Object Manager Guide

VERSION 2016.0.02

jade

Contents

Contents v

Before You Begin

The *JADE Object Manager Guide* is intended as the main source of information when administering the JADE Object Manager or writing JADE or C-level Application Programming Interface (API) calls to JADE.

Who Should Read this Guide

The main audience for the *JADE Object Manager Guide* is expected to be system administrators and JADE developers who are familiar with JADE and object-oriented concepts.

What's Included in this Guide

The *JADE Object Manager Guide* has six chapters.

Chapter 1	Covers JADE Object Manager architecture
Chapter 2	Covers JADE security, including deployed databases and the JADE development environment
Chapter 3	Gives a reference to the JADE Object Manager Application Programming Interface (API)
Chapter 4	Covers JADE system instrumentation and diagnosis, including node statistics sampling
Chapter 5	Covers the JADE Logical Certifier diagnostic tool and its errors and repairs
Chapter 6	Gives a reference to the C-level Application Programming Interface (API)

Related Documentation

Other documents that are referred to in this guide, 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 a JADE database
JADE Development Environment Administration Guide	Administering the JADE development environment
JADE Development Environment User's Guide	Using the JADE development environment
JADE Developer's Reference	Developing or maintaining JADE applications
JADE Encyclopaedia of Classes	System classes (Volumes 1 and 2), Window classes (Volume 3)
JADE Encyclopaedia of Primitive Types	Primitive types and global constants
JADE Initialization File Reference	Maintaining JADE initialization file parameter values
JADE Installation and Configuration Guide	Installing and configuring JADE

Title	Related to…
JADE Monitor User's Guide	Monitoring and examining your JADE environment
JADE Synchronized Database Service (SDS) Administration Guide	Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)

Conventions

The *JADE Object Manager Guide* 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 "JADE Object Handling" 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 JADE Object Manager

This chapter covers the following topics.

JADE Object Manager Distributed Processing

The JADE Object Manager implements object-oriented and distributed processing functionality.

The distributed processing components of the JADE Object Manager, listed in the following table, are the environmental objects maintained automatically by the JADE Object Manager.

Component	Description
system	A single object for a specific database. It contains a dictionary of the nodes that are currently signed on to the server node.
node	One node object exists for each *logical* workstation connected to the server node workstation. There is one fixed server node, and none, one, or many client nodes. A node represents a workstation that hosts the execution of one or several processes, and it contains a dictionary of the processes currently active in the node. A node object is created for each JADE executable program that is running; that is, a workstation that is running two JADE applications has two node objects, or logical workstation connections to the server.
process	One process object exists for every sign-on operation performed on a specific node. There is always a system process on every node, created automatically every time a node is initialized. This process is used for internal housekeeping of the node.

You can access these system-provided objects to assist in process distribution, by using the **jomGetExecInstances** Application Program Interface (API). For details, see "Getting Execution Instances", in Chapter 3.

The system-provided objects are created as instances of special classes: the **System**, **Node**, and **Process** classes. The instances of these classes correspond to the **system**, **node**, and **process** JADE system variables (JADE language reserved words), which are defined from the point of view of the method that uses them. For details, see "System Variables", in Chapter 1 of the *JADE Developer's Reference*.

A *node* object is created every time a new workstation starts executing a JADE Object Manager client application, as a result of a **jomInitialize** API call. (For details, see "Initializing a Node", in Chapter 3.)

A *process* object is created every time an application is started in a JADE Object Manager node, as a result of a **jomSignOn** API call. (For details, see "Opening a Process", in Chapter 3.) Every process executes in its own operating system thread, and the user is required to create a new thread before signing on again. When the JADE Object Manager application is the **jade.exe** program, methods available in the JADE language enable you to start new processes from a JADE user process.

Note Lock environmental object collections (for example, **node**.**processes** and **system**.**nodes**) with caution, as this can cause hold-ups when processes sign off and on and when nodes initiate and terminate.

No process can access uncommitted changes to a persistent object made by another process running in the same node or a different node. The changes are invisible to other processes until they are committed. Any other process making an unlocked access to an object being updated will receive the most-recently committed edition, and will not see the uncommitted updates.

Methods can be executed on a client node or a server node. Modifications to persistent objects must be done within persistent transactions. (For more details about server and client execution of methods, see "Method Options", in Chapter 1 of the *JADE Developer's Reference*.)

The following restrictions apply to persistent objects when using the server or client execution method option.

- Transactions must be total client transactions or total server transactions; that is, any **beginTransaction** and **commitTransaction** pair of instructions must be done while executing on the client (without executing an updating server method) or while executing on the server (without the execution of updating client methods). This restriction also applies to the **beginLoad** and **endLoad** and the **beginLock** and **endLock** instruction pairs.
- JADE enforces the starting, performing, and finalizing of persistent transactions on a single node (that is, a client or a server node). All of the update operations for the transaction must occur in the same node that started the transaction.

  JADE does not attempt to synchronize the client cache and server cache when accessed by the same process. Server methods should avoid accessing persistent objects that the process is updating on the client node, as changes made to persistent objects in a non-committed transaction are not usually visible to server methods executed during that interval. The same restriction applies to **clientExecution** methods called from server methods.

  Note Attempting such an access can cause an exception to be raised (that is, *1276 – Potential cache inconsistency*).

All changes that occur when transient objects are created, updated, or deleted during the execution of a server method are sent back to the client node. These changes are then merged with the current transient object population in the client node. The transient object space on both the client and server nodes is then the same as if the method had been executed locally.

As a transaction is effective across client and server nodes for shared transient objects, the **beginTransientTransaction** instruction makes transaction state active for client methods and server methods, regardless of the execution location. Similarly, the **commitTransientTransaction** instruction ends transaction state for client methods and server methods.

As is the case for persistent objects, no process can access updated shared transient objects until the **commitTransientTransaction** instruction is executed. A process attempting such an access without locking will receive the most-recently committed edition instead.

If the **commitTransientTransaction** instruction is executed by a server method, the updated objects are sent to the client node before they are removed from the server node cache. This ensures that other processes accessing the objects on the client node will see the latest edition.

JADE Object Handling

The JADE distributed processing architecture has a single semantic model for all objects and two basic lifetime variations: persistent and transient lifetimes.

Persistent objects, which are permanently stored in the persistent database, live across JADE executions and are shared by all processes in all nodes of the system. The server node enforces concurrency control for persistent objects. Update operations on persistent objects must be performed within transaction state; that is, inside a **beginTransaction** and **commitTransaction** pair of instructions.

Changes made to objects within a transaction state are hidden from other processes until committed. This means that other processes accessing these objects will not see the uncommitted updates, but will instead view the most-recently committed editions.

When the **prohibited** parameter in the **Process** class **prohibitPersistentUpdates** method is set to **true**, the following operations are prohibited in the current process.

- Executing the **beginTransaction** and **commitTransaction** instructions
- Updating, creating or deleting a persistent object
- Cloning or copying to a persistent object

Note The **prohibitPersistentUpdates** method is designed to prevent unexpected updates to persistent objects. It is not intended to be a comprehensive security measure.

Non-shared transient objects are local to the process that created them; that is, they cannot be accessed by any other process. Because of this, no concurrency control operations are performed when they are updated, which gives optimal performance. These transient objects are automatically removed when the process that created them finishes.

If the transient cache of a node overflows, non-shared transient objects are written out to a .**tmp** file. Each transient database file is in the following format and is located in the path specified by the **TransientDbPath** parameter in the [JadeClient] section of the JADE initialization file.

```
tdb_<host-name>_<pid><designator>.tmp
```

In this format, the *<pid>* value is the operating system process identifier and the *<designator>* value is one of the following.

- [shared], for a node's shared transient file
- [kbp_*nnnnn*], for transient files associated with a kernel background process
- [*process-oid*], for transient files associated with a user process

If a .**tmp** file is found to be growing, you can identify the application process to which it belongs by locating the process identifier in the **jommsg.log**; for example:

```
2008/02/22 14:23:26.897 000b8-0af4 Jom: Local process sign-on: oid=[187.2],
process=0x58f7fe8, no=2, id=2804, type=1 (non-prodn), scm=JadeSchema, app=Jade,
2016.0.0
```

In this log entry example, **[187.2]** is the oid for the process object.

