to which they are defined. (Use the JADE Report Writer Configuration application to define and maintain report folders.) Arranging report folders in a hierarchy makes it easy for users to find a specific report definition.

The **JadeReportWriterManager** class provides a superclass for each JADE Report Writer:

- Configuration application, which enables you to maintain views, folders, and system options from within the user system.

  Create a transient instance of the **JadeReportWriterManager** class for each configuration application that you require and call the **startReportWriterConfiguration** method, passing the user name of the current user and the name of the subclassed **JadeReportWriterSecurity** class.

- Designer application, which enables you to maintain report definitions and run reports from within the user system.

Create a transient instance of the **JadeReportWriterManager** class for each designer application that you require and call the **startReportWriterDesigner** method, passing the user name of the current user and the name of the subclassed **JadeReportWriterSecurity** class.

If the application user is not allowed access to the JADE Report Writer Configuration or Designer application, no action is taken. If the user is allowed read-only access only, the application is started in read-only mode. If the user has full access, the application is started in full maintenance mode. The application starts in the current process, so that the user context is available when reports are run.

For details about dynamically running an existing JADE Report Writer report, see "Running an Existing Report", in the following subsection. For details about the methods defined in the **JadeReportWriterManager** class, see "JadeReportWriterManager Methods", later in this section. See also the **Application** class **jadeReportWriterTimeDetails** method.

Inherits From: Object

Inherited By: (None)

Running an Existing Report

Methods defined in the **JadeReportWriterManager** and **JadeReportWriterReport** class enable you to dynamically override JADE Report Writer report details at run time. If you want to reimplement access security in the report at run time, subclass the **JadeReportWriterSecurity** class and reimplement the methods defined in that class.

To run an existing report

1. Create a transient instance of the **JadeReportWriterManager** class.
2. Call the **setSecurity** method, passing an instance of the **JadeReportWriterSecurity** subclass.
3. Call the **setUserName** method, passing the user name of the current user.
4. Call the **getReport** method, passing the name of the report. If the report does not exist or it is not accessible to the current user, this method returns **null**. If the report exists and it can be accessed by the user, it returns a transient instance of the **JadeReportWriterReport** class.

To use the transient instance of the **JadeReportWriterReport** class returned by the **getReport** method of the **JadeReportWriterManager** class:

1. Call the **setOutputDestination** method, passing the type of report output that you require (that is, output to the printer or to one of the valid file types).
2. If you want to use the current report output option values or you want to see the existing values before setting new ones, you can retrieve these by calling the appropriate ***get*** methods to obtain the current values.
3. Call the appropriate ***set*** methods (for example, **setPageOptions** and **setPreviewOptions** for a report to output to the printer or **setOutputFileTitle** for a report to a file, and so on).
4. If the report uses parameters (typically for report selection), set these individually by using the **setParameter** method, passing the parameter name and value.
5. If the report has more than one profile defined for it, call the **setProfile** method to set the profile on which you want to report, passing the profile name.
6. When the report setup is complete, call the **JadeReportWriterReport** class **run** method to start the report.

Alternatively, run the report by calling the **JadeReportWriterReport** class **runWithStatus** method if you want to display and refresh a progress dialog and return the success of the report and the page count if the report was output to a printer or the record count if the report was extracted to a file.

7. You can retain the **JadeReportWriterReport** instance to run further copies of the report, if required. To do this, repeat steps 1 through 6, as appropriate.

8. When you no longer require the **JadeReportWriterReport** instance, you should delete it so that you clean up all associated report definition transient objects that are associated with it.

JadeReportWriterManager Methods

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

Method	Description
createReport	Creates a new empty report with the specified parameter values
getAllReportNames	Updates the parameter array values with the folder path and report names visible to the current user
getAllViewNames	Updates the parameter array value with all report views visible to the current user
getFolderPaths	Gets an array of the full paths of each report folder to which the user has access
getFoldersInFolderPath	Gets an array of all child folders to which the current user has access in the specified folder
getReport	Returns a transient instance of the **JadeReportWriterReport** class for the specified report
getReportsInFolderPath	Gets an array of all report names in the folder to which the user has access
getViewDetails	Retrieves the details of the specified report view
isReportWriterInstalled	Specifies whether the **JadeReportWriterSchema** is loaded and available
setSecurity	Sets the **JadeReportWriterSecurity** class to the user subclass of **JadeReportWriterSecurity**
setSecurityObject	Sets the object to be passed to the **Object** class **jadeReportWriterCheck** method during the query phase of the JADE Report Writer process
setUserName	Sets the user name that is to be used in security checks
startReportDesignerForReport	Starts the JADE Report Writer Designer application and automatically opens the specified report
startReportWriterConfiguration	Starts the JADE Report Writer Configuration application, passing the current user code and security class
startReportWriterDesigner	Starts the JADE Report Writer Designer application, passing the current user code and security class

createReport

Signature
```
createReport(reportName:          String;
             folderPath:          String;
             viewName:            String;
             rootCollectionAlias: String;
             copyFromReportName:  String): Integer;
```

The **createReport** method of the **JadeReportWriterManager** class creates a new empty report with the specified parameter values listed in the following table.

Parameter	Description
reportName	Name of the report to be created
folderPath	Existing folder in which the report is to be located and which can be accessed by the current user
viewName	Name of an existing reporting view for the report that can be accessed by the current user
rootCollectionAlias	Existing reporting collection alias for the reporting view
copyFromReportName	Optional name of an existing report that can be accessed by the current user and from which the new report is copied

You must specify a value for all parameters other than the **copyFromReportName** parameter. The parameter values are checked before creating the report.

If the report is created successfully, this method returns a zero (**0**) value. If the report creation fails, the return value indicates the error status, which depends on the parameter value that caused the error. These return error values are listed in the following table.

Failure Return Value	Meaning
1	The report name contains spaces or the length is greater than 100 characters
2	A report already exists with the name specified in the **reportName** parameter
3	The folder path is invalid or it cannot be accessed by the current user
4	The reporting view name does not exist or it cannot be accessed by the current user
5	The reporting collection alias does not exist for the reporting view
6	The base report does not exist or it cannot be accessed by the current user

getAllReportNames

Signature
```
getAllReportNames(folders: HugeStringArray input;
                  reports: StringArray input);
```

The **getAllReportNames** method of the **JadeReportWriterManager** class updates the **folders** and **reports** parameter array values with the folder path and report names visible to the current user.

The array updated by the **folders** parameter contains a list of the folders in which each report is saved; that is, the first element in the folder array is the folder path of the first report name.

The array updated by the **reports** parameter contains a list of report names for all reports visible to the current user.

The folder path is built from the folder names at each level in the hierarchy, separated by a forward slash character (*/*). Template paths start with a forward slash, but report paths do not. For example:

- A folder named **Common** within a folder named **Payroll** within the root report folder named **Reports** has a folder path of *Reports/Payroll/Common*.
- A folder named **Secured** within a folder named **Sales** within the root template folder named **Templates** has a folder path of */Templates/Sales/Secured*.

The following example shows the use of the **getAllReportNames** method.

```
load() updating;
vars
    jrwm    : JadeReportWriterManager;
    reps    : StringArray;
    folders : HugeStringArray;
    strName : String;
begin
    create jrwm transient;
    create reps transient;
    create folders transient;
    jrwm.getAllReportNames(folders, reps);
    foreach strName in reps do
        lstReports.addItem(strName);
    endforeach;
epilog
    delete reps;
    delete folders;
    delete jrwm;
end;
```

getAllViewNames

Signature `getAllViewNames(viewNames: StringArray input);`

The **getAllViewNames** method of the **JadeReportWriterManager** class updates the **viewNames** parameter array value with a list of the names of all reporting views visible to the current user. You can then call the **JadeReportWriterManager** class **getViewDetails** method to retrieve the details of a specified reporting view.

getFolderPaths

Signature `getFolderPaths(paths: HugeStringArray);`

The **getFolderPaths** method of the **JadeReportWriterManager** class updates the **paths** parameter array value with the full paths of each report to which the current user has access.

getFoldersInFolderPath

Signature
```
getFoldersInFolderPath(path:    String;
                       folders: StringArray);
```

The **getFoldersInFolderPath** method of the **JadeReportWriterManager** class updates the **folders** parameter array value with the names of all child folders to which the current user has access in the folder specified in the **path** parameter.

getReport

Signature `getReport(reportName: String): JadeReportWriterReport;`

The **getReport** method of the **JadeReportWriterManager** class returns a transient instance of the **JadeReportWriterReport** class for the report specified in the **reportName** parameter.

If the report does not exist or is not accessible to the current user, it returns **null**. If the report exists and it can be accessed by the current user, it returns a transient instance of the **JadeReportWriterReport** class.

The following example shows the use of the **getReport** method.

```
lstReports_click(listbox: ListBox input) updating;
vars
    strName     : String;
    jrep        : JadeReportWriterReport;
    profiles    : StringArray;
    profileName : String;
begin
    if lstReports.listIndex = -1 then
        return;
    endif;
    strName := lstReports.itemText[lstReports.listIndex];
    jrep    := self.myJWRM.getReport(strName);
end;
```

getReportsInFolderPath

Signature
```
getReportsInFolderPath(path:    String;
                       reports: StringArray);
```

The **getReportsInFolderPath** method of the **JadeReportWriterManager** class updates the **reports** parameter array value with the names of all reports to which the current user has access in the folder specified in the **path** parameter.

getViewDetails

Signature
```
getViewDetails(viewName:         String;
               description:      String output;
               topSchemaName:    String output;
               lowestSchemaName: String output;
               concurrency:      Integer output;
               folders:          HugeStringArray input;
               allowTransients:  Boolean output);
```

The **getViewDetails** method of the **JadeReportWriterManager** class updates the **description**, **topSchemaName**, **lowestSchemaName**, **concurrency**, **folders**, and **allowTransients** parameter values with the details of the reporting view specified in the **viewName** parameter.

Use this method to obtain the details of the reporting views visible to the current user obtained by calling the **JadeReportWriterManager** class **getAllViewNames** method.

isReportWriterInstalled

Signature `isReportWriterInstalled(): Boolean;`

The **isReportWriterInstalled** method of the **JadeReportWriterManager** class specifies whether the **JadeReportWriterSchema** is installed.

This method returns a value of **true** if the **JadeReportWriterSchema** is loaded and available. If it is not, this method returns a value of **false**.

The following example shows the use of the **isReportWriterInstalled** method.

```
mnuFileConfigure_click(menuItem: MenuItem input) updating;
vars
    jrwm : JadeReportWriterManager;
begin
    create jrwm transient;
    if jrwm.isReportWriterInstalled then
       jrwm.startReportWriterConfiguration(app.myUser.name,
                                           MySecurityClass);
    endif;
epilog
    delete jrwm;
end;
```

setSecurity

Signature `setSecurity(securityObject: JadeReportWriterSecurity) updating;`

The **setSecurity** method of the **JadeReportWriterManager** class sets the **JadeReportWriterSecurity** class to the security object (specified in the **securityObject** parameter) to be used in the **getReport** and **getAllReportNames** methods.

setSecurityObject

Signature `setSecurityObject(securityObject: Object) updating;`

The **setSecurityObject** method of the **JadeReportWriterManager** class sets the object (specified in the **securityObject** parameter) that is to be passed to the **Object** class **jadeReportWriterCheck** method during the query phase of the JADE Report Writer process.

The JADE Report Writer keeps a reference to the security object reference set when the **securityObject** method is called. The security object is used whenever the JADE Report Writer needs to check security access to something. This reference is held in the internal equivalent of the user **app** object, so needs to be valid for the life of the application.

Instead of deleting it after calling the JADE Report Writer **startReportWriterConfiguration** or **startReportWriterDesigner** methods, it would be best to create it once and hold a reference in the user **app** object or equivalent, and reuse this instance so it stays valid. It can be deleted at the end of the application.

setUserName

Signature `setUserName(username: String) updating;`

The **setUserName** method of the **JadeReportWriterManager** class sets the user name (specified in the **username** parameter) to be used in the **getReport** and **getAllReportNames** methods.

startReportDesignerForReport

Signature `startReportDesignerForReport(reportName: String; userCode: String; securityClass: Class);`

The **startReportDesignerForReport** method of the **JadeReportWriterManager** class starts the JADE Report Writer Designer application with the user code and security class specified in the respective **userCode** and **securityClass** parameters in the same way as the **startReportWriterDesigner** method and then automatically opens the report specified in the **reportName** parameter.

If the specified report does not exist or it is not visible to the current user, the JADE Report Writer Designer application opens in the normal way, prompting for the report or template name.

startReportWriterConfiguration

Signature `startReportWriterConfiguration(usercode: String; securityClass: Class);`

The **startReportWriterConfiguration** method of the **JadeReportWriterManager** class starts the JADE Report Writer Configuration application using the user name specified in the **usercode** parameter and the user-defined subclass of the **JadeReportWriterSecurity** specified in the **securityClass** parameter (which can be a **null** value if you require only the default security).

If the specified user is not allowed to access the JADE Report Writer Configuration application, the application is not started.

The following example shows the use of the **startReportWriterConfiguration** method.

```
mnuFileConfigure_click(menuItem: MenuItem input) updating;
vars
    jrwm : JadeReportWriterManager;
begin
    create jrwm transient;
    if jrwm.isReportWriterInstalled then
        jrwm.startReportWriterConfiguration(app.myUser.name,
                                            MySecurityClass);
    endif;
epilog
    delete jrwm;
end;
```

startReportWriterDesigner

Signature `startReportWriterDesigner(usercode: String; securityClass: Class);`

The **startReportWriterDesigner** method of the **JadeReportWriterManager** class starts the JADE Report Writer Designer application using the user name specified in the **usercode** parameter and the user-defined subclass of the **JadeReportWriterSecurity** specified in the **securityClass** parameter (which can be a **null** value if you require only the default security).

If the specified user is not allowed to access the JADE Report Writer Designer application, the application is not started.

Note As the JADE Report Writer Designer application is run in the current process rather than as a separate JADE application, any methods that are called from the query phase have the appropriate context and user objects available (for example, the **app** system variable).

If you want the current application to still be available while the JADE Report Writer Designer application is in use, you should therefore run a new JADE application of the user-defined schema, with the information required to establish the same user context (for example, as if the current user had logged on to the new application in the same way he or she had logged on to the current application), and use this **startReportWriterDesigner** method to start the JADE Report Writer Designer application on the new JADE application.

The following example shows the use of the **startReportWriterDesigner** method.

```
btnDesigner_click(btn: Button input) updating;
vars
    jrwm : JadeReportWriterManager;
begin
    create jrwm transient;
    jrwm.startReportWriterDesigner(txtUsercode.text, MySecurityClass);
epilog
    delete jrwm;
end;
```

JadeReportWriterReport Class

The **JadeReportWriterReport** class, in conjunction with the **JadeReportWriterManager** class and **JadeReportWriterSecurity** class, provides methods that enable you to dynamically override JADE Report Writer details and programmatically run report definitions at run time.

A transient instance of the **JadeReportWriterReport** class is created and returned by the **getReport** method of the **JadeReportWriterManager** class. To use this class instance, call the appropriate ***set*** methods of the **JadeReportWriterReport** class, passing the parameters that you require (for example, call the **setOutputDestination** method, passing the type of file that you require for the report output). You can obtain the current values of the report options by using the appropriate ***get*** methods.

When you have finished setting up the report, call the **JadeReportWriterReport** class **run** method to run the report dynamically at run time or the **runWithStatus** method to run the report, display and refresh a progress dialog, and return the success of the report and the page count if the report was output to a printer or the record count if the report was extracted to a file.

For an overview of the JADE Report Writer and the relationship of the **JadeReportWriterManager**, **JadeReportWriterReport**, and **JadeReportWriterSecurity** classes, see "JadeReportWriterManager Class", earlier in this chapter. For details about dynamically running an existing JADE Report Writer report, see "Running an Existing Report", earlier in this chapter.

For details about the constants and methods defined in the **JadeReportWriterReport** class, see "JadeReportWriterReport Class Constants" and "JadeReportWriterReport Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeReportWriterReport Class Constants

The constants provided by the **JadeReportWriterReport** class for use in the **setOutputDestination** and **setQueryOptions** methods are listed in the following table.

Class Constant	Integer Value	Description
CONCURRENCY_NONE	1	Does nothing during the query evaluation
CONCURRENCY_READ_COMMITTED	3	Resynchronizes every object that is accessed, but holds on to locks on collections
CONCURRENCY_READ_OPTIMISTIC	4	Resynchronizes every object that is accessed, locks objects needed later, and checks the edition (the default value)
CONCURRENCY_READ_PESSIMISTIC	5	Brackets a read-only transaction to ensure that objects referenced after the start of the query and before the end of the query are the latest editions of the objects (that is, acquires an implicit shared lock on each object as it is referenced)
CSV	3	Comma-Separated Value (CSV) file format
DELIMITED_FILE	3	Delimited file format (for example, containing commas, semicolons, tabs, or spaces)
HTML	2	HyperText Markup Language (HTML) file

Class Constant	Integer Value	Description
PRINTER	0	Output to the default printer of the user
RESOURCE_LIMIT_NONE	0	Not applicable (that is, ignored), which is the default value
RESOURCE_LIMIT_READS	1	Maximum number of objects to read
RESOURCE_LIMIT_RESULTS	2	Maximum number of result objects to add
RESOURCE_LIMIT_TIME	3	Maximum number of milliseconds the report query runs
RESOURCE_LIMIT_QUERYOPS	4	Maximum number of query operations
RTF	4	Rich Text File (RTF) file format for Microsoft Word for Windows
TEXT	6	Text (or ASCII) file format, and can contain new page characters
XML	5	Extensible Markup Language (XML) file format

The **CSV** and **DELIMITED_FILE** constants, which have the same integer value, are the same format in that in both you can use commas, semicolons, tabs, or spaces as field delimiters. Traditionally, as the name suggests, **.csv** was applied to files in which only commas could be used as delimiters. **CSV** is therefore retained as a class constant for JADE Report Writer applications even though it has the same integer value as the **DELIMITED_FILE** constant.

JadeReportWriterReport Methods

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

Method	Description
getDefaultOutputDestination	Returns an integer value containing **JadeReportWriterReport** class constants representing the default output destination
getDefaultProfile	Returns a string value containing the name of the default profile
getDelimitedFileOptions	Obtains the current values to be used when running the report for extraction to a delimited file
getExtraParameterDetails	Obtains whether a value is mandatory and the user prompt value of the specified parameter
getFolder	Returns a string containing the path to the folder in which the report is saved
getHtmlOptions	Obtains the current values to be used when running the report for extraction to an HTML file
getOutputFileTitle	Returns a string value containing the full title of the default output file
getPageOptions	Obtains the current report attributes relating to paper size values used when running the report
getParameterDetails	Obtains the parameter type, length and scale factor, current value, is used, and ignore status
getParameters	Obtains the names and types of all individual parameters

Method	Description
getParametersForProfile	Populates the string arrays specified in the **names** and **types** parameters with the parameter names and types that are used in the profile specified in the **profileName** parameter.
getProfileDescription	Returns the description of the specified profile
getProfiles	Returns an array of profile names defined in the report
getQueryOptions	Returns the concurrency and resource limits that are to be applied at the query phase of the report run
getReportDescription	Returns the report description (entered in the **Description** text box on the Report Properties dialog)
getReportingViewName	Returns the name of the reporting view for the report
getRootCollections	Returns the alias and path of each report root collection or join, or both root collections and joins
getTextOptions	Obtains the number of lines on pages when running the report for extraction to a text file
getUseClientFileSystem	Returns **true** if the presentation client file system is used or it returns **false** if the application server file system is used
getXmlOptions	Obtains the current values to be used when running the report for extraction to an XML file
run	Runs the current report, using the parameter values specified in the appropriate ***set*** methods
runWithStatus	Runs the current report using the parameter values specified in the appropriate ***set*** methods, displays and refreshes a progress dialog, and returns the status of the report after it has been run, indicating the success of the report run and the page count of a report output to a printer or the record count of a report extracted to file
setDelimitedFileOptions	Sets the values to be used when running the report for extraction to a delimited file
setEndingNotification	Passes an object that is to be notified when the report finishes
setHtmlOptions	Sets the values to be used when running the report for extraction to an HTML file
setLocaleDateOptions	Sets date formats to be used when running the report
setLocaleNumericOptions	Sets number and currency formats to be used when running the report
setLocaleTimeOptions	Sets time formats to be used when running the report
setOutputDestination	Sets the destination of the report output
setOutputFileTitle	Sets the title of the output file that is created when the report runs
setPageOptions	Sets the **Printer** class attributes relating to paper size values to be used when running the report
setParameter	Sets the name and type of a parameter to be used when running the report
setParameterIgnoreInSelection	Specifies whether a parameter is to be ignored when selecting parameters for the current run of the report

Method	Description
setPreviewOptions	Sets the **Printer** class preview attributes to be used when running the report
setProfile	Sets the report profile name to use when running the report
setQueryOptions	Sets specified limits on the execution of a query when running the report
setStartEndMethods	Sets the names of the application initialize method that is called before the report is run and the finalize method that is called after the report has completed
setTextOptions	Sets the number of lines on pages when running the report for extraction to a text file
setUseClientFileSystem	Specifies whether the presentation client file system is used (**true**) or the application server file system is used (**false**)
setXmlOptions	Sets the values used when running the report for extraction to an XML file

getDefaultOutputDestination

Signature `getDefaultOutputDestination(): Integer;`

The **getDefaultOutputDestination** method of the **JadeReportWriterReport** class returns an integer value containing the **JadeReportWriterReport** class constant representing the default output destination. The output destination can be one of the values listed in the following table.

Class Constant	Integer Value	Description
CSV	3	Comma-Separated Value (CSV) file format
DELIMITED_FILE	3	Delimited file format (for example, containing commas, semicolons, tabs, or spaces)
HTML	2	HyperText Markup Language (HTML) file
PRINTER	0	Output to the default printer of the user
RTF	4	Rich Text File (RTF) file format for Microsoft Word for Windows
TEXT	6	Text (or ASCII) file format, and can contain new page characters
XML	5	Extensible Markup Language (XML) file format

The **CSV** and **DELIMITED_FILE** constants, which have the same integer value, are the same format in that in both you can use commas, semicolons, tabs, or spaces as field delimiters.

Traditionally, as the name suggests, .**csv** was applied to files in which only commas could be used as delimiters. **CSV** is therefore retained as a class constant for JADE Report Writer applications even though it has the same integer value as the **DELIMITED_FILE** constant.

getDefaultProfile

Signature `getDefaultProfile(): String;`

The **getDefaultProfile** method of the **JadeReportWriterReport** class returns a string value containing the name of the default profile for the report.

getDelimitedFileOptions

Signature

```
getDelimitedFileOptions(fieldSeparator:         Character output;
                        fieldDelimiter:         Character output;
                        extractGroupHeaders:    Boolean output;
                        extractGroupFooters:    Boolean output;
                        extractReportHeaders:   Boolean output;
                        extractReportFooters:   Boolean output;
                        combineGroupsAndDetail: Boolean output;
                        formatDates:            Boolean output;
                        formatNumerics:         Boolean output);
```

The **getDelimitedFileOptions** method of the **JadeReportWriterReport** class obtains the current values to be used when running the report for extraction to a delimited file.

These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog. If you want to override these values at run time, you can call the **setDelimitedFileOptions** method and pass as parameters the delimited file values that you require.

The parameters for the **getDelimitedFileOptions** method are listed in the following table.

Parameter	Specifies…
fieldSeparator	The delimiter that is used between each field of data (for example, a comma or a tab character)
fieldDelimiter	The character that is placed around each non-null field (and is typically a quote (') character)
extractGroupHeaders	Whether group header data is extracted to the output file
extractGroupFooters	Whether group footer data is extracted to the output file
extractReportHeaders	Whether report header data is extracted to the output file
extractReportFooters	Whether report footer data is extracted to the output file
combineGroupsAndDetail	Whether group header and detail fields are combined into one record
formatDates	Whether date fields are written as formatted strings as they are for a printed report (when **true**) or as a numeric value (when **false**)
formatNumerics	Whether numeric fields are written as formatted strings as they are for a printed report (when **true**) or as unformatted numeric values (when **false**)

If the values of both the **combineGroupsAndDetail** and **extractGroupHeaders** parameters are **true**, each extract record contains the group header fields in order and then the detail fields combined into one record.

If both the **combineGroupsAndDetail** and **extractGroupHeaders** parameters are *not* set to **true** and the **extractGroupHeaders** or **extractGroupFooters** parameter is set to **true**, group data is extracted as separate records.

getExtraParameterDetails

Signature
```
getExtraParameterDetails(parameterName:  String;
                         mandatoryValue: Boolean output;
                         userPrompt:     String output);
```

The **getExtraParameterDetails** method of the **JadeReportWriterReport** class obtains the following information for the parameter specified in the **parameterName** field.

Parameter	Returns…
mandatoryValue	Whether the entry of a value for the specified parameter is mandatory (that is, it is set to **true**)
userPrompt	The required value that is to be displayed as a user prompt

You can use this method to check that a value has been entered for the specified parameter and what is displayed in bubble help.

getFolder

Signature
```
getFolder(): String;
```

The **getFolder** method of the **JadeReportWriterReport** class returns a string containing the path to the folder in which the report is saved.

The folder path is built from the folder names at each level in the hierarchy, separated by a forward slash character (*/*). For example, a folder named **Common** within a folder named **Payroll** within the root report folder named **Reports** would be returned as the **"Reports/Payroll/Common"** folder path.

See also the **JadeReportWriterManager** class **getAllReportNames** method.

getHtmlOptions

Signature
```
getHtmlOptions(outputTOC:     Boolean output;
               fullPageWidth: Boolean output);
```

The **getHtmlOptions** method of the **JadeReportWriterReport** class obtains the current values to be used when running the report for extraction to a HyperText Markup Language (HTML) file.

These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog.

If you want to override these values at run time, you can call the **setHtmlOptions** method and pass as parameters the HTML file values that you require.

The parameters for the **getHtmlOptions** method are listed in the following table.

Parameter	Specifies…
outputTOC	Whether the HTML is generated as several files including a table of contents frame down the left-hand side
fullPageWidth	That the HTML file is built to show the page width as the screen width when set to **true** or the page width set to the same size in pixels as the paper width when set to **false**

getOutputFileTitle

Signature `getOutputFileTitle(): String;`

The **getOutputFileTitle** method of the **JadeReportWriterReport** class returns a string value containing the full title of the default output file.

getPageOptions

Signature
```
getPageOptions(documentType: Integer output;
               orientation:  Integer output;
               copies:       Integer output;
               topOfPage:    Integer output;
               bottomOfPage: Integer output;
               leftMargin:   Integer output;
               rightMargin:  Integer output;
               borderWidth:  Integer output;
               collate:      Boolean output;
               duplex:       Integer output);
```

The **getPageOptions** method of the **JadeReportWriterReport** class obtains the current **Printer** class attributes relating to paper size values to be used when running the report and directing output to the default printer.