The temporary (.**tmp**) file for each process is deleted when the process terminates. Certain kinds of failure (for example, a node crash) can result in these temporary files not being deleted. You can delete any temporary transient object overflow file that has been left behind, if required.

Shared transient objects are transient objects that are shared by processes in the same node and exist for the lifetime of the node of the process that created them. In addition, concurrency control is locally enforced by the node on which they reside.

Update operations on shared transient objects must be performed within transient transaction state; that is, inside a **beginTransientTransaction** and **commitTransientTransaction** pair of instructions.

Shared transient objects and non-shared transient objects reside in the transient cache. In addition, shared transient objects are backed up in a **.tmp** file with a unique hexadecimal number **1**. The shared transient temporary file is deleted when the node terminates.

Shared transient objects were designed to allow you to share objects among the processes of a node at a relatively low cost. Shared transient objects avoid the overhead of performing a persistent transaction. Transient transactions are lightweight transactions that do not involve auditing or communications with the server node.

Updates to shared transient objects are applied when the **commitTransientTransaction** instruction is executed. As shared transient objects are exclusively locked while being updated, other processes cannot lock these objects until the **commitTransientTransaction** instruction is executed. Other processes attempting to access these objects without locking will not see the uncommitted updates but instead will view the most-recently committed editions.

No transaction is necessary when creating, deleting, or updating ordinary non-shared transient objects. These objects can be updated at any time.

Transient transactions provide the means to group updates to shared transient objects. All shared transient objects to be updated must be exclusively locked, either manually by the user or automatically by JADE, and they remain locked until the transaction is finished. Other processes trying to get a consistent view of any of those objects by using shared locks or to update them in another transaction will be put into wait state until the completion of the current transient transaction. This guarantees a correct sequence of operations on shared transient or persistent objects, as every JADE process executes independently in its own operating system thread.

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.

Relationship between Shared Transient and Non-Shared Objects

Attempts to reference a non-shared transient object from a shared transient object raise exception 1289 (*Shared transient objects cannot reference non-shared transient objects*). This can occur as a result of explicit update or as a result of an automatic update. In such a case, you should:

- Avoid establishing a relationship between the non-shared and shared transient objects.
- Allow a reference to a shared transient object from a non-shared transient object without its inverse being maintained, by calling the **Process** class **allowTransientToSharedTranInvs** method with the **allow** parameter set to **true**.

Non-Shared Transient Objects

Non-shared transient objects are specific to each process, and they can be used only by the process that created them.

No process can access non-shared transient objects created by another process.

Cache Coherency

Note When the JADE development environment is run in multiuser mode, it requires cache coherency to be enabled.

If you start the JADE development environment in multiuser mode with cache coherency disabled, a message box is displayed, warning you that the system is not using cache coherency. If you choose to proceed (by clicking the **Yes** button), a message is logged in the **jommsg*n*.log** file, indicating that you are using the JADE development environment without cache coherency.

In a multiuser JADE system, a persistent object cached in a node can be made obsolete when a process on another node updates that object. JADE provides a default mechanism that ensures that persistent objects in the object cache of a client node are always current. Automatic client cache coherency is enabled on all nodes when the **AutomaticCacheCoherencyDefault** parameter in the [JadeServer] section of the JADE initialization file used by the server node is set to the default value of **true**.

With automatic cache coherency, an object updated on another node is automatically reloaded in cache, even when it is the receiver of a method currently being executed.

The server maintains information concerning the objects that are held in the persistent object cache for each client node that has enabled automatic cache coherency. When a transaction is committed, the server sends a message to each client, identifying the objects that have been updated. On receipt of this message, the client marks all updated objects that are present in the cache as being *obsolete*. The next time a process on the client attempts to access the obsolete object, the updated copy is retrieved from the server.

Automatic cache coherency has been designed for applications that frequently access highly volatile data. Considerations have been made to minimize the associated overhead, especially with respect to network traffic.

Alternatively, your applications can implement a customized caching strategy. Common strategies that are often used include a combination of object locks, class notifications, object notifications, object edition checking, and calls to the **Object** class **resynchObject** method.

You can enable or disable automatic cache coherency on a specific node, regardless of the default setting on the server node, by setting the **AutomaticCacheCoherency** parameter in the [JadeClient] section of the JADE initialization file used by the node to **true** or **false**.

Notes Cache coherency has no implications in locking strategies except when locks are used for cache coherence-only purposes. All locks used for update and concurrency control purposes are still necessary.

An out-of-date object can still be accessed, even when cache coherency is enabled, if the object is retrieved from the local cache while it us being updated by another node. You can guarantee that an object is not being updated by another process only by locking it.

When automatic cache coherency is enabled, calling the **Object** class **resynch** or **resynchObject** method has no effect.

Managing Transient Cache Space

All transient objects, whether shared or non-shared, are stored in the transient cache.

The shared transient objects in cache are accessible by all JADE processes. However, the non-shared transient objects can be accessed only by the processes that created them.

- When a process needs to place an object in the transient cache and the cache is full, objects must be swapped out of the cache to make room.

If the process currently has the most amount of cache allocated to it, it swaps out its own objects. If not, one of the other processes is requested to swap out its objects. Because there may be a delay before this process swaps out its objects, the cache is temporarily expanded to allow the new object to be placed into it.

The process that has been requested to swap out objects does so when it next attempts to allocate cache space or it has been detected as idle (that is, there have been no cache allocations for between 10 and 20 seconds). When temporarily expanded, the maximum that the cache is allowed to exceed the specified size is by 50 percent. If the excess would be more than that, exception 1018 (*No memory for buffers*) is raised.

- Inactive processes are activated by sending them a specific Windows message.

  This has an important consequence for users who have external C++ threads using the JADE Object Manager or C-level Application Programming Interface (API). If these threads have objects in the transient cache and are likely to be idle for extended periods, it is important that the thread logic includes a Windows message loop so that it can respond to the message sent by the kernel cache housekeeping thread.

  The message type is **WM_THREAD_CALLBACK**. The first parameter is the handle of the method to be called and the second is a parameter to be passed to the method. (Note that external C++ threads already require a Windows message loop if they are to correctly respond to JADE notifications and timer events.)

  Without a Windows message loop, the transient cache may remain in excess of its specified maximum memory size until the thread next attempts to insert a transient object into the cache.

Cache Sizes

Default cache size specifications for JADE systems are as follows.

- The default size for cache in the:
  - **TransientCacheSizeLimit** and **RemoteTransientCacheSizeLimit** parameters in the [JadeClient] and [JadeServer] sections of the JADE initialization file in client and server nodes is **40**M bytes.
  - **ObjectCacheSizeLimit** parameter in the [JadeClient] and [JadeServer] sections of the JADE initialization file in client and server nodes is **80**M bytes.

  If one of these parameters specifies a cache size beyond that allowed (that is, a percentage of the physical memory size), the parameter value is changed to reflect the maximum size allowed.
- The minimum size for all object, transient, and remote transient caches on client and server nodes is **3**M bytes.

Note The cache size on 32-bit systems cannot exceed 4G bytes.

Chapter 2 JADE Security

This chapter covers the following topics.

Overview

The JADE Object Manager provides independent forms of security hooks to enable you to incorporate your own security mechanisms into a JADE application. Security hooks are primarily of relevance in environments that allow network access to a JADE server. The security protocols can be implemented by non–JADE clients.

The security hooks provided by the JADE Object Manager are:

- Connection authentication
- Schema source encryption
- Network Remote Procedure Call (RPC) message encryption
- User validation

Each type of security hook has a different purpose and can be configured independently in combination with any other type.

Connection authentication and user validation hooks enable you to install mechanisms to ensure that external agents (either software or human) connecting to a JADE server are both genuine and authorized.

Encryption hooks enable you to incorporate data encryption algorithms of your choice, to make it difficult for anyone monitoring data on the network to observe confidential information (such as credit card numbers) that would otherwise be visible in plain-text form in packets transmitted across a local or wide-area network.

The JADE Object Manager security facilities are *server driven*; that is, if they are enabled on a server-capable node, client nodes attaching to the server must conform to the security requirements. The environment hosting a JADE server node is assumed to be physically secured from unauthorized tampering, either directly by physical access or indirectly by remote access.