These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog.

If you want to override these values at run time, you can call the **setPageOptions** method and pass as parameters the paper size values that you require.

The parameters for the **getPageOptions** method are listed in the following table.

Parameter	Contains the…
documentType	Printer form type (for details, see the **Printer** class **documentType** property)
orientation	Orientation of your printed output (for details, see the **Printer** class **orientation** property)
copies	Number of copies to be printed (for details, see the **Printer** class **copies** property)
topOfPage	Margin at the top of the printed page of output (for details, see the **Printer** class **topOfPage** property)
bottomOfPage	Margin at the bottom of the printed page of output (for details, see the **Printer** class **bottomOfPage** property)
leftMargin	Left margin of the printed page of output (for details, see the **Printer** class **leftMargin** property)
rightMargin	Right margin of the printed page of output (for details, see the **Printer** class **rightMargin** property)
borderWidth	Width of a border that is to be printed around report pages (for details, see the **Printer** class **pageBorderWidth** property)
collate	Collation setting for print output; that is, whether the copies are printed in proper binding order by separating copies into groups (for details, see the **Printer** class **collate** property)
duplex	Duplex setting for the report output; that is, the number of sides on which the paper is printed (for details, see the **Printer** class **duplex** property)

getParameterDetails

Signature
```
getParameterDetails(parameterName:     String;
                    type:              String output;
                    length:            Integer output;
                    scaleFactor:       Integer output;
                    value:             Any output;
                    ignoreInSelection: Boolean output;
                    isUsedInReport:    Boolean output);
```

The **getParameterDetails** method of the **JadeReportWriterReport** class obtains the following information for the parameter specified in the **parameterName** field.

Parameter	Returns…
type	Type
length	Length
scaleFactor	Scale factor
value	Current value
ignoreInSelection	When set to **true** (by using the **setParameterIgnoreInSelection** method), any selection criteria using the value specified in the **parameterName** parameter is dropped from the query (by generating **true** for the selection itself within the rest of the selection criteria)
isUsedInReport	**True** if used anywhere in the report, otherwise **false**

Use this method to write reports that have multiple selection criteria (for example, company, department, cost center, and so on) that are compared with specified parameter values and to run these reports with one or more of the selections being an **ALL** action (for example, a specific department but all cost centers). You can use the **ignoreInSelection** parameter so that you do not have to define a specific value as **all** for each primitive type. Bracketing and negation are unchanged for the selection overall.

If you use two or more parameters in a single selection (for example, **is between** or **is one of**), you must set the **ignoreInSelection** parameter for each of the parameters that are used for it to take effect. If not, the values of these parameters are used.

As the JADE Report Writer Designer application print and extract dialogs do not ask for values for parameters that are defined but not actually used anywhere, the **isUsedInReport** parameter provides that functionality. The parameter values are set when the report is designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog and by the **JadeReportWriterReport** class **setParameterIgnoreInSelection** method.

getParameters

Signature
```
getParameters(names: StringArray output;
              types: StringArray output);
```

The **getParameters** method of the **JadeReportWriterReport** class updates the **names** and **types** parameter array values with the names of parameters and the types (that is, **Primitive** or **Class** type) of those parameters, respectively. These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog.

If you want to override these values at run time, you can call the **setParameter** method and pass the appropriate parameter name and type for each individual parameter in the report.

getParametersForProfile

Signature `getParametersForProfile(profileName: String;`
`names: StringArray input;`
`types: StringArray input);`

The **getParametersForProfile** method of the **JadeReportWriterReport** class updates the **names** and **types** parameter array values with the names of parameters and the types (that is, **Primitive** or **Class** type) of those parameters used in the profile specified in the **profileName** parameter.

getProfileDescription

Signature `getProfileDescription(profileName: String): String;`

The **getProfileDescription** method of the **JadeReportWriterReport** class returns a string containing the description of the profile specified in the **profileName** parameter. These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Profile Properties dialog.

getProfiles

Signature `getProfiles(): StringArray;`

The **getProfiles** method of the **JadeReportWriterReport** class returns a string array of profile names. These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Profile Properties dialog.

If you want to override these values at run time, you can call the **setProfile** method and pass as a parameter the name of the profile to use when running the report.

A report can have a number of *profiles* associated with it. Each profile can modify the report by having different:

- Sort orders
- Selection criteria
- Grouping fields

This enables you to use one report layout for several different styles of report output. Use the **setProfile** method to set the profile that is required when the report is run.

The following example shows the use of the **getProfiles** method.

```
lstReports_click(listbox: ListBox input) updating;
vars
    repName     : String;
    jrep        : JadeReportWriterReport;
    profiles    : StringArray;
    profileName : String;
begin
    if lstReports.listIndex = -1 then
        return;
    endif;
    repName := lstReports.itemText[lstReports.listIndex];
    jrep := myJWRM.getReport(repName);
    if jrep <> null then
        create profiles transient;
```

```
            profiles := jrep.getProfiles();
            lstProfiles.clear;
            foreach profileName in profiles do
                lstProfiles.addItem(profileName);
            endforeach;
        endif;
    end;
```

getQueryOptions

Signature `getQueryOptions(concurrency: Integer output;`
`limitOption: Integer output;`
`limitValue: Integer output);`

The **getQueryOptions** method of the **JadeReportWriterReport** class returns the locking concurrency and current resource limit option values that are to be applied to the report at the query phase of the report run.

If you want to override these values at run time, you can call the **setQueryOptions** method and pass as the appropriate parameters the values that you require.

The **concurrency** parameter values used to lock objects during the query process are listed in the following table.

Class Constant	Value	Concurrency Strategy	Description
CONCURRENCY_NONE	1	Nothing	Does nothing during the query evaluation
CONCURRENCY_READ_COMMITTED	3	Read Committed	Resynchronizes every object that is accessed, but holds on to locks on collections
CONCURRENCY_READ_OPTIMISTIC	4	Repeatable Read Optimistic	Resynchronizes every object that is accessed, locks objects needed later, and checks the edition (the default value)
CONCURRENCY_READ_PESSIMISTIC	5	Repeatable Read Pessimistic	Brackets a read-only transaction to ensure that objects referenced after the start of the query and before the end of the query are the latest editions of the objects (that is, acquires an implicit shared lock on each object as it is referenced)

The **limitOption** and **limitValue** parameter values, respectively, are listed in the following table.

Class Constant	Value	Resource to Limit	Limit Value (as an Integer)
RESOURCE_LIMIT_NONE	0	No resource limits defined	Not applicable (that is, ignored), which is the default value
RESOURCE_LIMIT_READS	1	Object reads from the database	Maximum number of objects to read
RESOURCE_LIMIT_RESULTS	2	Result objects to add to the query result	Maximum number of result objects to add

Class Constant	Value	Resource to Limit	Limit Value (as an Integer)
RESOURCE_LIMIT_TIME	3	Time that the query runs	Maximum number of milliseconds the report query runs
RESOURCE_LIMIT_QUERYOPS	4	Query operations performed	Maximum number of query operations

getReportDescription

Signature `getReportDescription(): String;`

The **getReportDescription** method of the **JadeReportWriterReport** class returns the description of the report.

The report description can be searched during the open process, providing you with the ability to locate reports with specific content.

This value was set when the report was designed by using the **Description** text box on the **Details** sheet of the JADE Report Writer Designer application Report Properties dialog.

getReportingViewName

Signature `getReportingViewName(): String;`

The **getReportingViewName** method of the **JadeReportWriterReport** class returns a string value containing the name of the reporting view for the report.

getRootCollections

Signature
```
getRootCollections(collectionAliases: StringArray input;
                   collectionPaths:   HugeStringArray input);
```

The **getRootCollections** method of the **JadeReportWriterReport** class updates the **collectionAliases** and **collectionPaths** parameter array values with the user alias and path of each report root collection or join, or both root collections and view joins.

getTextOptions

Signature
```
getTextOptions(linesPerPage: Integer output;
               recordSize:   Integer output);
```

The **getTextOptions** method of the **JadeReportWriterReport** class obtains the number of lines per page and the size of each fixed-length record when running the report for extraction to a text file.

This value was set when the report was designed by using the appropriate control in the JADE Report Writer Designer application Report Properties dialog.

If you want to override this value at run time, you can call the **setTextOptions** method and pass as the parameter the number of lines per page that you require.

getUseClientFileSystem

Signature `getUseClientFileSystem(): Boolean;`

The **getUseClientFileSystem** method of the **JadeReportWriterReport** class returns **true** if the presentation client file system is used or it returns **false** if the application server file system is used.

getXmlOptions

Signature
```
getXmlOptions(reportTag: String output;
              detailTag: String output);
```

The **getXmlOptions** method of the **JadeReportWriterReport** class obtains the current values to be used when running the report for extraction to an Extensible Markup Language (XML) file.

These values were set when the report was designed by using the appropriate controls in the JADE Report Writer Designer application Report Properties dialog. If you want to override these values at run time, you can call the **setXmlOptions** method and pass as parameters the XML file values that you require.

run

Signature `run();`

The **run** method of the **JadeReportWriterReport** class runs the current report, using the parameter values specified in the ***set*** methods.

When the report is run, a call is made to the **Object** class **jadeReportWriterCheck** method for each object and the object is reported only if the **jadeReportWriterCheck** method returns **true**. Call the **run** method as the last method that you call following the completion of the setting up of your report requirements. See also the **runWithStatus** method, which runs the report, displays and refreshes a progress dialog, and returns the success of the report and the page count if the report was output to a printer or the record count if the report was extracted to a file.

The following example shows the use of the **run** method.

```
btnRun_click(btn: Button input) updating;
vars
    jrep                : JadeReportWriterReport;
    strName, strProfile : String;
    intOrient, intIndex : Integer;
begin
    if lstReports.listIndex = -1 then
        return;
    endif;
    strName := lstReports.itemText[lstReports.listIndex];
    jrep    := myJWRM.getReport(strName);
    if jrep = null then
        return;
    endif;
    if chkOrientation.value then
        intOrient := Print_Landscape;
    else
        intOrient := Print_Portrait;
    endif;
    intIndex := lstProfiles.listIndex;
    if lstProfiles.listIndex = -1 and lstProfiles.listCount > 0 then
        intIndex := 1;
```

```
    endif;
    if intIndex > 0 then
        strProfile := lstProfiles.itemText[intIndex];
        jrep.setProfile(strProfile);
    endif;
    jrep.setPreviewOptions(chkPreview.value, true, false, false, "Printing "
                           & strName);
    jrep.setPageOptions(Print_A4, intOrient, 1, 0, 0, 10, 10, 0, false, 1);
    jrep.run;
epilog
    delete jrep;
end;
```

runWithStatus

Signature

```
runWithStatus(dialogForm:   Form;
              dialogMethod: Method;
              status:       Integer output;
              outputCount:  Integer output);
```

The **runWithStatus** method of the **JadeReportWriterReport** class runs the current report, using the parameter values specified in the ***set*** methods, displaying the progress dialog form specified in the **dialogForm** parameter, which is refreshed by the method specified in the **dialogMethod** parameter.

The **dialogForm** and **dialogMethod** parameters are optional. If you set the **dialogForm** parameter to **null**, the value of the **dialogMethod** parameter is ignored and the report is run in the same way as the **JadeReportWriterReport** class **run** method, but the values of the **status** and **outputCount** parameters are returned.

If you specify a **dialogForm** and **dialogMethod** parameter value, the method specified in the **dialogMethod** parameter must be defined on the form instance specified in the **dialogForm** parameter and it must have two **String** parameters. When the JADE Report Writer calls the specified method, the first parameter is a descriptive string of the current phase of the report process. When called in the query phase, the second parameter contains the number of database objects that have been read. When called during the printing phase, the second parameter contains the number of pages that have been printed or records that have been extracted. In the printing phase, the page number is returned every 20 pages to the method specified by the **dialogMethod** parameter, and the standard JADE print progress dialog is not shown.

Any value returned from the method specified by the **dialogMethod** parameter is cast as a **Boolean**. If the resulting value is **false**, the report continues to run. If the value is **true**, the report is stopped. This enables you to cancel a report.

If the method specified in the **dialogMethod** parameter is not defined for the **Form** class specified in the **dialogForm** parameter or it does not contain two **String** parameters, the specified method is not called.

When the report is run, a call is made to the **Object** class **jadeReportWriterCheck** method for each object and the object is reported only if the **jadeReportWriterCheck** method returns **true**.

Call the **runWithStatus** method as the last method that you call following the completion of the setting up of your report requirements. (See also the **JadeReportWriterReport** class **run** method, which runs the current report without a progress dialog and with no return values.)

When the report has been run, the **status** parameter contains zero (**0**) if the report ran successfully, minus 1 (**-1**) if the report run was cancelled, or the appropriate exception code (for example, error code **15017**, or **Print_Not_Available**, if the specified printer does not match the available printers).

The **outputCount** parameter contains the page count if the report was output to a printer, or it contains the record count if the report was extracted to a file.

The following example shows the use of the **runWithStatus** method.

```
btnRun_click(btn: Button input) updating;
vars
    dialogForm          : MyDialogForm;
    dialogMethod        : Method;
    jrep                : JadeReportWriterReport;
    strName, strProfile : String;
    intOrient, intIndex : Integer;
    stat, count         : Integer;
begin
    if lstReports.listIndex = -1 then
        return;
    endif;
    create dialogForm transient;
    dialogMethod := dialogForm.class.getMethod("pageUpdate ");
    strName      := lstReports.itemText[lstReports.listIndex];
    jrep         := myJWRM.getReport(strName);
    if jrep = null then
        return;
    endif;
    if chkOrientation.value then
        intOrient := 2;          // landscape
    else
        intOrient := 1;          // portrait
    endif;
    intIndex := lstProfiles.listIndex;
    if lstProfiles.listIndex = -1 and lstProfiles.listCount > 0 then
        intIndex   := 1;
    endif;
    intIndex := lstProfiles.listIndex;
    if lstProfiles.listIndex = -1 and lstProfiles.listCount > 0 then
        intIndex   := 1;
    endif;
    if intIndex > 0 then
        strProfile := lstProfiles.itemText[intIndex];
        jrep.setProfile(strProfile);
    endif;
    jrep.setPreviewOptions(chkPreview.value, true, false, true, "");
    jrep.setPageOptions(Print_A4, intOrient, 1, 0, 0, 10, 10, 0, true, 1);
    jrep.runWithStatus(dialogForm, dialogMethod, stat, count);
epilog
    delete jrep;
    if dialogForm <> null then
        dialogForm.unloadForm;
    endif;
end;
```

setDelimitedFileOptions

Signature
```
setDelimitedFileOptions(fieldSeparator:         Character;
                        fieldDelimiter:         Character;
                        extractGroupHeaders:    Boolean;
                        extractGroupFooters:    Boolean;
                        extractReportHeaders:   Boolean;
                        extractReportFooters:   Boolean;
                        combineGroupsAndDetail: Boolean;
                        formatDates:            Boolean;
                        formatNumerics:         Boolean);
```

The **setDelimitedFileOptions** method of the **JadeReportWriterReport** class sets the values to be used when running the report for extraction to a delimited file. (Use the **getDelimitedFileOptions** method to obtain the current values set up for a delimited file.)

The parameters for the **setDelimitedFileOptions** method are listed in the following table.

Parameter	Description
fieldSeparator	Specifies the delimiter that is used between each field of data (for example, a comma or a tab character)
fieldDelimiter	Specifies the character that is placed around each non-null field (typically a quote (‘) character)
extractGroupHeaders	Specifies whether group header data is extracted to the output file
extractGroupFooters	Specifies whether group footer data is extracted to the output file
extractReportHeaders	Specifies whether report header data is extracted to the output file
extractReportFooters	Specifies whether report footer data is extracted to the output file
combineGroupsAndDetail	Specifies whether group header and detail fields are combined into one record
formatDates	Specifies whether date fields are written as formatted strings as they are for a printed report (when **true**) or as a numeric value (when **false**)
formatNumerics	Specifies whether numeric fields are written as formatted strings as they are for a printed report (when **true**) or as unformatted numeric values (when **false**)

If the values of both the **combineGroupsAndDetail** and **extractGroupHeaders** parameters are **true**, each extract record contains the group header fields in order and then the detail fields combined into one record. If both the **combineGroupsAndDetail** and **extractGroupHeaders** parameters are *not* set to **true** and the **extractGroupHeaders** or **extractGroupFooters** parameter is set to **true**, group data is extracted as separate records.

setEndingNotification

Signature
```
setEndingNotification(notificationObject: Object;
                      eventType:          Integer);
```

The **setEndingNotification** method of the **JadeReportWriterReport** class specifies the object that is notified when the report finishes.

An event of type **eventType** is raised if the object specified in the **notificationObject** parameter is a valid object and the value specified in the **eventType** is a valid user 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)

setHtmlOptions

Signature
```
setHtmlOptions(printTOC:       Boolean;
               fullPageWidth: Boolean);
```

The **setHtmlOptions** method of the **JadeReportWriterReport** class sets the current values to be used when running the report for extraction to a HyperText Markup Language (HTML) file.

Use the **getHtmlOptions** method to obtain the current values set up for extraction to HTML.

The parameters for the **setHtmlOptions** method are listed in the following table.

Parameter	Description
printTOC	Specifies whether HTML is generated as several files including a table of contents frame down the left-hand side.
fullPageWidth	Specifies that the HTML file is built to show the page width as the screen width when set to **true** or the page width is set to the same size in pixels as the paper width when set to **false**

setLocaleDateOptions

Signature
```
setLocaleDateOptions(dateOrder:         Integer;
                     dayDateSeparator: String;
                     dateSeparator:    String): Boolean;
```

The **setLocaleDateOptions** method of the **JadeReportWriterReport** class sets the format for **Date** fields to be used when running the report. The method overrides default date settings specified on the System Formats dialog, the Report Formats dialog, and the Field Properties dialog for date fields.

The parameters for the **setLocaleDateOptions** method are listed in the following table.

Parameter	Description
dateOrder	An integer specifying the order in which the day, month, and year are displayed. The following constants of the **DateFormat** class can be used. ■ **DayMonthYear** (**1**) ■ **MonthDayYear** (**0**) ■ **YearMonthDay** (**2**)
dayDateSeparator	A string with a maximum of three characters that is displayed between the day name and the date.
dateSeparator	A string with a maximum of three characters that is displayed between the day, the month, and the year.

If a parameter has an invalid value (for example, a string has too many characters), the method returns **false** and default formats are used.

Overriding does not take place for fields where the report or field format value is **null**. For example, if the date separator for a field is set to **null** (to concatenate the day, month, and year values) calling the **setLocaleDateOptions** method with a value of "**-**" for the **dateSeparator** parameter would *not* insert the separator characters.

If you do not want to override a format setting, set the value of the corresponding parameter to **null**. In the following code fragment, the **setLocaleDateOptions** method is used to override the date order only.

```
report.setLocaleDateOptions(DateFormat.DayMonthYear, null, null);
```

setLocaleNumericOptions

Signature
```
setLocaleNumericOptions(decimalSeparator:  String;
                        thousandSeparator: String;
                        currencySymbol:    String): Boolean;
```

The **setLocaleNumericOptions** method of the **JadeReportWriterReport** class sets the format for number or currency fields to be used when running the report. The method overrides default number and currency settings specified on the System Formats dialog, the Report Formats dialog, and the Field Properties dialog for number and currency fields.

The parameters for the **setLocaleNumericOptions** method are listed in the following table.

Parameter	Description
decimalSeparator	A string with a maximum of three characters that is used to separate decimal part of a number from the rest.
thousandSeparator	A string with a maximum of three characters that is used as the thousands separator.
currencySymbol	A string with a maximum of five characters that is used as the currency symbol.

If a parameter has an invalid value (for example, a string has too many characters), the method returns **false** and default formats are used.

Overriding does not take place for fields where the report or field format value is **null**. For example, if the thousands separator for a number field is set to **null** in a report, calling the **setLocaleNumericOptions** method with a **thousandSeparator** parameter value of **","** would *not* insert the separator characters.

If you do not want to override a format setting, set the value of the corresponding parameter to **null**. In the following code fragment, the **setLocaleNumericOptions** method is used to override the currency symbol only.

```
report.setLocaleNumericOptions(null, null, "$");
```

setLocaleTimeOptions

Signature
```
setLocaleTimeOptions(timeSeparator: String;
                     timeAMText:    String;
                     timePMText:    String): Boolean;
```

The **setLocaleTimeOptions** method of the **JadeReportWriterReport** class sets the format for **Time** fields to be used when running the report. The method overrides default time settings specified on the System Formats dialog, the Report Formats dialog, and the Field Properties dialog for time fields.

The parameters for the **setLocaleTimeOptions** method are listed in the following table.

Parameter	Description
timeSeparator	A string with a maximum of 10 characters that is displayed between the hours, minutes, and seconds.
timeAMText	A string with a maximum of 30 characters that is displayed for times before midday.
timePMText	A string with a maximum of 30 characters that is displayed for times after midday.

If a parameter has an invalid value (for example, a string has too many characters), the method returns **false** and default formats are used.

Overriding does not take place for fields where the report or field format value is **null**. For example, if the time separator for a field is set to **null** (to concatenate the hours, minutes, and seconds values), calling the **setLocaleTimeOptions** method with a **timeSeparator** parameter value of **"-"** would *not* insert the separator characters.

If you do not want to override a format setting, set the value of the corresponding parameter to **null**. In the following code fragment, the **setLocaleTimeOptions** method is used to override the time separator only.

```
report.setLocaleDateOptions(" :: ", null, null);
```

setOutputDestination

Signature `setOutputDestination(destination: Integer);`

The **setOutputDestination** method of the **JadeReportWriterReport** class sets the output destination of the report. The **destination** parameter specifies the output destination that you require, represented by one of the **JadeReportWriterReport** class constants listed in the following table.

Class Constant	Integer Value	Description
CSV	3	Comma-Separated Value (CSV) file format
DELIMITED_FILE	3	Delimited file format (for example, containing commas, semicolons, tabs, or spaces)
HTML	2	HyperText Markup Language (HTML) file
PRINTER	0	Output to the default printer of the user
RTF	4	Rich Text File (RTF) file format for Microsoft Word for Windows
TEXT	6	Text (or ASCII) file format, and can contain new page characters
XML	5	Extensible Markup Language (XML) file format

setOutputFileTitle

Signature `setOutputFileTitle(fileTitle: String);`

The **setOutputFileTitle** method of the **JadeReportWriterReport** class sets the title of the output file that is created when the report runs to the value specified in the **fileTitle** parameter.

For details about the valid output file types, see the **setOutputDestination** method.

setPageOptions

Signature

```
setPageOptions(documentType: Integer;
               orientation:  Integer;
               copies:       Integer;
               topOfPage:    Integer;
               bottomOfPage: Integer;
               leftMargin:   Integer;
               rightMargin:  Integer;
               borderWidth:  Integer;
               collate:      Boolean;
               duplex:       Integer);
```

The **setPageOptions** method of the **JadeReportWriterReport** class sets the **Printer** class attributes relating to paper size values to be used when running the report and directing output to the default printer.

Use the **getPageOptions** method to obtain the current paper size values and the **setPreviewOptions** method to set the preview options that you require.

Your report is output to the current printer. Use the functionality provided by the **Printer** class (for example, **app.printer.setPrinter**) to handle the transient instance of the **Printer** class at run time.

The parameters for the **setPageOptions** method are listed in the following table.

Parameter	Description
documentType	Contains the printer form type (for details, see the **Printer** class **documentType** property)
orientation	Contains the orientation of your printed output (for details, see the **Printer** class **orientation** property)
copies	Contains the number of copies to be printed (for details, see the **Printer** class **copies** property)
topOfPage	Contains the margin at the top of the printed page of output (for details, see the **Printer** class **topOfPage** property)
bottomOfPage	Contains the margin at the bottom of the printed page of output (for details, see the **Printer** class **bottomOfPage** property)
leftMargin	Contains the left margin of the printed page of output (for details, see the **Printer** class **leftMargin** property)
rightMargin	Contains the right margin of the printed page of output (for details, see the **Printer** class **rightMargin** property)
borderWidth	Contains the width of a border that is to be printed around report pages (for details, see the **Printer** class **pageBorderWidth** property)
collate	Contains the collation setting for print output; that is, whether the copies are printed in proper binding order by separating copies into groups (for details, see the **Printer** class **collate** property)
duplex	Contains the duplex setting for the report output; that is, the number of sides on which the paper is printed (for details, see the **Printer** class **duplex** property)

The following example shows the use of the **setPageOptions** method.

```
btnRun_click(btn: Button input) updating;
vars
```

```
    jrep                : JadeReportWriterReport;
    strName, strProfile : String;
    intOrient, intIndex : Integer;
    stat, count         : Integer;
begin
    if lstReports.listIndex = -1 then
        return;
    endif;
    strName := lstReports.itemText[lstReports.listIndex];
    jrep := myJWRM.getReport(strName);
    if jrep = null then
        return;
    endif;
    if chkOrientation.value then
        intOrient := 2;          // landscape
    else
        intOrient := 1;          // portrait
    endif;
    intIndex := lstProfiles.listIndex;
    if lstProfiles.listIndex = -1 and lstProfiles.listCount > 0 then
        intIndex := 1;
    endif;
    if intIndex > 0 then
        strProfile := lstProfiles.itemText[intIndex];
        jrep.setProfile(strProfile);
    endif;
    jrep.setPreviewOptions(chkPreview.value, true, false, false,
                           "Printing " & strName);
    jrep.setPageOptions(Print_A4, intOrient, 1, 0, 0, 10, 10, 0, true, 1);
    jrep.run;
end;
```

setParameter

Signature

```
setParameter(name:  String;
             value: Any);
```

The **setParameter** method of the **JadeReportWriterReport** class sets the name and value of a parameter to be used when running the report to the values specified in the **name** and **value** parameters, respectively.

Use the **getParameters** method to obtain the names and types of all individual parameters that can be used when the report is run.

setParameterIgnoreInSelection

Signature

```
setParameterIgnoreInSelection(parameterName: String;
                              option:        Boolean output);
```

The **setParameterIgnoreInSelection** method of the **JadeReportWriterReport** class specifies whether the value specified in the **parameterName** parameter is to be ignored when selecting parameters for the current run of the report, by setting the **option** parameter to **true**.

Any selection criterion using the specified parameter is dropped from the query (by generating **true** for the selection itself within the rest of the selection criteria).

Use the **getParameterDetails** method to write reports that have multiple selection criteria (for example, company, department, cost center, and so on) that are compared with specified parameter values, and to run these reports with one or more of the selections being an **ALL** action (for example, a specific department but all cost centers). So that you do not have to define a specific value as **all** for each primitive type, you can use the appropriate *ignore in selection* value for each parameter.