JADE Object Manager security is designed so that it cannot be defeated by disabling or removing security support at the client-side alone, as doing so only prevents the client from being able to connect to a secured JADE server. The JADE Object Manager is responsible for conditionally invoking user hooks only if the hooks are enabled and are correctly defined.

The JADE security libraries provide a:

- Default implementation of the connection authentication hooks
- Dummy ("do nothing") version of the encryption hooks

By default, the JADE implementation of authentication hooks and the dummy encryption hooks are installed and configured with your JADE software, and configuration for all security hooks is *enabled*. To turn security hooks off, you must manually disable them.

Caution If character data is passed in or out, your routines must be aware of the ANSI or Unicode character size (that is, 1 byte for ANSI, 2 bytes for Unicode).

Connection Authentication

When a client node establishes a connection to a server node, the server challenges the correspondent to prove that it is genuine, by using a challenge/response protocol.

If the correspondent fails the challenge, JADE assumes that it is an impostor and abruptly drops the connection.

End-User Validation

When each user signs on to a JADE node, the user code and password are passed to optional validation methods defined in the **Global** class in the application subschema.

Your user validation methods can allow or disallow the user sign-on. If the user sign-on is disallowed, the application and process are terminated, and further JADE Object Manager requests (other than sign-on or finalize) are rejected until a successful sign-on is completed (that is, multiple retries are permitted).

Network Message Encryption

RPC (request or response) messages sent across the network are passed to a user-supplied encryption routine at the originator end point and a decryption routine on the receiver end point.

The encryption routine can encrypt the data stream in any way (and can change the length before transmission). The decryption routine must ensure that it restores the data stream to its initial value before processing.

Security for Runtime Class Instance Inspection

A security filter method provided by the JADE Root Schema **Object** class enables you to define security to specify users who can inspect class instances in a deployed (run time) database. For more details, see "Deployed JADE Database Inspection Security", later in this chapter.

Security in the JADE Development Environment

You can provide an entry point library that implements security hooks to enable you to control the access to tasks or entities in the JADE development environment.

For details, see "JADE Development Environment Security", later in this chapter.

Security for the JADE Smart Thin Client

You can provide optional encryption support for the presentation client and application server communication, as well as application execution control and Secure Sockets Layer (SSL) security.

For details, see "JADE Smart Thin Client Security", later in this chapter.

Encrypting Schema Source Files

The encryption hooks provided by the schema extract and load facilities enable you to incorporate data encryption algorithms of your choice in your schema extract files. These hooks encrypt the JADE method source code in your schema extract files, enabling you to release schema extract files without making their source code easily visible.

For more details, see "Schema Source File Security", later in this chapter.

Security for the JADE Report Writer

The JADE Report Writer schema provides a superclass for all of your **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.

For details, see "JADE Report Writer Security", later in this chapter.

Security when Accessing JADE from the Internet

When accessing your JADE systems from the Internet, you can implement security by using Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008 security and the Internet Information Server (IIS) or Apache HyperText Transfer Protocol (HTTP) Server Web server security for data access and the Secure Sockets Layer (SSL) for data transmission.

Notes These security facilities are provided by Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008 and the Web server; they are not specific to JADE.

As unknown XML Web service consumers can access an XML Web service provider, the Web server must provide basic security services at the protocol level.

For more details, see "Internet Access Support", later in this chapter. (See also "Secure Sockets Layer (SSL) Security" under "JADE Smart Thin Client Security", later in this chapter.)

Security for the JADE Monitor

If JADE development security is enabled, the JADE Monitor sets up menus via calls to the **jadeDevelopmentFunctionSelected** user-supplied hook function to provide accessibility to JADE Monitor activities.

For details, see "JADE Monitor Security", later in this chapter.

Connection Authentication Support

Connection authentication performs the following actions.

1. Calls a user-authentication routine, which issues the challenge in the form of a generated key
2. Sends the generated key to the prospective "client"
3. Calls a user-response at the client, which must reply with the correct response
4. Calls a user hook again when the response data is returned to the server-side, which can either allow or disallow the connection, based on the response

Authentication is enabled by default, using the JADE-supplied default library and authentication hooks. If you want to avoid enforcement of the authentication protocol, you must manually disable authentication in the JADE initialization file at the server.

When connection authentication is enabled:

- JADE looks in the [JadeSecurity] section of the JADE initialization file for the name of a user-defined authentication library (DLL).

  For details, see "JADE Security Section [JadeSecurity]", in the *JADE Initialization File Reference*.

- The JADE RPC module dynamically loads the authentication DLL by a **LoadLibrary** call during initialization and attempts to obtain the address of predefined functions by name, using **GetProcAddress**.

- On a remote access-capable node, if the dynamic load fails, an exception is logged in the JADE event log and remote access provider support is disabled, preventing any clients from connecting.

JADE invokes the Authentication Challenge hook, Authentication Response hook, and Authentication Verification hook described in the following subsections only if the authentication feature is enabled. Your site-supplied library must export all three hook functions to enable correct operation of authentication support.

Authentication Challenge Hook

The **secGenerateChallenge** Authentication Challenge hook is a user-supplied function that is called when authentication is enabled on a server-capable node. The **secGenerateChallenge** function has the following format.

```
secGenerateChallenge(DskParam *pChallenge);
```

The JADE RPC module is responsible for calling the authentication challenge hook every time a connection is established. This challenge is sent to the client connection in the form of a special RPC server-to-client callback request. The RPC module saves the challenge as part of connection administration, to save your user library from having to retain each connection state and to provide increased security. The connection state is set to "waiting for authentication response". If this authentication response is not received within a finite period or a message other than an authentication response is received, the connection is dropped immediately.

Your user routine should generate a random challenge in the form of binary data, which must fit into the data part of the **DskParam** (approximately 512 bytes).

As an optimization only, if a JADE client-side connection receives a challenge request and authentication is not enabled at the client, the client RPC fails (and drops) the connection and raises an exception.

pChallenge

Use the **pChallenge** output parameter for a randomly generated challenge.

Authentication Response Hook

The **secGenerateResponse** Authentication Response hook is a user-supplied function called by the RPC module in JADE client nodes on receipt of an authentication challenge request. If authentication is not enabled at the client node, the connection attempt is rejected and an exception is raised.

The **secGenerateResponse** function has the following format.

```
int JOMAPI secGenerateResponse(const DskParam *pChallenge,
                               DskParam       *pResponse);
```

When the client-side RPC receives a challenge request and authentication is enabled, the Authentication Response function is invoked, passing the challenge parameter sent in the challenge request message.

On successful completion of the call, the **pResponse** output parameter is sent back to the waiting server-side connection.

Using the **pChallenge** input parameter, your user routine must generate and return the correct response in the **pResponse** output parameter. This requires the routine to implement an algorithm compatible with the algorithm the server-side uses to authenticate the response.

If the routine returns a non-zero result, the connection attempt is rejected, the connection is dropped immediately, and an exception is raised.

The **secGenerateResponse** Authentication Response hook returns the values listed in the following table.

Value	Description
0	The response was successful
Non-zero	The response failed

pChallenge

Use the **pChallenge** input parameter for the challenge received from the server-side of the connection.

pResponse

Use the **pResponse** output parameter for the response generated from the challenge.

Authentication Verification Hook

The **secAuthenticateResponse** Authentication Verification hook is a user-supplied function that is called by the server-side RPC module on receipt of an authentication challenge response for connections waiting for such a response.

The **secAuthenticateResponse** function has the following format.

```
int JOMAPI secAuthenticateResponse(const DskParam *pChallenge,
                                   const DskParam *pResponse);
```

The JADE RPC module is responsible for calling the Authentication Verification hook for connections waiting for a challenge response when the response is received. The hook is passed both the original challenge for each connection and the received response parameter. If the verification hook returns a non-zero result, the connection is dropped immediately.

Using the input parameters, your user routine must verify that the response is the correct one for the original challenge. If the response is invalid, the routine should return a non-zero result.

The **secAuthenticateResponse** Authentication Response hook returns the values listed in the following table.

Value	Description
0	The response was successful
Non-zero	The response failed

pChallenge

The **pChallenge** input parameter is the challenge received from the server-side of the connection.

pResponse

The **pResponse** input parameter is the response returned from the client-side of the connection.

User-Validation Support

The **Global** system class provides the **getAndValidateUser** and **isUserValid** methods. You can reimplement these methods in the **Global** class of your subschema to selectively apply user validation for applications defined in that subschema. (For more details, see "Global Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.)

The **getAndValidateUser** and **isUserValid** methods of the **Global** class allow for a separation between obtaining user code and password information that often requires a dialog interaction with the end-user at the client and secondary validation, which may need to be executed at the server.

Tip Use secondary (server-side) user validation to limit the possibility of illegal system access. The **getAndValidateUser** method provides an extra level of protection at the start of a process or application in conjunction with the **isUserValid** secondary validate method run at the server.

If you do not implement the **isUserValid** method (which in turn depends on the **getAndValidateUser** method), a hacker can bypass your application security entirely by substituting his or her own client program and then accessing your schema directly. Without this level of security, you would be unable to stop this illegal access of your system.

When no user code is supplied in the **jomSignOn** API call, the JADE Object Manager invokes the **Global** class **getAndValidateUser** method. The **isUserValid** method secondary validation is always invoked. In practice, this means that for:

- JADE applications, both validation methods are invoked.
- Non–JADE applications that do not supply a user code in the **jomSignOn** API call, both validation methods are invoked. Higher-level interfaces (for example, ODBC) require that a user code is supplied to establish a database connection, which is in turn passed to the **jomSignOn** API call, so that secondary validation only is invoked.

When developing the **getAndValidateUser** and **isUserValid** methods, consider that these methods will be called in non-GUI applications. Creating and attempting to show forms in non-GUI applications will raise an exception.

Tip In the **Global** class **getAndValidateUser** method, check if the type of application is non-GUI or Web-enabled non-GUI. If so, manufacture your own user code and password; for example, you could set the user code to the application name and the password to the current schema name. Your **isUserValid** method can then check that the combination of non-GUI, user code, and password are valid, to protect against running non-GUI applications that are defined in the schema and are not intended to be run in production.

The user-validate methods have normal access to the database and can create and manipulate forms. The validate methods provide update capability so that you can retain state information, if required; for example, the number of log-on attempts for each user or a security audit trail in persistent objects.

The default implementation of the user-validate methods performs the following actions.

- Manufactures a user code
- Allows applications to be run without any user validation

The user-validate methods are invoked from a tentative process created by the JADE Object Manager as a result of the **jomSignOn** Application Program Interface (API) call. (For more details, see "Opening a Process", in Chapter 3.)

If user-validation succeeds, the **jomSignOn** API returns a valid process handle to the caller, which allows the application to proceed as usual. If the user-validate methods signal failure, the JADE Object Manager destroys the tentative application and process objects, and a **null** process handle is returned to the caller.

Network Message Encryption

In the Remote Procedure Call (RPC) context, *client-side* refers to the node initiating a request and *server-side* refers to the node processing a request. Each node can take on either client or server roles, and a node processing a request received from server-side RPC can invoke client-side RPC for a callback to the initiating client.

Although the network message encryption is enabled by default, the default implementation supplied by JADE is a dummy "do nothing" procedure. To avoid the server-side enforcement of encryption, you must manually disable encryption in the [JadeSecurity] section of your JADE initialization file.

When network message encryption is enabled, the following actions are performed.

- JADE looks in the [JadeSecurity], [JadeAppServer], or [JadeThinClient] section of the JADE initialization file for the name of a user-defined encryption library (DLL). For details, see "JADE Security Section [JadeSecurity]" or "Application Server Section [JadeAppServer]" and "JADE Presentation Client Section [JadeThinClient]" under "JADE Thin Client Sections", in the *JADE Initialization File Reference*.
- The JADE RPC modules attempt to dynamically load the encryption DLL by a **LoadLibrary** call during initialization and use the **GetProcAddress** call to attempt to obtain the address of predefined functions by

name.

- On a JADE client-only node, if the **LoadLibrary** call or any of the **GetProcAddress** calls fails, the server connections are refused and an exception is raised.
- On a remote access-capable node, if the dynamic load fails, an exception is logged in the JADE event log and remote access provider support is disabled, preventing any client from connecting.

When RPC message encryption is enabled on a node, RPC messages sent across the network are passed to a user-supplied encryption routine on the client-side and a decryption routine on the server-side. The encryption routine can encrypt the data stream in any fashion and can change the length before transmission.

The server-side decryption routine must ensure that it restores the data stream to its initial pre-encrypted value. If one peer correspondent in an RPC connection is employing encryption and the other is not, this is detected by the protocol, and offending messages are rejected with the appropriate exception raised.

When encryption is enabled, the JADE RPC generates a 32-bit integer Cyclic Redundancy Check (CRC) value and stores this in the RPC header.

The CRC is used to provide a level of protection against invalid encryption or decryption processing that fails to restore the original message intact and prevents spurious messages being passed upwards to higher-level APIs. Only the data part of the RPC message is passed to the user encryption routine; the header remains non-encrypted.

The encryption and decryption hooks are invoked by JADE only if encryption support is enabled. Your library must export both encryption and decryption functions to enable correct operation of encryption support. Encrypted RPC messages cannot be longer that 50,000 bytes following encryption.

Encryption Hook

The **rpcEncryptMessage** user-supplied Encryption hook function is called when RPC encryption is enabled on a node. The **rpcEncryptMessage** function has the following format.

```
int JOMAPI rpcEncryptMessage(const void *inData,
                             unsigned   inLength,
                             void       **outData,
                             unsigned   *pOutLength,
                             int        *pUserData);
```

The RPC encryption hook is called by the JADE client-side RPC for request messages and the server-side RPC for response messages before the message is handed to the network transport layer for transmission.

Before calling the user encryption routine, a CRC is generated and inserted into the RPC header. A flag is also set in the header, indicating that the message is encrypted. The optional **pUserData** parameter value is also sent in the header and passed to the decryption routine on the other side. You can use the **pUserData** parameter to identify the version or type of algorithm being used.

The **rpcEncryptMessage** function returns the following values.

Value	Description
0	The encryption routine was successful
Non-zero	The encryption routine failed

inData

The **inData** input parameter contains a reference to the input data stream for encryption.

inLength

The **inLength** input parameter specifies the length of the input data stream.

outData

Use the **outData** output parameter to specify the location of the output data stream after encryption.

If your encryption algorithm is capable of encrypting the data stream in place, you can set the value of the **outData** parameter to the value of the **inData** parameter. If not, the **outData** parameter should refer to the buffer that is allocated by your routine.

As the buffer is not de-allocated by JADE, it should be a static buffer or a dynamically allocated buffer that is referenced by a static or global pointer.

pOutLength

Use the **pOutLength** output parameter to specify the length of the encrypted data stream.

pUserData

Use the **pUserData** output parameter if you want to specify the user value that is passed to the decryption routine.

Decryption Hook

The RPC protocols *always* check the encrypted message flag in the header against the enabled or disabled state of the encryption feature. If a message is received that does not match the expected state, the message is rejected and the appropriate exception is raised.

The **rpcDecryptMessage** user-supplied Decryption hook function is called when RPC encryption is enabled on a node. The **rpcDecryptMessage** function has the following format.

```
int JOMAPI rpcDecryptMessage(const void    *inData,
                             unsigned      inLength,
                             void          **outData,
                             unsigned      *pOutLength,
                             int           userData);
```

The decryption hook is called by the JADE client or server-side RPC module after obtaining a message from the network layer.

After calling your decryption routine, a CRC is generated against the data part of the message and compared to the CRC in the header. A CRC mismatch results in the message being rejected and an error raised. The **rpcDecryptMessage** function returns the following values.

Value	Description
0	The decryption routine was successful
Non-zero	The decryption routine failed

inData

The **inData** input parameter contains a reference to the input data stream for decryption.

inLength

The **inLength** input parameter specifies the length of the input data stream.

outData

Use the **outData** output parameter to specify the location of the output data stream after decryption. If your decryption algorithm is capable of decrypting the data stream in place, you can set the value of the **outData** parameter to the value of the **inData** parameter. If not, the **outData** parameter should refer to the buffer that is allocated by your routine.