If you use two or more parameters in a single selection (for example, **is between** or **is one of**), you must set the **ignoreInSelection** parameter of the **getParameterDetails** method for each of the parameters that are used for it to take effect. If not, the values of these parameters are used.

You can set the *ignore in selection* value for each parameter by checking or unchecking the check box for each parameter name on the **Parameters** sheet of the **Report Properties** dialog in the JADE Report Writer Designer application.

setPreviewOptions

Signature

```
setPreviewOptions(printPreview:      Boolean;
                  previewAllowPrint: Boolean;
                  previewReduce:     Boolean;
                  suppressDialog:    Boolean;
                  title:             String);
```

The **setPreviewOptions** method of the **JadeReportWriterReport** class sets the **Printer** class preview attributes to be used when running the report and directing output to a printer.

Use the **setPageOptions** method to set the paper size values.

The parameters for the **setPreviewOptions** method are listed in the following table.

Parameter	Description
printPreview	Specifies whether the printed output is to be directed to the preview file (for details, see the **Printer** class **printPreview** property)
previewAllowPrint	Specifies whether previewed output can be directed to the printer (for details, see the **Printer** class **printPreviewAllowPrint** property)
previewReduce	Specifies whether previewed output is reduced to display a full page on the screen (for details, see the **Printer** class **printPreviewReduce** property)
suppressDialog	Specifies whether the system-supplied print progress dialog is to be displayed (for details, see the **Printer** class **suppressDialog** property)
title	Contains the title to be displayed on the system-supplied print progress dialog (for details, see the **Printer** class **title** property)

The following example shows the use of a **JadeScript** class **setPreviewOptions** method.

```
runReport();
vars
    jrwm : JadeReportWriterManager;
    jrep : JadeReportWriterReport;
begin
    create jrwm transient;
    jrep := jrwm.getReport("Employees - ABC");
    if jrep = null then
        return;
    endif;
```

```
    jrep.setPreviewOptions(true, true, false, false, "Printing");
    jrep.run;
epilog
    delete jrwm;
    delete jrep;
end;
```

setProfile

Signature `setProfile(profileName: String);`

The **setProfile** method of the **JadeReportWriterReport** class sets the name of the report profile to use when running the report. If the name specified in the **profileName** parameter is not valid, the current profile is used.

Use the **getProfiles** method to return an array of the profile names that can be used when the report is run.

The code fragment in the following example shows the use of the **setProfile** method.

```
intIndex := lstProfiles.listIndex;
if lstProfiles.listIndex = -1 and lstProfiles.listCount > 0 then
    intIndex := 1;
endif;
if intIndex > 0 then
    strProfile := lstProfiles.itemText[intIndex];
    jrep.setProfile(strProfile);
endif;
```

setQueryOptions

Signature
```
setQueryOptions(concurrency:        Integer;
                resourceLimitType:  Integer;
                resourceLimitValue: Integer);
```

The **setQueryOptions** method of the **JadeReportWriterReport** class sets specific limits in the execution of a query when running the report.

The **concurrency** parameter values used to lock objects during the query process are listed in the following table.

Class Constant	Value	Concurrency Strategy	Description
CONCURRENCY_NONE	1	Nothing	Does nothing during the query evaluation
CONCURRENCY_READ_COMMITTED	3	Read Committed	Resynchronizes every object that is accessed, but holds on to locks on collections
CONCURRENCY_READ_OPTIMISTIC	4	Repeatable Read Optimistic	Resynchronizes every object that is accessed, locks objects needed later, and checks the edition (the default value)

Class Constant	Value	Concurrency Strategy	Description
CONCURRENCY_READ_PESSIMISTIC	5	Repeatable Read Pessimistic	Brackets a read-only transaction to ensure that objects referenced after the start of the query and before the end of the query are the latest editions of the objects (that is, acquires an implicit shared lock on each object as it is referenced)

The **resourceLimitType** and **resourceLimitValue** parameter values, respectively, are listed in the following table.

Class Constant	Value	Resource to Limit	Limit Value (as an Integer)
RESOURCE_LIMIT_NONE	0	No resource limits defined	Not applicable (that is, ignored), which is the default value
RESOURCE_LIMIT_READS	1	Object reads from the database	Maximum number of objects to read
RESOURCE_LIMIT_RESULTS	2	Result objects to add to the query result	Maximum number of result objects to add
RESOURCE_LIMIT_TIME	3	Time that the query runs	Maximum number of milliseconds the report query runs
RESOURCE_LIMIT_QUERYOPS	4	Query operations performed	Maximum number of query operations

Use the **getQueryOptions** method to get the locking concurrency and resource limits that are to be applied at the query phase of the report run.

setStartEndMethods

Signature

```
setStartEndMethods(initializeMethodName: String;
                   finalizeMethodName:   String);
```

The **setStartEndMethods** method of the **JadeReportWriterReport** class sets the names of the **Application** class methods that are to be called before the report is run (for example, to set up context for the report) and after the report has completed (for example, to delete transient instances). Use the **initializeMethodName** parameter to specify the **initialize** method and the **finalizeMethodName** parameter to specify the **finalize** method for the report, if required.

Null parameter values indicate that you do not want **initialize** and **finalize** method calls for this report run.

setTextOptions

Signature

```
setTextOptions(linesPerPage: Integer;
               recordSize:   Integer);
```

The **setTextOptions** method of the **JadeReportWriterReport** class sets the number of lines per page and the size of each fixed-length record that you require when running the report for extraction to a text file. (Use the **getTextOptions** method to obtain the current lines per page for report output extracted to a text file.)

If you specify a value greater than zero (**0**) in the **linesPerPage** parameter, a new page character is written to the file when the specified number of lines has been written to the file. No new page characters are written to the file when the **linesPerPage** parameter value is zero (**0**).

setUseClientFileSystem

Signature `setUseClientFileSystem(clientFiles: Boolean);`

The **setUseClientFileSystem** method of the **JadeReportWriterReport** class specifies whether the presentation client file system is used (when the **clientFiles** parameter is set to **true**) or the application server file system is used (when the **clientFiles** parameter is set to **false**).

setXmlOptions

Signature
```
setXmlOptions(reportTag: String;
              detailTag: String);
```

Use the **setXmlOptions** method of the **JadeReportWriterReport** class to set the values to be used when running the report for extraction to an Extensible Markup Language (XML) file.

The value specified in the **reportTag** parameter is placed around the whole file and the value specified in the **detailTag** parameter is placed around each detail line.

Group headers are written using the group alias as a tag that is terminated by each group footer. Each data item field within the class specified for each detail tag is written using the field title as the tag and the field data as data.

Use the **getXmlOptions** method to obtain the current XML file options.

JadeReportWriterSecurity Class

The **JadeReportWriterSecurity** class provides a superclass for all user **JadeReportWriterSecurity** subclasses. Report folders containing the JADE Report Writer reports can be unsecured so that all users have access to them or you can dynamically define runtime access to folders by implementing the required security rules in a subclass of the **JadeReportWriterSecurity** class in the schema in which the report is defined.

The methods defined in the **JadeReportWriterSecurity** class are used in the JADE Report Writer Configuration application to control access to views and folders and in the Designer application to control access to reports.

If you want to reimplement instance-based security in a JADE Report Writer report at run time, subclass the **JadeReportWriterSecurity** class and reimplement the methods defined in that class to use the security mechanism to return an integer value that indicates the type of access that the user has.

For details about the class constants provided by the **JadeReportWriterSecurity** class that are returned by these methods to indicate the type of access that a specified user has, see "JadeReportWriterSecurity Class Constants", in the following section. For an overview of the JADE Report Writer and the relationship of the **JadeReportWriterManager**, **JadeReportWriterReport**, and **JadeReportWriterSecurity** classes, see "JadeReportWriterManager Class", earlier in this chapter. For details about dynamically running an existing JADE Report Writer report, see "Running an Existing Report" under "JadeReportWriterManager Class", earlier in this chapter.

For details about the constants and methods defined in the **JadeReportWriterSecurity** class, see "JadeReportWriterSecurity Class Constants" and "JadeReportWriterSecurity Methods", in the following subsections.

Inherits From: Object

Inherited By: (None)

JadeReportWriterSecurity Class Constants

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

Class Constant	Integer Value	Description
FULL_ACCESS	2	Allows full access to the report for definition and use
NO_ACCESS	0	No access is allowed to the report
READ_ONLY_ACCESS	1	The report can be accessed and run, but the definitions cannot be changed

When you implement security in your JADE Report Writer reports, these values are returned by the methods in your subclass of the **JadeReportWriterSecurity** class, to indicate the type of access that the user has.

JadeReportWriterSecurity Methods

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

Method	Description
canAccessConfiguration	Returns the type of access that the specified user has to the Configuration application

Method	Description
canAccessDesigner	Returns the type of access that the specified user has to the Designer application
canAccessFolder	Returns the type of access that the specified user has to the specified folder in the Configuration application
canAccessReport	Returns the type of access that the specified user has to the specified report when reports are listed for selection
canAccessView	Returns the type of access that the specified user has to the specified view when reports are listed for selection or extraction
canAccessViewClass	Controls visibility of view classes in the Designer application
canAccessViewFeature	Controls visibility of view features in the Designer application
canDeleteReport	Restricts the reports that a specified user can delete
canMaintainFolders	Returns the type of access that the specified user has to folders in the Configuration application
canMaintainSystemOptions	Returns the type of access that the specified user has to system options in the Configuration application
canMaintainViews	Returns the type of access that the specified user has to views in the Configuration application
folderDeleted	Called when a folder is deleted
folderPathChanged	Called when a folder path is changed
isViewFeatureAccessSet	Specifies whether the user can access the view
newFolderAdded	Enables the user who created a new folder to access that folder when security is set
newReportAdded	Enables the user who created a new report to access that report when security is set
newViewAdded	Enables the user who created a new view to access that view when security is set
reportDeleted	Called when a report is deleted
reportNameChanged	Called when a report name is changed
viewDeleted	Called when a view is deleted
viewNameChanged	Called when the name of a view is changed

canAccessConfiguration

Signature `canAccessConfiguration(userName: String): Integer;`

The **canAccessConfiguration** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the JADE Report Writer Configuration application when it is invoked by that user.

The following example shows how to control access to a JADE Report Writer application, in this case the JADE Report Writer Configuration application.

```
canAccessConfiguration(userName: String): Integer;
begin
    if process.userCode = "USER1" then
        return FULL_ACCESS;
    elseif process.userCode = "USER2" then
        return READ_ONLY_ACCESS;
    else
        return NO_ACCESS;
    endif;
end;
```

In this example, **USER1** has full access to the JADE Report Writer Configuration application, which means that **USER1** can create and change reporting views. **USER2** has read-only access, and every other user is prevented from running the JADE Report Writer Configuration application.

By default, users have full access to the JADE Report Writer Configuration application (that is, they can both access and change views, folders, and system options). When the return value of this method is checked at the time the application is invoked, the application is not started if the user has no access. If the user has read-only access, the application is started in read-only mode and the user can access but not change views, folders, and system options. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessDesigner

Signature `canAccessDesigner(userName: String): Integer;`

The **canAccessDesigner** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the JADE Report Writer Designer application when it is invoked by that user.

By default, users have full access to the JADE Report Writer Designer application (that is, they can both access and change reports and templates). When the return value of this method is checked at the time the application is invoked, the application is not started if the user has no access. If the user has read-only access, the application is started in read-only mode and the user can access but not change reports and templates. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessFolder

Signature
```
canAccessFolder(userName:   String;
                folderPath: String): Integer;
```

The **canAccessFolder** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the folder specified in the **folderPath** parameter when the user attempts to access that folder.

By default, users have full access to folders in the JADE Report Writer Configuration application. When the return value of this method is checked at the time the folder is accessed, the folder is not displayed if the user has no access. If the user has read-only access, contents of the folder are displayed in read-only mode and the user cannot change values.

The folder path is built from the folder names at each level in the hierarchy, separated by a forward slash character (*/*). Template paths start with a forward slash, but report paths do not. For example:

- A folder named **Common** within a folder named **Payroll** within the root report folder named **Reports** has a folder path of *Reports/Payroll/Common*.
- A folder named **Secured** within a folder named **Sales** within the root template folder named **Templates** has a folder path of */Templates/Sales/Secured*.

The following example shows the **canAccessFolder** method to control access to folders.

```
canAccessFolder(userName: String; folderPath: String): Integer;
begin
    if folderPath = "/Reports/Clients" then
        if process.userCode = "USER1" or
            process.userCode = "USER2" then
            return 2;
        else
            return 0;
        endif;
    else
        return 2;
    endif;
end;
```

In this example, **USER1** and **USER2** have full access to the **Clients** folder. All other users have no access to the **Clients** folder but they have full access to folders other than the **Clients** folder.

For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessReport

Signature

```
canAccessReport(userName:   String;
                reportName: String): Integer;
```

The **canAccessReport** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the report specified in the **reportName** parameter when the user attempts to access that report. As the access for the user is checked in the standard common Open dialog for reports, only reports to which the user has the appropriate type of access are displayed in the list of available reports.

By default, users have full access to JADE Report Writer reports. When the return value of this method is checked at the time reports are displayed for selection, reports to which the user does not have the appropriate type of access are not listed as available for selection. If the user has read-only access, displayed reports can be accessed only in read-only mode and the user cannot change values. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessView

Signature

```
canAccessView(userName: String;
              viewName: String): Integer;
```

The **canAccessView** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the view specified in the **viewName** parameter when the user attempts to access a report over that view.