As the buffer is not de-allocated by JADE, it should be a static buffer or a dynamically allocated buffer that is referenced by a static or global pointer.

pOutLength

Use the **pOutLength** output parameter to specify the length of the decrypted data stream.

userData

The **userData** input parameter contains the user value that was returned from the encryption routine.

JADE Development Environment Security

The security hooks provided by the JADE development environment enable you to install mechanisms to restrict access to tasks in the JADE development environment.

JADE development environment authentication performs the following actions.

1. Calls a user authentication routine, which validates the user identifier and password for access to the JADE development environment.
2. Calls a user hook each time a specific development task is attempted, to verify that the function is available to the user.

When a user signs on to the JADE development environment, the following actions are performed.

- JADE looks in the **DevelopmentSecurityLibrary** parameter in the [JadeSecurity] section of the JADE initialization file for the name of your user-defined security library (DLL). For details, see "JADE Security Section [JadeSecurity]", in the *JADE Initialization File Reference*.
- The compiler attempts to load the security library by calling **LoadLibrary** and gets the address of your predefined **jadeDevelopmentUserInfo** routine by calling **GetProcAddress**. An error is raised if the library cannot be located in the JADE executable (binary) directory or in the current path.
- When a user accesses a task (for example, a menu command), the action is passed to your user-supplied **jadeDevelopmentFunctionSelected** routine. An error is raised if the user does not have access to that task.

When you specify your user-defined library containing your entry points in the **DevelopmentSecurityLibrary** parameter in the [JadeSecurity] section of the JADE initialization file but JADE cannot find the library or its entry points, JADE development environment access security is enabled. Any developer who does not have this library or who does not have access to this library is then prevented from using the JADE development environment.

For details about the patch control security hook, see "Patch Control Hook", later in this section.

User Information Security Hook

The **jadeDevelopmentUserInfo** user-supplied security hook function is called when a user first signs on to the JADE development environment and the **DevelopmentSecurityLibrary** parameter is located in the [JadeSecurity] section of the JADE initialization file. The user name and the password that are specified in the JADE log-on form are passed to this function.

The **jadeDevelopmentUserInfo** function has the following format.

```
int JOMAPI jadeDevelopmentUserInfo(const Character *userName,
                                   const Character *password);
```

The **jadeDevelopmentUserInfo** function returns the following values.

Value	Description
0	The user has access
Non-zero	The user has no access, and the routine failed

userName

The **userName** input parameter specifies the user name of the developer specified in the **User Id** text box in the JADE log-on form.

password

The **password** input parameter specifies the optional password of the developer specified in the **Password** text box in the JADE log-on form.

Patch Control Hook

JADE development provides a **jadeDevelopmentPatchControl** hook to the development security module, allowing:

- Patch number conflicts (where a patch is being applied to an entity that currently has an open patch number) to be resolved by an external system
- Renaming of properties, methods, classes, and constants and the saving of forms
- Changes to Relational Population Service (RPS) class maps

Patch Number Conflicts

When a user attempts to change an entity that has been modified by a currently open patch number that has been assigned to another developer, the security module is called with the following information.

The **jadeDevelopmentPatchControl** function has the following format.

```
int JOMAPI jadeDevelopmentPatchControl(const *userName,
                                       const *patchNumber,
                                       const *entityName,
                                       const *entityType,
                                       const *operation);
```

The patch control entry point values are:

- Name of the user attempting the change.
- Patch details, which has the following format.

  ```
  <entity-patch-number>:<user-patch-number>:<status>
  ```

  The *entity-patch-number* value is the current patch number of the entity being modified and is zero (**0**) if it is new or unset.

  The *user-patch-number* value is the patch number that is currently assigned to the user. During a schema load, the *user-patch-number* value is the number in the schema file.

  The *status* value is **N** if the entity is being modified for the first time against the patch number assigned to the user, **O** if the entity has been modified previously against the patch number assigned to the user and the patch number is still open, or **C** if the entity has been modified previously against the patch number assigned to the user and the patch number is closed.
- Fully qualified name of the schema entity, in the following format.

  ```
  schema-name::locale-name::string-name
  ```

  For an RPS mapping, the fully qualified name of the of the external RPS class map (**ExternalRPSClassMap**) entity, in the following format.

  ```
  schema-name::rps-mapping-name::class-name::table-names
  ```

  The *table-names* value can be a comma-separated list of tables, if multiple tables are mapped using the same external RPS class map.
- Class of the object being modified.
- Type of operation; that is, add (**A**), update (**U**), delete (**D**), move (**M**), or reorganize when the schema gets versioned (**R**).

If the **JadePatchControlSecurity** parameter in the [JadeSecurity] section of the JADE initialization file is set to **true**, the **jadeDevelopmentPatchControl** entry point must be implemented in the security module of the user or the user is prevented from updating the entity. A return value of zero (**0**) indicates that the user can proceed with the update and a non-zero value indicates that the user cannot update this entity.

When the update is not allowed, the JADE development environment informs the user of this. If this occurs during a schema load, the load is cancelled.

Note This hook is called for all methods that need to be recompiled as result of a rename action (for example, the renaming of a method). The renaming is rejected if the update is not allowed for any of these methods. If the update is allowed, the method or methods are assigned the patch number of the current user, although an extraction of the previous patch number still extracts the method or methods.

Other Calls to the Patch Control Hook

JADE provides patch control hooks for the renaming of properties, methods, classes, and constants and the saving of forms, with the following format.

```
old-name new-name literal patch-details
```

The *literal* value for renaming properties, methods, classes, and constants has the following format.

```
Type=[P] [CL] [M] [C]
```

In this format, **P** indicates a property, **CL** a class, **M** a method, and **C** a constant. The specified type is followed by **RBC**, indicating renaming before a commit, or **RAC**, indicating renaming after a commit. A rename action makes two calls to the hook: one before the transaction and one after it. For example, to rename a class, the hook is called with the **CLRBC** and the **CLRAC** literal values.

The *literal* value for forms has the following format.

```
Type=[SF] [SFA]
```

In this format, **SF** indicates the saving of a form and **SFA** indicates the saving of a form as another name. The specified type is followed by **BC**, indicating saving before a commit, or **AC**, indicating saving after a commit (for example, **SFBC** or **SFAC**).

The *patch details* value has the following format.

```
<entity-patch-number>:<user-patch-number>:<status>
```

The *entity-patch-number* value is the current patch number of the entity being modified, and is zero (**0**) if it is new or unset.

The *user-patch-number* value is the patch number that is currently assigned to the user.

The *status* value is **N** if the entity is being modified for the first time against the patch number assigned to the user, **O** if the entity has been modified previously against the patch number assigned to the user and the patch number is still open, or **C** if the entity has been modified previously against the patch number assigned to the user and the patch number is closed.

Development Function Security Hook

The **jadeDevelopmentFunctionSelected** user-supplied hook function is called whenever a specific development task is requested.

The user name from the log-on screen, the task requested, and the entity against which the task is requested are passed to this function.

The **jadeDevelopmentFunctionSelected** function has the following format.

```
int JOMAPI jadeDevelopmentFunctionSelected(const Character *userName,
                                           const Character *taskName,
                                           const Character *entityName);
```

The **jadeDevelopmentFunctionSelected** function returns the following values.

Value	Description
0	The user has access
Non-zero	The user has no access, and the routine failed

userName

The **userName** input parameter specifies the user name of the developer specified in the **User Id** text box in the JADE log-on form.

taskName

The **taskName** input parameter specifies the JADE development environment task that the developer requested.

For details, see "Browser Functions", "Painter Functions", and "Administration Functions", later in this section.

entityName

The **entityName** input parameter specifies the JADE development environment entity against which the task is requested.

For details, see "Browser Functions", "Painter Functions", and "Administration Functions", later in this section.

Browser Functions

The JADE development environment browser tasks and entities that you can define in the **jadeDevelopmentFunctionSelected** function **taskName** and **entityName** input parameters are listed in the following table.

Task Name	Entity Name	Description
addApplication	*Schema-name*	Adds an application
addCategory	*Schema-name*	Adds a constant category
addConstant	*Schema-name::type-name*	Adds a class constant
addDelta	*Schema-name*	Adds a delta
addExposedList	*Schema-name*	Adds an exposure
addFile	*Schema-name::database-name*	Adds a map file
addFunction	*Schema-name*	Adds an external function
addFunctionLibrary	*Schema-name*	Adds a library for external function
addGlobalConstant	*Schema-name*	Adds a global constant
addHTMLDocument	*Schema-name*	Adds an HTML document
addInterface	*Schema-name*	Adds an interface
addLibrary	*Schema-name*	Adds a library for external methods
addMethodView	*Schema-name*	Adds a method view
administration	*Schema-name*	Administration option (sign-on form)
addPackage	*Schema-name*	Adds a package
addSchema	*Schema-name*	Adds a schema
addSchemaView	*Schema-name*	Adds a schema view
addType	*Schema-name*	Adds a class
addVariable	*Schema-name::class-name*	Adds an attribute
alMethod	*Schema-name::type-name*	Adds a JADE method
applicationOptions	*Schema-name::application-name*	Changes application options
broadcastMessage	*Schema-name*	Sends a message to other users
browsePackageClasses	*Schema-name::package-name*	Opens the Export Browser or Import Browser

Task Name	Entity Name	Description
browsePackageUsages	*Schema-name::package-name*	Shows all schemas that use a package
browseSchemaEntityViews	*Schema-name*	Opens a Methods View Browser window
changeCategory	*Schema-name::category-name*	Changes a constant category
changeConstant	*Schema-name::type-name::constant-name*	Updates a class constant
changeFile	*Schema-name::database-name::file-name*	Changes a map file definition
changeGlobalConstant	*Schema-name::global-constant-name*	Changes a global constant definition
changeHTMLDocument	*Schema-name::document-name*	Changes an existing HTML document
changeMethodView	*Schema-name::view-name*	Changes a method view
changeType	*Schema-name::class-name*	Updates class definition
changeVariable	*Schema-name::class-name::property-name*	Updates a property
checkInAll	*Schema-name::delta-name*	Checks in all methods in delta
checkInMethod	*Schema-name::type-name::method-name*	Checks in a selected method
checkOutMethod	*Schema-name::type-name::method-name*	Checks out a selected method
clearHistory	*Schema-name*	Clears the history of accessed classes and methods
closeDelta	*Schema-name::delta-name*	Closes the selected delta
closeSchemaWindows	*Schema-name*	Closes open windows for the current schema
collectionReferences	*Schema-name::type-name*	Collection references to a class or type
compile	*Schema-name::type-name::routine-name*	Compiles a selected method
compiledMethods	*Schema-name*	Status List compiled methods
compileType	*Schema-name::type-name*	Compiles a class, primitive type, or interface
compareMethods	*Schema-name::type-name::method-name*	Compares original and checked out method
constantReferences	*Schema-name::global-constant-name*	Global constant references
constantsMenu	*Schema-name::type-name*	Class Constants Menu
constantText	*Schema-name::type-name::constant-name*	Updates constant text
constantUsages	*Schema-name::type-name::constant-name*	Class constant references
copy	*Schema-name::type-name::method-name*	Copies selected text to a clipboard

Task Name	Entity Name	Description
copyForm	*Schema-name::locale-name::form-name*	Copies a form (saves as)
copyMethod	*Schema-name::class-name::method-name*	Copies a method (saves as)
copyMethodView	*Schema-name::view-name*	Changes a method view
copyXaml	*Schema-name::class-name*	Saves a XAML document as another name
cut	*Schema-name::type-name::method-name*	Saves selected text to a clipboard
debugExecute	*Schema-name::*JadeScript*::method-name*	Debugs a **JadeScript** method
deleteForm	*Schema-name::locale-name::form-name*	Deletes a form
displayVersionInfo	*Schema-name*	Displays items that are versioned in the schema
editForm	*Schema-name::locale-name::form-name*	Edits a form
editorWindow	*Schema-name*	Toggles the display of the browser editor pane
executeit	*Schema-name::*JadeScript*::method-name*	Executes a selected **JadeScript** method or Workspace
expandHierarchy	*Schema-name::class-name*	Expands branch of hierarchy tree
expandSchemaHierarchy	*Schema-name*	Expands Schema Browser branch
externalMethod	*Schema-name::type-name*	Adds an external method
extractActiveX	*Schema-name::component-library-name*	Extracts the ActiveX type library
extractDotNetLib	*Schema-name::component-library-name*	Extracts the .NET assembly
extractExposureList	*Schema-name::exposure-name*	Extracts the exposure list
extractHTMLDocument	*Schema-name::document-name*	Extracts an HTML document
extractInterface	*Schema-name::interface-name*	Extracts an interface
extractMethod	*Schema-name::class-name::method-name*	Extracts a method
extractMethods	*Schema-name::type-name::feature-name*	Extracts all methods in the active window
extractMethodView	*Schema-name::view-name*	Extracts all methods in the current schema from the method view
extractPackage	*Schema-name::package-name*	Extracts a package
extractView	*Schema-name::view-name*	Extracts the Schema View
fileOutMethod	*Schema-name::type-name::method-name*	Extracts the selected method

Task Name	Entity Name	Description
findCodePosition	*Schema-name::type-name::method-name*	Finds/Replaces text in the active window
findInterface	*Schema-name*	Searches for an interface
findReplace	*Schema-name::type-name::method-name*	Finds/Replaces text in the active window
findType	*Schema-name*	Searches for a class
formWizard	*Schema-name*	Opens painter and starts the Form Wizard
functionText	*Schema-name::function-name*	Updates text for external function
generateJavaScriptConsumer	*Schema-name::function-name*	Generates a JavaScript Web service consumer
globalConstantText	*Schema-name::global-constant-name*	Updates global constant text
globalSearch	*Schema-name*	Finds/Replaces in multiple methods
graphicalReferences	*Schema-name::type-name*	Shows relationships graphically
graphViewOptions	*Schema-name::class-name*	Graph view options
htmlText	*Schema name::document-name*	Edits HTML text
implementors	*Schema-name::type-name::method-name*	Method implementors
importAutomationLib	*Schema-name*	Imports an ActiveX automation library using a wizard
importControlLib	*Schema-name*	Imports an ActiveX control library using a wizard
importDotNetLib	*Schema-name*	Imports a .NET assembly
inErrorMethods	*Schema-name*	Status List In Error methods
inspectAllInstances	*Schema-name::class-name*	Inspects persistent instances of selected class and subclasses
inspectInstances	*Schema-name::class-name*	Inspects persistent instances of a selected class
inspectTransients	*Schema-name::class-name*	Inspects shared transient instances of a selected class
inspSchemaTransients	*Schema-name*	Inspects shared transient objects in a schema
installSchema	*Schema-name*	Loads an extract file
interfaceMapper	*Schema-name::class-name*	Maps a class to one or more interfaces

Task Name	Entity Name	Description
interfaceMethodMappings	*Schema-name*	Shows interfaces in which implemented methods are defined
librarySuperschemas	*Schema-name*	Views superschemas
localImplementors	*Schema-name::type-name::method-name*	Method local implementors
localImplementorsRefs	*Schema-name::type-name::method-name*	References to local implementors of the method
localSenders	*Schema-name::type-name::method-name*	Method local references
maintainExposedList	*Schema-name::exposure-name*	Updates an Exposure Definition
maintainPackage	*Schema-name::package-name*	Changes a package definition
maintainSchemaView	*Schema-name::view-name*	Updates a Schema View
mapFileText	*Schema-name::database-name::file-name*	Updates map file text
menuAdd	*Schema-name*	Adds an external database
menuAddCurrency	*Schema-name*	Adds a currency format
menuAddLongDate	*Schema-name*	Adds a long date format
menuAddNumeric	*Schema-name*	Adds a numeric format
menuAddShortDate	*Schema-name*	Adds a short date format
menuAddTime	*Schema-name*	Adds a time format
menuChange	*Schema-name::external-database-name*	Changes an external database definition
menuEditFormat	*Schema-name::format-name*	Changes the format
menuExtract	*Schema-name::external-database-name*	Extracts an external database
menuHistory	*Schema-name*	Accesses the history of classes and methods
menuInterfaces	*Schema-name*	Browses interfaces
menuLoad	*Schema-name*	Loads an external database
menuLocales	*Schema-name*	Browses locales
menuODBCAdd	*Schema-name*	Adds a relational view
menuODBCChange	*Schema-name::view-name*	Updates a relational view
menuODBCExtract	*Schema-name::view-name*	Extracts a relational view
menuODBCLoad	*Schema-name*	Loads a relational view
menuODBCPrint	*Schema-name::view-name*	Prints a relational view
menuODBCRemove	*Schema-name::view-name*	Removes a relational view
menuPrint	*Schema-name::external-database-name*	Prints an external database