As the access for the user is checked in the standard common Open dialog for reports over views to which the user has access, only reports over the views to which the user has the appropriate type of access are displayed in the list of available reports in the New Report dialog when building a list of views to report against and in the View Extract dialog when building a list of views to extract.

By default, users have full access to JADE Report Writer reports over all views. When the return value of this method is checked at the time reports are displayed for selection, reports to which the user does not have the appropriate type of access are not listed as available for selection. If the user has read-only access, displayed reports can be accessed only in read-only mode, and the user cannot change values. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessViewClass

Signature

```
canAccessViewClass(userName:  String;
                   viewName:  String;
                   className: String): Integer;
```

The **canAccessViewClass** method of the **JadeReportWriterSecurity** class controls the visibility of view classes in the JADE Report Writer Designer application. This method returns the type of access that the user specified in the **userName** parameter has to the view specified in the **viewName** parameter for the class specified in the **className** parameter when the user attempts to access a report over that view class.

Reports can be based only on root collections over classes to which the specified user has access, and references to classes at the time the report is designed are displayed only if access to the referenced class is allowed by this method.

By default, users have full access to views over all classes in the JADE Report Writer Designer application. When the return value of this method is checked at the time class references are displayed for selection, classes to which the user does not have the appropriate type of access are not listed as available for selection. If the user has read-only access, displayed classes can be accessed only in read-only mode, and the user cannot change values. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canAccessViewFeature

Signature

```
canAccessViewFeature(userName:    String;
                     viewName:    String;
                     className:   String;
                     featureName: String): Integer;
```

The **canAccessViewFeature** method of the **JadeReportWriterSecurity** class controls the visibility of view class features in the JADE Report Writer Designer application.

This method returns the type of access that the user specified in the **userName** parameter has to the view specified in the **viewName** parameter for the feature specified in the **featureName** parameter class of the class specified in the **className** parameter when the user attempts to access a report over that view class feature.

Because of the extra processing that is required, the report writer calls the **canAccessViewFeature** method every time it builds a list of class features only when the **isViewFeatureAccessSet** method returns **true**.

This method controls visibility of view class features in the:

- JADE Report Writer Designer application
- Catalog when painting an item on a report

- Group properties when selecting a group item
- Script form
- Profile selection
- Sort criteria

By default, users have full access to views over all features in all classes in the JADE Report Writer Designer application. When the return value of this method is checked at the time class features are accessed, class features to which the user does not have the appropriate type of access are not available for selection. If the user has read-only access, class features can be accessed only in read-only mode, and the user cannot change values. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canDeleteReport

Signature `canDeleteReport(userName: String; reportName: String): Integer;`

The **canDeleteReport** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to the report specified in the **reportName** parameter when the user attempts to delete the report. This method is automatically called from the JADE Report Writer Designer application for the Delete Report form before displaying reports that are available for deletion.

If the method does not return the **JadeReportWriterSecurity** class **FULL_ACCESS** constant value, the report is not displayed and therefore cannot be deleted. As all users can delete all reports by default, the default implementation of this method returns **FULL_ACCESS**. Reimplement this method in your **JadeReportWriterSecurity** subclass to return **NO_ACCESS** (**0**) for reports that specified users cannot delete.

canMaintainFolders

Signature `canMaintainFolders(userName: String): Integer;`

The **canMaintainFolders** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to folders in the JADE Report Writer Configuration application. Access to folders is checked when the user attempts to load, extract, or maintain folders. By default, users have full access to folders in the JADE Report Writer Configuration application.

When the return value of the **canMaintainFolders** method is checked at the time access to folders is attempted, folders are not invoked if the user has no access. If the user has read-only access, the folders are displayed in read-only mode, and the user cannot change values.

The following example shows applying security to the use of JADE Report Writer folders.

```
canMaintainFolders(userName: String): Integer;
begin
    if process.userCode = "USER1" then
        return FULL_ACCESS;
    elseif process.userCode = "USER2" then
        return READ_ONLY_ACCESS;
    else
        return NO_ACCESS;
    endif;
end;
```

For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canMaintainSystemOptions

Signature `canMaintainSystemOptions(userName: String): Integer;`

The **canMaintainSystemOptions** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to system options in the JADE Report Writer Configuration application. Access to system options is checked when the user attempts to load, extract, or maintain system options.

By default, users have full access to system options in the JADE Report Writer Configuration application. When the return value of this method is checked at the time access to system options is attempted, options are not displayed if the user has no access. If the user has read-only access, system options are displayed in read-only mode, and the user cannot change values.

For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

canMaintainViews

Signature `canMaintainViews(userName: String): Integer;`

The **canMaintainViews** method of the **JadeReportWriterSecurity** class returns the type of access that the user specified in the **userName** parameter has to views in the JADE Report Writer Configuration application. Access to views is checked when the user attempts to build, delete, extract, load, create, open, or validate views.

By default, users have full access to views in the JADE Report Writer Configuration application. When the return value of this method is checked at the time access to views is attempted, views are not invoked if the user has no access. If the user has read-only access, views are displayed in read-only mode, and the user cannot change values. For details about the integer values that can be returned by this method, see "JadeReportWriterSecurity Class Constants", earlier in this chapter.

folderDeleted

Signature `folderDeleted(folderPath: String);`

The **folderDeleted** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the folder of the receiver is deleted, passing the folder path as the **folderPath** parameter value.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the value specified in the method parameter.

folderPathChanged

Signature
```
folderPathChanged(oldPath: String;
                  newPath: String);
```

The **folderPathChanged** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the folder path of the receiver is changed, passing the existing folder path in the **oldPath** parameter and the new folder path in the **newPath** parameter.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters.

isViewFeatureAccessSet

Signature `isViewFeatureAccessSet(userName: String; viewName: String): Boolean;`

The **isViewFeatureAccessSet** method of the **JadeReportWriterSecurity** class specifies whether the user specified in the **userName** parameter can access the view specified in the **viewName** parameter.

Because of the extra processing that is required, the report writer calls the **canAccessViewFeature** method every time it builds a list of class features only when the **isViewFeatureAccessSet** method returns **true**.

newFolderAdded

Signature `newFolderAdded(userName: String; folderPath: String);`

The **newFolderAdded** method of the **JadeReportWriterSecurity** class enables the user who created a new folder to subsequently access that folder when security is set. When a user creates a new folder, this method is called by the JADE Report Writer Configuration application, passing the current user name and folder path as values to the **userName** and **folderPath** parameters.

You can reimplement this method in the **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters so that the user can then access and maintain that folder.

newReportAdded

Signature `newReportAdded(userName: String; reportName: String);`

The **newReportAdded** method of the **JadeReportWriterSecurity** class enables the user who created a new report to subsequently access that report when security is set. When a user creates a new report, this method is called by the JADE Report Writer Configuration application, passing the current user name and report name as values to the **userName** and **reportName** parameters.

You can reimplement this method in the **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters so that the user can then access and maintain that report.

newViewAdded

Signature `newViewAdded(userName: String; viewName: String);`

The **newViewAdded** method of the **JadeReportWriterSecurity** class enables the user who created a new view to subsequently access that view when security is set. When a user creates a new view, this method is called by the JADE Report Writer Configuration application, passing the current user name and view name as values to the **userName** and **viewName** parameters.

You can reimplement this method in the **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters so that the user can then access and maintain that view.

reportDeleted

Signature `reportDeleted(reportName: String);`

The **reportDeleted** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the report of the receiver is deleted, passing the name of the report as the **reportName** parameter value.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the value specified in the method parameter.

reportNameChanged

Signature
```
reportNameChanged(oldName: String;
                  newName: String);
```

The **reportNameChanged** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the report name of the receiver is changed, passing the existing report name in the **newName** parameter and the new report name in the **newName** parameter.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters.

viewDeleted

Signature `viewDeleted(viewName: String);`

The **viewDeleted** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the view of the receiver is deleted, passing the name of the view as the **viewName** parameter value.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the value specified in the method parameter.

viewNameChanged

Signature
```
viewNameChanged(oldName: String;
                newName: String);
```

The **viewNameChanged** method of the **JadeReportWriterSecurity** class is called by the JADE Report Writer Configuration application when the view name of the receiver is changed, passing the existing view name in the **newName** parameter and the new view name in the **newName** parameter.

You can reimplement this method in your **JadeReportWriterSecurity** subclass, to automatically create a security access entry for the values specified in the method parameters.

JadeRestService Class

The **JadeRestService** class maintains all Internet REST service provider information.

A transient copy of a subclass is created by each Representational State Transfer (REST) services application and reused by each REST services message received. The **processRequest** method is called on this object, passing the message details. That method will decode the URL and any objects passed in Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format, and call the required method on the same REST object. The result returned by the method is encoded into XML or JSON as requested. The **reply** method is called, passing the string to be returned to the client.

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

Inherits From: Object

Inherited By: (None)

JadeRestService Class Constants

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

Constant	Integer Value
OutputFormat_Json	0
OutputFormat_Json_NewtonSoft	2
OutputFormat_Xml	1

JadeRestService Properties

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

Property	Contains the …
httpStatusCode	HTTP status code error returned to the client
objectsToBeDeleted	Array for non-shared transient objects to be deleted when the REST method has completed

httpStatusCode

Type: Integer

The **httpStatusCode** property of the **JadeRestService** class enables you to return an HTTP status code error to the client. The property is initialized to **200** (**OK**) before the REST method is called. If the property is set to a value other than **0** or **200**, an HTML error response is generated to report the HTTP error to the client. Any data returned by the called method is also passed as the body of the message.

The application can use this status or the returned null values to indicate that the objects could not found.

objectsToBeDeleted

Type: ObjectArray

The **objectsToBeDeleted** property of the **JadeRestService** class is an array for non-shared transient objects used in a REST method. After the method returns, the array is purged. You should add transient objects to this array in your logic.

Anything added to the **objectsToBeDeleted** collection is expected to be non-shared transients, and any other object lifetime will cause the logic to fail because the logic is not in transient state.

Note If the object returned by the called REST method is a non-shared transient, it is automatically deleted even if it is not included in this array.

A returned *shared* transient object is not deleted on completion of the REST Services processing. It would be unsafe to do so, because JADE cannot be certain of whether that is the intention and JADE would have to go into transaction state to do so.

JadeRestService Methods

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

Method	Description
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
getOutputFormat	Returns an **Integer** value that represents the output format
getServerVariable	Returns the specified HTTP header information for your REST service request from IIS
isVDFilePresent	Returns **true** if the specified file is present in the virtual directory used by the **jadehttp** library
processRequest	Processes the received message
reply	Sends the returned value from the called method to the client

createVirtualDirectoryFile

Signature

```
createVirtualDirectoryFile(filename: String;
                           contents: Binary;
                           retain:   Boolean): Integer;
```

The **createVirtualDirectoryFile** method of the **JadeRestService** 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
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 from an end point method that is being executed when called from a REST Services request.

deleteVirtualDirectoryFile

Signature
```
deleteVirtualDirectoryFile(filename:          String;
                           deleteIfReadOnly: Boolean): Integer;
```

The **deleteVirtualDirectoryFile** method of the **JadeRestService** 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 from an end point method that is being executed when called from a REST Services request.

getOutputFormat

Signature
```
getOutputFormat(): Integer;
```

The **getOutputFormat** method of the **JadeRestService** class returns one of the following **Integer** values or class constants, which represent the format of the output requested by the client.

Integer Value	JadeRestService Class Constant	Output Format
0	OutputFormat_Json	JSON in Microsoft format
1	OutputFormat_Xml	XML in Microsoft format
2	OutputFormat_Json_NewtonSoft	JSON in NewtonSoft format

The value returned by this method is assigned after the **processRequest** method of the **JadeRestService** class has been called.

getServerVariable

Signature `getServerVariable(var: String): String;`

The **getServerVariable** method of the **JadeRestService** class returns the specified HTTP header information for your REST service request from the Internet Information Server (IIS). This method must be called during the processing of a REST service message; for example, from a re-implementation of the **JadeRestService** class **processRequest** method. Calling the method when a message is not being processed results in null always being returned.

As the **var** parameter is IIS-dependent, it is therefore subject to change. Refer to the **ServerVariables** function in your Internet Information Services (IIS) documentation for details.

The method in the following example returns the IP address of the REST service as determined by IIS.

```
processRequest(httpIn: String; queryStr: String; pathIn: String; methodType:
String) updating;
vars
    str : String;
begin
    str := self.getServerVariable("ALL_HTTP");
    inheritMethod(httpIn, queryStr, pathIn, methodType);
end;
```

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
SERVER_PORT	Port number to which the request was sent
URL	Base portion of the URL

The method must be called on the same node as the application. If you call the method from a server method and the application is not running on the server, a 31039 error (*Connection invalid invocation*) occurs when trying to access the TCP/IP connection, and error 1242 (*A method executing in another node was aborted*) is reported to the REST service.

isVDFilePresent

Signature `isVDFilePresent(fileName: String): Boolean;`

The **isVDFilePresent** method of the **JadeRestService** 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 from an end point method that is being executed when called from a REST Services request.

processRequest

Signature
```
processRequest(httpIn:      String;
               queryStr:    String;
               pathIn:      String;
               methodType:  String);
```

The **processRequest** method of the **JadeRestService** class is called to process the received message. Although you can reimplement the method to allow the application to pre-process the input, you must call the **JadeRestService** implementation, by using the **inheritMethod** instruction, to complete the processing.