Task Name	Entity Name	Description
menuReferences	*Schema-name::format-name*	Format references
menuRemove	*Schema-name::external-database-name*	Removes an external database
menuRemoveFormat	*Schema-name::format-name*	Removes a format
menuStrings	*Schema-name*	Strings menu
menuTranslateForms	*Schema-name*	Translates forms
menuView	*Schema-name::external-database-name*	External Database View menu
messages	*Schema-name::type-name::method-name*	Messages sent by a selected method
methodsMenu	*Schema-name::type-name*	Methods menu
methodsWindow	*Schema-name*	Toggles the display of the browser Methods List
methodsViewMenu	*Schema-name*	Methods View menu
methodText	*Schema-name::type-name::method-name*	Updates method text
mnuActiveXComponent	*Schema-name*	Accesses the Import submenu
mnuAddConsumer	*Schema-name*	Adds a new Web service consumer
mnuAddPatch	*Schema-name*	Adds a new patch number
mnuAdHocIndexAdd	*Schema-name*	Adds a new ad hoc index
mnuAdHocIndexBuild	*Schema-name*	Builds and ad hoc index definition
mnuAdHocIndexCancel	*Schema-name*	Cancels the building of an ad hoc index definition
mnuAdHocIndexChange	*Schema-name*	Changes an ad hoc index definition
mnuAdHocIndexDelete	*Schema-name*	Deletes an ad hoc index definition
mnuAdHocIndexDrop	*Schema-name*	Drops an ad hoc index definition
mnuAdHocIndexLoad	*Schema-name*	Loads an ad hoc index definition from an XML file
mnuAdHocIndexRun	*Schema-name*	Runs the ad hoc index controller application
mnuAdHocIndexSave	*Schema-name*	Saves an ad hoc index definition to an XML file
mnuAllPatches	*Schema-name*	Shows all patch numbers
mnuBreakpoints	*Schema-name*	Browses breakpoints

Task Name	Entity Name	Description
mnuBrowseActiveXLib	*Schema-name*	Displays the ActiveX Browser
mnuBrowseAdHocIndexes	*Schema-name*	Browses ad hoc index definitions
mnuBrowseApplications	*Schema-name*	Browses applications
mnuBrowseClasses	*Schema-name*	Browses classes
mnuBrowseClassesInUse	*Schema-name*	Browses in-use classes
mnuBrowseConstants	*Schema-name*	Browses global constants
mnuBrowseConsumer	*Schema-name*	Browses Web service consumers
mnuBrowseDeltas	*Schema-name*	Browses deltas
mnuBrowseExposedLists	*Schema-name*	Browses exposures
mnuBrowseExposureList	*Schema-name*	Browses exposures
mnuBrowseFunctions	*Schema-name*	Browses external functions
mnuBrowseHTML	*Schema-name*	Browses HTML documents
mnuBrowseInterfaces	*Schema-name*	Browses interfaces
mnuBrowseLibraries	*Schema-name*	Browses libraries
mnuBrowseMaps	*Schema-name*	Browses class maps
mnuBrowseMethods	*Schema-name*	Browses methods
mnuBrowseMethsAndVars	*Schema-name*	Browses methods and properties
mnuBrowsePatches	*Schema-name*	Displays the Patches Browser
mnuBrowseRPS	*Schema-name*	Browses Relational Population Service (RPS) mappings
mnuBrowseStatusCurrent	*Schema-name*	Views methods last modified by the current user
mnuBrowseTypes	*Schema-name*	Browses primitive types
mnuBrowseVariables	*Schema-name*	Browses properties
mnuBrowseViews	*Schema-name*	Browses schema views
mnuChangeConsumer	*Schema-name::consumer-name*	Reloads the Web service consumer
mnuChangeDelta	*Schema-name::delta-identifier*	Updates the delta identifier and description
mnuChangeInterface	*Schema-name::interface-name*	Updates the interface definition
mnuChangedMethods	*Schema-name*	Browses changed methods
mnuChangePatch	*Schema-name::patch-number*	Edits patch number details

Task Name	Entity Name	Description
mnuCheckedOutMethods	*Schema-name*	Browses checked out methods
mnuClosePatch	*Schema-name::patch-number*	Closes a patch number
mnuClosedPatches	*Schema-name*	Shows closed patches
mnuCurrentUserPatches	*Schema-name*	Shows patches for current user only
mnuDelta	*Schema-name::type-name::method-name*	Displays the Delta submenu
mnuDispPatchSummary	*Schema-name*	Opens a patch summary window
mnuDotNetComponent	*Schema-name*	Accesses the .NET submenu
mnuExportBrowser	*Schema-name*	Browses export packages
mnuExternalDb	*Schema-name*	Browses external databases
mnuExtractConsumer	*Schema-name::consumer-name*	Extracts Web service consumer classes
mnuExtractFunction	*Schema-name*	Extracts the selected external function or library
mnuExtractPatch	*Schema-name::patch-number*	Extracts patches for a specific patch number
mnuFindAtCaret	*Schema-name::type-name::routine-name*	Finds the next occurrence of the text on which the caret is positioned
mnuFindTransientLeaks	*Schema-name*	Finds transient object leaks
mnuFindUnusedClassEntities	*Schema-name*	Finds unused entities
mnuGenerateTestCase	*Schema-name*	Generates a set of stub methods that can be used for Web service consumer unit testing
mnuImportBrowser	*Schema-name*	Browses import packages
mnuMacroEdit	*Schema-name*	Edits a keystroke macro
mnuMacroLibrary	*Schema-name*	Selects a persistent keystroke macro
mnuMacroPlay	*Schema-name*	Plays a keystroke macro
mnuMacroRecord	*Schema-name*	Starts or stops the recording of a keystroke macro
mnuMethIFaceMapping	*Schema-name::type-name::method-name*	Opens the Interface Implementation Mapper dialog
mnuMethImplementedBy	*Schema-name::type-name*	Displays the Interface method maps to the following dialog

Task Name	Entity Name	Description
mnuMethodStatusList	*Schema-name*	Browses method status list
mnuMethodsView	*Schema-name*	Browses method views
mnuNRemoveUnusedLocalVars	*Schema-name*	Removes unused local variables
mnuOpenPatches	*Schema-name*	Shows all open patches
mnuRefreshPatchList	*Schema-name*	Refreshes the patches browser
mnuRelationalViews	*Schema-name*	Browses relational views
mnuRemoveConsumer	*Schema-name::consumer-name*	Removes the Web service consumer
mnuRemoveInterface	*Schema-name::interface-name*	Removes the interface
mnuReopenPatch	*Schema-name::patch-number*	Reopens a closed patch number
mnuReorgAbort	*Schema-name*	Aborts the current active reorganization
mnuReorgAllSchemas	*Schema-name*	Reorganizes all schemas marked for reorganization
mnuReorgRestart	*Schema-name*	Restarts a stopped reorganization
mnuReorgSchema	*Schema-name*	Reorganizes the current schema
mnuReorgTransition	*Schema-name*	Initiates the transition phase of the reorganization
mnuReplaceIndent	*Schema-name::type-name::routine-name*	Replaces the editor indentation with tabs or spaces
mnuRpsCreateJcf	*Schema-Name::rps-mapping-name*	Creates an exclude command file for the selected RPS mapping
mnuSchemaReorg	*Schema-name*	Displays the reorganization submenu
mnuSetPatch	*Schema-name::patch-number*	Sets to a patch number
mnuSetPatchNumber	*Schema-name*	Opens the Set Patch Number dialog
mnuShowAllMethods	Schema-name	Displays methods that were updated or added in the selected patch for all schemas
mnuShowCompositeView	*Schema-name*	Toggles the display of all version views