Note REST service messages can contain only public and read-only properties that are going to be serialized. Protected properties are excluded from the serialization process.

The parameters for this method are listed in the following table.

Parameter	Description
httpIn	The string returned from a Web browser request as a result of a **PUT**, **POST**, or **DELETE** action on a Web page. When there is no such action, the string that is returned is the same as the value returned by the **queryString** parameter. This information can include an XML or JSON script used to populate an object parameter called on the required method.
queryString	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.
pathIn	The path part of the URL used to construct the method name and its primitive type parameters; for example, **customer/123**.
methodType	The type of the request; that is, **GET**, **PUT**, **PUSH**, or **DELETE**.

reply

Signature
```
reply(msg: String);
```

The **reply** method of the **JadeRestService** class is called to send the returned value from the called method. Although you can reimplement the method to allow the application to pre-process the input, you must call the **JadeRestService** implementation to complete the processing.

The **msg** parameter is the returned result that is sent to the client.

Note REST service messages can contain only public and read-only properties that are going to be serialized. Protected properties are excluded from the serialization process.

JadeRpsDataPumpIF Interface

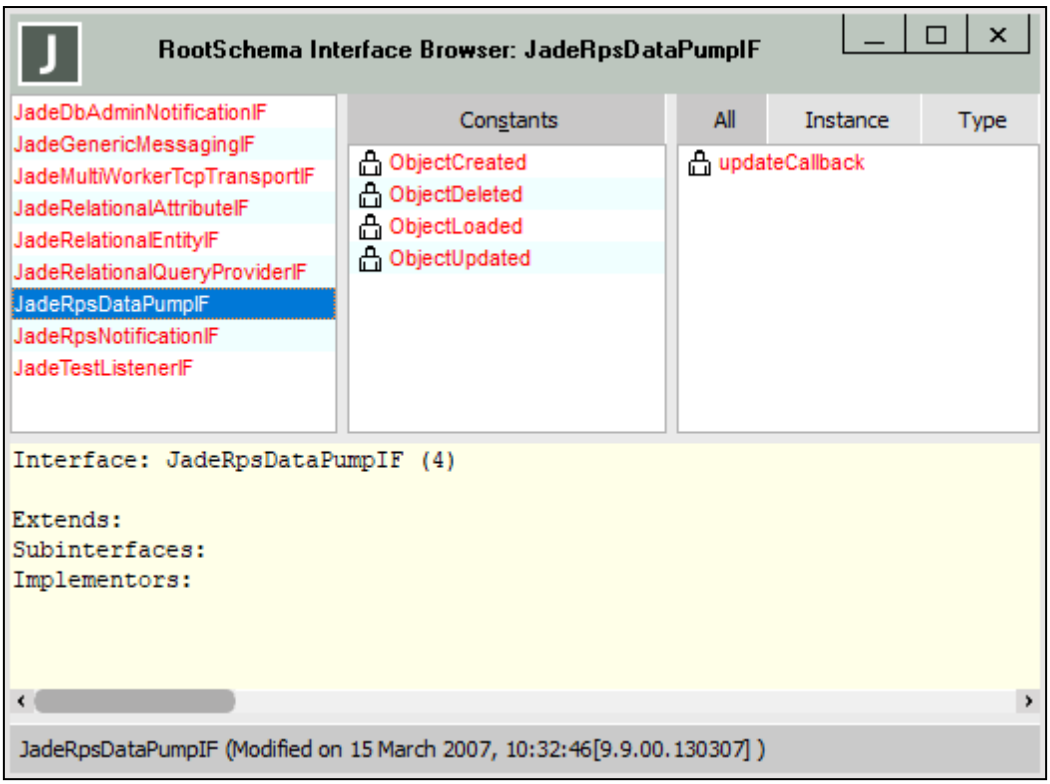
The **JadeRpsDataPumpIF** interface provides the functionality required to filter and manipulate the output sent to the target relational database from RPS.

If you have a user-defined non-GUI data pump application, you can define a class that implements the **JadeRpsDataPumpIF** interface and its **updateCallback** method. The **updateCallback** method determines whether a row is to be propagated to the relational database table and enables you to manipulate the row data.

For details, see "Implementing a User-Defined Data Pump Application", in Chapter 2 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

Note You can also filter output to the target relational database from RPS by using the **rpsSuppressTransactionDeletes** method defined in the **Process** class in transactions on the primary system. This method suppresses the replication of delete transactions in the relational database.

You can view the **JadeRpsDataPumpIF** interface and its constants and method in the Interface Browser of the **RootSchema**, as shown in the following image.



For details about implementing the **JadeRpsDataPumpIF** 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*. Note that automatically generated stub methods in classes that implement the interface contain no body logic.

For details about the **JadeRpsDataPumpIF** interface constants and method, see "JadeRpsDataPumpIF Interface Constants" and "JadeRpsDataPumpIF Interface Method Callback Signature", in the following subsections.

JadeRpsDataPumpIF Interface Constants

The constants provided by the **JadeRpsDataPumpIF** interface are listed in the following table.

Constant	Integer Value
ObjectCreated	1
ObjectDeleted	3
ObjectLoaded	4
ObjectUpdated	2

JadeRpsDataPumpIF Interface Callback Method Signature

The signature of the callback method provided by the **JadeRpsDataPumpIF** interface is summarized in the following table.

Method	Callback method for the …
updateCallback	Update of a row

updateCallback

Signature

```
updateCallback(tranID:           Integer64;
               timestamp:        TimeStamp;
               oid:              Object;
               operation:        Integer;
               tableName:        String;
               dropRowOperation: Boolean io;
               outputRowList:    JadeDynamicObject);
```

The **updateCallback** method of the **JadeRpsDataPumpIF** interface determines whether the row is replicated in the relational database table and, for an object create, update, or load operation, enables you to manipulate the data in the row.

The parameters for this method are listed in the following table.

Parameter	Description
tranID	A 64-bit integer value that uniquely identifies a transaction
timestamp	The timestamp of the journal record that describes the transaction
oid	The object being created, updated, deleted, or loaded
operation	An integer value identifying whether the object was created, updated, deleted, or loaded
tableName	The name of the table being output
dropRowOperation	A boolean value that determines whether the transaction is replicated in the relational database
outputRowList	A dynamic object with one attribute for each column in the mapped relational table

The **dropRowOperation** io parameter has the value **false** by default. If you set it to **true**, the row is not replicated in the relational database.

For create, update, and load operations, an attribute is added to the **outputRowList** dynamic object parameter for each column in the mapped relational table. The attribute names are the column names.

The attribute values are populated with default values obtained from the JADE object, including values returned by user-defined column-mapping methods. The **updateCallback** method can interrogate attributes and access or change attribute values.

For more details and an example of the use of the **updateCallback** method, see "Implementing User-Defined Output Control", Chapter 2 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

JadeSerialPort Class

The **JadeSerialPort** class encapsulates the behavior required for communicating with external systems and external applications (either JADE or non-JADE systems) through a serial port.

In JADE thin client mode, connections can be made either to the thin client or the application server.

The **JadeSerialPort** 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 successfully, 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 operation can be performed at one time.

Performing a synchronous write operation stops any additional requests from being queued until the synchronous operation is completed.

You can use serial ports with numbers greater than **9**, by using the extended name format; for example, **"\\.\COM11"**. (For details, see "Communications Resources" under the Microsoft Developer Network (MSDN) CreateFile function documentation.)

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

Inherits From: Connection

Inherited By: (None)

JadeSerialPort Class Constants

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

Constant	Integer Value	Constant	Integer Value
DataBits16	4	DataBits16x	5
DataBits5	0	DataBits6	1
DataBits7	2	DataBits8	3
FlowControl_Hardware	0	FlowControl_None	2
FlowControl_XonXoff	1	Parity_Even	2
Parity_Mark	3	Parity_None	0
Parity_Odd	1	Parity_Space	4
StopBits_1_0	0	StopBits_1_5	1
StopBits_2_0	2		

JadeSerialPort Properties

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

Property	Specifies …
dataBits	The number of data bits transmitted per character
flowControl	The method used to pause and resume transmission of data
parity	The method used to detect errors in transmission
speedBps	The transmission rate in bits per second
stopBits	The duration of the framing signal sent after transmitting each byte
usePresentationClient	Whether the connection is opened on the thin client or application server

dataBits

Type: Integer

The **dataBits** property of the **JadeSerialPort** class contains a value representing the number of data bits transmitted per character (or unit of transmission). This number does not include the parity bit. The default **DataBits8** (3) value for the **dataBits** property is typically used, as the character size matches that of a byte. **DataBits7** (2) is used for transmission of US-ANSI characters.

The values for the **dataBits** property are listed in the following table.

Constant	Integer Value
DataBits5	0
DataBits6	1
DataBits7	2
DataBits8	3
DataBits16	4
DataBits16x	5

The following code fragment shows the use of the **dataBits** property.

```
vars
    sp: JadeSerialPort;
begin
    create sp transient;
    sp.dataBits := JadeSerialPort.DataBits8;
```

flowControl

Type: Integer

The **flowControl** property of the **JadeSerialPort** class contains the action used to pause and resume transmission of data.

The values for the **flowControl** property are listed in the following table.

Constant	Integer Value	Controlled by the …
FlowControl_Hardware	0	Hardware on separate circuits
FlowControl_None	2	No data flow control
FlowControl_XonXoff	1	Receiver sending pause and resume signals

The default value for the **flowControl** property is **FlowControl_Hardware** (0).

The following code fragment shows the use of the **flowControl** property.

```
vars
    sp: JadeSerialPort;
begin
    create sp transient;
    sp.flowControl := JadeSerialPort.FlowControl_Hardware;
```

parity

Type: Integer

The **parity** property of the **JadeSerialPort** class contains the action used to detect errors in transmission

The values for the **parity** property are listed in the following table.

Constant	Integer Value	Description
Parity_None	0	No parity bit is added
Parity_Odd	1	Number of 1 bits in each character, including the parity bit, is always odd
Parity_Even	2	Number of 1 bits in each character, including the parity bit, is always even
Parity_Mark	4	Parity bit is always set to the mark signal condition (logical 1)
Parity_Space	5	Parity bit is always set to the space signal condition (logical 0)

The default value for the **parity** property is **Parity_None** (0).

The following code fragment shows the use of the **parity** property.

```
vars
    sp: JadeSerialPort;
begin
    create sp transient;
    sp.parity := JadeSerialPort. Parity_None;
```

speedBps

Type: Integer

The **speedBps** property of the **JadeSerialPort** class contains the transmission rate measured in bits per second.

The following values are valid.

- 300
- 600
- 1200
- 2400
- 4800
- 9600 (the default)
- 14400
- 19200
- 38400
- 56000
- 57600
- 115200
- 128000
- 256000

The following code fragment shows the use of the **speedBps** property.

```
vars
    sp: JadeSerialPort;
begin
    create sp transient;
    sp.speedBps := 2400;
```

stopBits

Type: Integer

The **stopBits** property of the **JadeSerialPort** class contains the duration of the framing signal that is sent after transmitting each byte. The default value is **StopBits_1_0** (0).

The values for the **stopBits** property are listed in the following table.

Constant	Integer Value	Duration of Framing Signal
StopBits_1_0	0	One bit
StopBits_1_5	1	One and a half bits
StopBits_2_0	2	Two bits

The following code fragment shows the use of the **stopBits** property.

```
vars
    sp: JadeSerialPort;
begin
```

```
    create sp transient;
    sp.stopBits := JadeSerialPort.StopBits_1_0;
```

usePresentationClient

Type: Boolean

The **usePresentationClient** property of the **JadeSerialPort** 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.

The following code fragment shows the use of the **usePresentationClient** property.

```
vars
    sp: JadeSerialPort;
begin
    create sp transient;
    sp.usePresentationClient := true;
```

JadeSerialPort Methods

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

Method	Description
close	Closes a connection to a remote application and then returns
closeAsynch	Closes a connection to a remote application and returns immediately
listen	Listens for a remote application to connect to JADE and returns when a connection is established
listenAsynch	Listens for a remote application to connect to JADE
open	Establishes a connection to a remote application and returns when established
openAsynch	Establishes a connection to a remote application and returns immediately
readBinary	Reads binary data from the connection and returns when the data 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 **JadeSerialPort** class closes a connection to a remote application or device 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 it has been left in connection state.

```
buttonUnload_click(btn: Button input) updating;
begin
    // If a connection is present, closes the connection
    if self.connection.state = Connection.Connected then
        self.connection.close;
    endif;
    self.unloadForm;
end;
```

closeAsynch

Signature
```
closeAsynch(receiver: Object;
            msg:      String);
```

The **closeAsynch** method of the **JadeSerialPort** 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. Although you can call the **closeAsynch** method while the connection is in any state, there can be no other pending asynchronous call active at the time (for example, there is no current timeout pending for a **readBinaryAsynch** or **readUntilAsynch** method).

When the **closeAsynch** method completes, the user-written callback method specified in the **msg** parameter is called.

Note On asynchronous calls, the state may not change immediately, and it can remain **Connected** (2) for a short period until JADE has rescheduled the request.

The callback method must match the signature required by the calling **closeAsynch** method, as follows.

Signature `closeCallback(connection: Connection);`

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
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
```

```
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // 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 updateCloseCalls method.
    self.connection.closeAsynch(conlog, "updateCloseCalls");
    statusLine1.caption := "Disconnected";
end;
```

listen

Signature `listen();`

The **listen** method of the **JadeSerialPort** class listens for a remote application to connect to JADE and returns when a connection is established. You can call the **listen** method only when the **state** property is **Disconnected** (0). The **state** property changes to **Connected** (2) when the **listen** method completes.

The code fragment in the following example sets the connection to listen to the current port, sets the status bar to read *connected*, and fills the text box with the name information if a connection is made.

```
self.connection.listen;
if self.connection.state = Connection.Connected then
    statusLine1.caption := "Connected";
    textBox.text        := self.connection.name;
endif;
...
```

See also the **Connection** class **timeout** property.

listenAsynch

Signature
```
listenAsynch(receiver: Object;
             msg:      String);
```

The **listenAsynch** method of the **JadeSerialPort** class listens for a remote application to connect to JADE.

When a connection is established, the object specified in the **receiver** parameter is sent the name of the callback method specified in the **msg** parameter.

You can call the **listenAsynch** method only when the **state** is **Disconnected** (0). When this method is called, the state is changed to **Connecting** (1), or listening.

Note On asynchronous calls, the state may not change immediately, and it can remain **Disconnected** (0) for a short period until JADE has rescheduled the request.

The following example shows the use of the **listenAsynch** method.

```
listenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
```

```
    // If none exists, the object is created and its properties
    // are initialized.
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // Sets the connection to listen on the current port and returns
    // immediately.  If a connection is made, the ConnectionLog object
    // referenced by conlog is called, to run the updateListenCalls method.
    self.connection.listenAsynch(conlog, "updateListenCalls");
end;
```

See also the **Connection** class **timeout** property.

When a connection is established, the user-written callback method specified in the **msg** parameter is called. The callback method must match the signature required by the calling **listenAsynch** method, as follows.

Signature `listenCallback(connection: Connection);`

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(connection: Connection) updating;
begin
    beginTransaction;
    self.numberOfListenCalls := self.numberOfListenCalls + 1;
    commitTransaction;
end;
```

open

Signature `open();`

The **open** method of the **JadeSerialPort** class establishes a connection to a remote application or device and returns when the connection is established. The **open** method can be called only when the **state** is **Disconnected** (0).

The code fragment in the following example shows the use of the **open** method.

```
if bOpen.value = true then
    self.connection.open;
elseif bListen.value = true then
    statusLine1.caption := "Listening";
    self.connection.listen;
else
    self.connection.close;
endif;
```

openAsynch

Signature `openAsynch(receiver: Object; msg: String);`

The **openAsynch** method of the **JadeSerialPort** 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 name of the callback method specified in the **msg** parameter.

The **openAsynch** method can be called only when the **state** property is **Disconnected** (0). When this method is called, the value of the **state** property is changed to **Connecting** (1).

Note On asynchronous calls, the state may not change immediately, and it can remain **Disconnected** (0) for a short period until JADE has rescheduled the request.

The following example shows the use of the **openAsynch** method.

```
buttonOpenAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, it is created and its properties initialized.
    beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
    commitTransaction;
    // 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.connection.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(connection: Connection);`

The following method is an example of a **ConnectionLog** class callback method for the **openAsynch** method, which updates the number of method invocations recorded for this method.

```
updateOpenCalls(connection: Connection) updating;
begin
    beginTransaction;
    self.numberOfOpenCalls := self.numberOfOpenCalls + 1;
    commitTransaction;
```

```
    self.connection.readBinaryAsynch(1024, connection, "readCallback");
end;
```

readBinary

Signature `readBinary(length: Integer): Binary;`

The **readBinary** method of the **JadeSerialPort** 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 **fillReadBuffer** property. This method can be called only when the value of the **state** property is **Connected** (2).

Only one synchronous or asynchronous read operation can be performed at one time on a connection. See also the **timeout** property inherited from the **Connection** class.

The following example shows the use of the **readBinary** method.

```
openButton_click(btn: Button input) updating;
vars
    pos : Integer;
    bin : Binary;
begin
    if openButton.caption = $X_Open then
        self.connection.name := connectionName.text;
        self.connection.open;
        openButton.caption   := $X_OK;
    else
        if sendIt.value then
            if loop.value then
                self.multiSend;
            else
                self.connection.writeBinary(input.text.Binary);
            endif;
        elseif receiveIt.value then
            if loop.value then
                self.multiReceive;
            else
                self.connection.fillReadBuffer := false;
                bin := self.connection.readBinary(200);
                sl1.caption := bin.String;
            endif;
        endif;
    endif;
end;
```

readBinaryAsynch

Signature
```
readBinaryAsynch(length:   Integer;
                 receiver: Object;
                 msg:      String);
```

The **readBinaryAsynch** method of the **JadeSerialPort** 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 **fillReadBuffer** property, the object specified in the **receiver** parameter is sent the name of the callback method 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 **state** property is **Connected** (2).

The following example shows the use of the **readBinaryAsynch** method.

```
receiveAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the variable conlog to reference a ConnectionLog object.  If
    // none exists, the object is created and its properties initialized.
    if self.connection.state = Connection.Connected then
        beginTransaction;
        conlog := ConnectionLog.firstInstance;
        if conlog = null then
            create conlog;
            conlog.numberOfListenCalls  := 0;
            conlog.numberOfOpenCalls    := 0;
            conlog.numberOfCloseCalls   := 0;
            conlog.numberOfBinaryReads  := 0;
            conlog.numberOfBinaryWrites := 0;
        endif;
        commitTransaction;
    // Reads binary data from the connection and returns immediately.
    // When data is read, the ConnectionLog object referenced by conlog is
    // called to run the updateBinaryReads method.  It is passed a parameter
    // containing the binary data that was read from the connection.
        self.connection.readBinaryAsynch(50, conlog, "updateBinaryReads");
    endif;
end;
```

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 callback method must match the signature required by the calling **readBinaryAsynch** method, as follows.

Signature

```
readBinaryCallback(connection: Connection;
                   buffer:     Binary);
```

The following is an example of a **ConnectionLog** class callback method for the **readBinaryAsynch** method, which updates the number of method invocations recorded for this method.

```
updateBinaryReads(connection: Connection; buffer: Binary) updating;
begin
    beginTransaction;
    self.numberOfBinaryReads := self.numberOfBinaryReads + 1;
    commitTransaction;
end;
```

See also the **timeout** property inherited from the **Connection** class.

readUntil

Signature `readUntil(delimiter: Binary; maxLength: Integer): Binary;`

The **readUntil** method of the **JadeSerialPort** 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 one character at a time and scan or handle your own data buffering.

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 **state** property inherited from the **Connection** class is **Connected** (2). See also the **timeout** property inherited from the **Connection** class.

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 **JadeSerialPort** 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 one 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 **timeout** property inherited from the **Connection** class.

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(connection: Connection;
                bin:        Binary);
```

writeBinary

Signature
```
writeBinary(buffer: Binary);
```

The **writeBinary** method of the **JadeSerialPort** 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 **state** property is **Connected** (**2**).

The following example shows the use of the **writeBinary** method.

```
openButton_click(btn: Button input) updating;
vars
    pos : Integer;
    bin : Binary;
begin
    if openButton.caption = $X_Open then
        self.connection.name := connectionName.text;
        self.connection.open;
        openButton.caption   := $X_OK;
        listenButton.caption := $X_Close;
    else
        if sendIt.value then
            if loop.value then
                self.multiSend;
            else
                self.connection.writeBinary(input.text.Binary);
            endif;
        elseif receiveIt.value then
            if loop.value then
                self.multiReceive;
            else
                self.connection.fillReadBuffer := false;
                bin := self.connection.readBinary(200);
                sl1.caption := bin.String;
            endif;
        endif;
    endif;
end;
```

See also the **timeout** property inherited from the **Connection** class.

writeBinaryAsynch

Signature
```
writeBinaryAsynch(buffer:   Binary:
                  receiver: Object;
                  msg:      String);
```

The **writeBinaryAsynch** method of the **JadeSerialPort** 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 **state** property is **Connected** (2).

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.

```
buttonSendAsynch_click(btn: Button input) updating;
vars
    conlog : ConnectionLog;
begin
    // Sets the conlog variable to reference a ConnectionLog object.
    // If none exists, it is created and its properties initialized.
    if self.connection.state = Connection.Connected then
        beginTransaction;
            conlog := ConnectionLog.firstInstance;
            if conlog = null then
                create conlog;
                conlog.numberOfListenCalls  := 0;
                conlog.numberOfOpenCalls    := 0;
                conlog.numberOfCloseCalls   := 0;
                conlog.numberOfBinaryReads  := 0;
                conlog.numberOfBinaryWrites := 0;
            endif;
        commitTransaction;
        // 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.
        self.connection.writeBinaryAsynch(textBox1.text.Binary, conlog,
                                          "updateBinaryWrites");
    endif;
end;
```

The callback method must match the signature required by the calling **writeBinaryAsynch** method, as follows.

Signature `writeBinaryCallback(connection: Connection);`

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(connection: Connection) updating;
begin
    beginTransaction;
    self.numberOfBinaryWrites := self.numberOfBinaryWrites + 1;
    commitTransaction;
end;
```

See also the **timeout** property inherited from the **Connection** class.

JadeSkin Class (superseded from JADE release 6.0)

The **JadeSkin** class contains JADE skins defined for your JADE release 5.1 and 5.2 applications and encapsulates the behavior required to define and maintain JADE skins using the **JadeSkinMaint** and **JadeSkinSelect** forms provided by the JADE RootSchema.

Note As the functionality of this class has been replaced by new skin classes and this class provides limited functionality required to define skins only for the application, it may be deimplemented in a later release.

For details about the extended functionality that enables you to define and maintain skins for your runtime applications, forms, or controls and use these only when specific criteria are met, see the **JadeSkinEntity** and **JadeSkinRoot** classes, later in this chapter, and "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the *JADE Runtime Application Guide*.

A skin is a series of images that is applied to the caption line, menu line, and border areas of each form to provide an enhanced look and feel to each form. The skin can also define images for buttons, **JadeMask** controls, check boxes, and option buttons to further enhance the look and feel forms.

In JADE, applications can set a skin that is applied to all JADE forms displayed during the running of that application in the current work session. User applications could enforce a specific skin (for example, a company logo and so on), or they could provide the user with the ability to select a preferred skin that is set by the application **initialize** method (by calling the **app.setSkin** method).

JADE provides a collection of skins for the JADE development environment and a global collection that contains any user-defined skins that are available to all schemas. As JADE must be able to upgrade existing systems by just replacing the JADE system files, the skins used by the JADE development environment cannot be updated, and access to user-defined skins is not permitted.

Note As references to skins information are contained in the **_usergui.dat** system schema file, when the **ReadOnlySchema** parameter in the JADE initialization file is set to **true**, skins cannot be loaded.

JADE provides the following facilities that are available to any developer.

- **JadeSkinMaint** form, which enables you to define and maintain skins for user applications at run time.
- **JadeSkinSelect** form, which enables a user to select the skin to use in the application at run time.
- **JadeSkin** class, in which the skins are stored.
- **Application** class **getSkin** method, which returns a reference to the JADE skin that is currently set. If no skin is currently set, a **null** value is returned.
- **Application** class **getSkinCollection** method, which returns the global collection of skins.

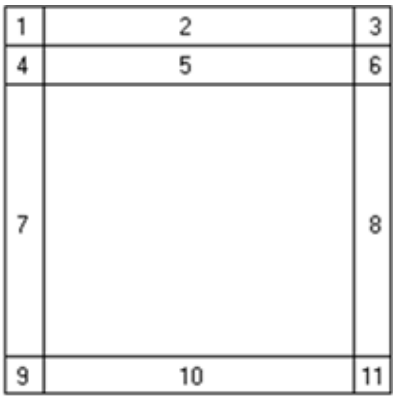
  This collection is global to all schemas and is automatically created by the first **app.getSkinCollection** call.

  To implement your own selection facility, display the **name** property for each **JadeSkin** object in the collection.
- **Application** class **setSkin** method, which defines the skin to be used by the application by setting the skin that applies to the application that is currently running.

  To cancel skin usage for the application, pass **null** as the skin object; that is, **app.setSkin(null)**.

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 in JADE 5.1 and 5.2 applications.)

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 above 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 edge of the area defined by image 3.
- MDI child form icons are placed adjacently at the top right edge of the area defined by image 6.
- Form icons that are disabled are not displayed.

Button, **CheckBox**, and **OptionButton** control images are optional. If these images are not included, the standard drawing of controls is performed. If not all images for a button are included, the required state is drawn by using the button up picture. For example, if there is no roll over picture, no rollover state is displayed.

If the skin includes button images, any **Button** or **JadeMask** control adopts the appearance defined by the skin. Any button then also behaves like a **JadeMask** control with roll over and roll under effects. (For details, see "JadeMask Class", in Chapter 2.)

The following areas are not affected by using a skin.

- Only JADE forms adopt the skin presentation. Environment-defined forms such as common dialogs and the JADE exception dialogs are unchanged.
- Form icons (for example, the **Terminate** button) do not display button down effects.
- When a form is resized, the environment may draw the standard form image while the resize is occurring.
- Windows-drawn menu items are unchanged by the skin. This includes the system menu.

- Any changes made to the skin do not affect any current users of that skin.
- Default sounds do not occur when forms are minimized and maximized, as their actions must be performed programmatically by JADE.

For details about maintaining and using JADE skins, see "Specifying Your JADE Installation Preferences", in Chapter 2 of the *JADE Installation and Configuration Guide*, and "Defining and Maintaining JADE Skins" under "Using JADE Skins in Your Runtime Applications", in Chapter 2 of the *JADE Runtime Application Guide*.

Inherits From: Object

Inherited By: (None)