Task Name	Entity Name	Description
mnuShowImportedIFace	*Schema-name::type-name*	Toggles the display of imported interfaces
mnuShowMethods	Schema-name	Displays methods that were updated or added in the selected patch for the currently selected schema
mnuShowPatchSummary	*Schema-name*	Opens a patch summary window
mnuTypeIFaceMapping	*Schema-name::type-name*	Opens the Interface Implementation Mapper dialog
mnuUnreferencedMethods	*Schema-name*	Browses unreferenced methods
mnuUnsetPatch	*Schema-name::patch-number*	Unsets a set patch number
mnuVersionCompare	*Schema-name::type-name::method-name*	Compares original and versioned method
mnuVersionedMethods	*Schema-name*	Opens the Versioned Methods dialog
mnuWebServicesOptions	*Schema-name::type-name::method-name*	Specifies SOAP header selections
monitor	*Schema-name*	Invokes the JADE Monitor
moveClass	*Schema-name::class-name*	Moves a class
moveMethod	*Schema-name::class-name::method-name*	Moves a method to a different class
movePatchDetail	*Schema-name::patch-number*	Moves patches to a different patch number
newSchemaEntityView	*Schema-name*	Displays the Methods View Browser window
newWorkspace	*Schema-name*	Opens a new Workspace
openView	*Schema-name::view-name*	Opens a Class Browser with a Schema View
openWorkspace	*Schema-name*	Opens an existing Workspace
painter	*Schema-name*	Opens the Painter window
paste	*Schema-name::type-name::method-name*	Pastes text from the clipboard
printMethod	*Schema-name::type-name::method-name*	Prints a selected method source
printSchema	*Schema-name*	Prints classes in a schema
printSelected	*Schema-name::type-name*	Prints a selected object
printSetup	*Schema-name*	Printer set up options

Task Name	Entity Name	Description
promoteConstant	*Schema-name::type-name::constant-name*	Promotes a class constant to a global constant
propertiesWindow	*Schema-name*	Toggles the display of the browser Properties List
quickNavigation	*Schema-name*	Mnemonic shortcut dialog
readReferences	*Schema-name::class-name*	Property read references
redo	*Schema-name*	Performs multiple redo operations
references	*Schema-name::type-name::routine-name*	Method references
registerServer	*Schema-name*	Displays the common File dialog
reloadHTMLDocument	*Schema-name::document-name*	Reloads an HTML document
removeActiveX	*ActiveX-library-name*	Removes the ActiveX control or object type library
removeApplication	*Schema-name::application-name*	Removes an application
removeBreakpoint	*Schema-name*	Removes a breakpoint
removeCategory	*Schema-name::category-name*	Removes a constant category
removeConstant	*Schema-name::type-name::constant-name*	Removes a class constant
removeDotNetLib	*Schema-name::component-name*	Removes a .NET assembly
removeExposedList	*Schema-name::exposure-name*	Removes an exposure list
removeFile	*Schema-name::database-name::file-name*	Removes a map file
removeFromList	*Schema-name::type-name::method-name*	Removes a method from the list
removeFunction	*Schema-name::function-name*	Removes an external function
removeFunctionLibrary	*Schema-name::library-name*	Removes a function library
removeGlobalConstant	*Schema-name::global-constant-name*	Removes a global constant
removeHTMLDocument	*Schema-name::document-name*	Deletes an HTML document
removeLibrary	*Schema-name::library-name*	Removes an external method library
removeMethod	*Schema-name::type-name::method-name*	Removes a method
removeMethodView	*Schema-name::view-name*	Removes a method view
removePackage	*Schema-name::package-name*	Delete a package
removeSchema	*Schema-name*	Removes the highlighted schema
removeSchemaView	*Schema-name::view-name*	Removes a Schema View
removeType	*Schema-name::class-name*	Removes a class

Task Name	Entity Name	Description
removeVariable	*Schema-name::class-name::property-name*	Removes a property
renameMethod	*Schema-name::type-name::method-name*	Renames a method
reorgOption	*Schema-name*	Reorganizes the database
resetRoot	*Schema-name*	Resets the root class back to **Object**
runApplication	*Schema-name*	Runs an application
save	(varies)	Saves the text in the active window
saveAll	*Schema-name::type-name*	Extracts the selected class and all its subclasses
saveAs	*Schema-name*	Saves Workspace as a different name
saveMethod	*Schema-name::type-name::method-name*	Saves a method
saveMethodAs	*Schema-name::type-name::method-name*	Saves a method as another name
saveType	*Schema-name::type-name*	Extracts the selected class or primitive type
schemaMenuFormats	*Schema-name*	Formats
schemaSave	*Schema-name*	Extracts a schema
schemaViewText	*Schema-name::view-name*	Updates Schema View text
selectClass	*Schema-name::class-name*	Selects the specified class in the Class Browser
selectSchema	*Schema-name*	Selects a schema
setApplication	*Schema-name::application-name*	Sets to an application
setBookmark	*Schema-name*	Sets a bookmark
setDelta	*Schema-name::delta-name*	Sets to a delta
setRoot	*Schema-name*	Sets the root class
setText	*Schema-name::document-name*	Modifies text in an HTML document
setView	*Schema-name::view-name*	Sets to a Schema View
showAllMethodsView	*Schema-name*	Displays all method views in the Methods View Browser
showBubbleHelp	*Schema-name*	Toggles the display of bubble help in browser lists
showHistory	*Schema-name::type-name::method-name*	Displays the patch history of the selected entity
showInherited	*Schema-name::class-name*	Shows inherited method, properties, and constants

Task Name	Entity Name	Description
showInterfaceMappings	*Schema-name::class-name::method-name*	Shows interface methods mapped to the class method
showProtected	*Schema-name*	Toggles the display of protected properties or methods
showPublic	*Schema-name*	Toggles the display of public properties or methods
showReadOnly	*Schema-name*	Toggles the display of read-only properties or methods
showTypeHistory	*Schema-name::type-name*	Displays the patch history of the selected type
sortedByValue	*Schema-name*	Sorts constants in ASCII collating sequence
sortedOrder	*Schema-name::class-name*	Changes Class Browser to an alphabetical list
subclass	*Schema-name::locale-name::form-name*	Allows form to be subclassed
superschemas	*Schema-name*	Views superschemas
textSchema	*Schema-name*	Updates schema text
toggleBreakpoint	*Schema-name::type-name::method-name*	Toggles a breakpoint
typeReferences	*Schema-name::type-name*	References to selected class, primitive type, or interface
typesMenu	*Schema-name*	Classes menu
typeText	*Schema-name::type-name*	Updates text for selected class, primitive type, or interface
uncheckoutAll	*Schema-name::delta-name*	Unchecks out all methods in a selected delta
uncheckoutMethod	*Schema-name::type-name::method-name*	Undoes checkout of a method
uncompiledMethods	*Schema-name*	Status List uncompiled methods
undo	*Schema-name*	Performs multiple undo operations
unregisterServer	*Schema-name*	Unregisters an ActiveX type library
unsetDelta	*Schema-name::delta-name*	Unsets a delta
unusedLocalVars	*Schema-name::type-name::method-name*	Shows unused local variables
updateComments	*Schema-name::global-constant*	Updates global constant text
updateMethod	*Schema-name::type-name::method-name*	Methods being modified

Task Name	Entity Name	Description
updateReferences	*Schema-name::class-name*	Property update references
usages	*Schema-name::class-name::property-name*	Property references
userPreferences	*Schema-name*	User preferences
validateSchema	*Schema-name*	Validates a schema
variablesMenu	*Schema-name::class-name*	Properties menu
variableText	*Schema-name::class-name::property-name*	Updates text for a property
viewForm	*Schema-name::class-name*	Opens Painter for a selected form
viewMethod	*Schema-name::type-name::method-name*	Methods about to be viewed
viewMethods	*Schema-name::delta-name*	Views method in a delta
zoomIn	*Schema-name::class-name*	Graph zoom in
zoomOut	*Schema-name::class-name*	Graph zoom out

Painter Functions

The JADE development environment Painter tasks and entities that you can define in the **jadeDevelopmentFunctionSelected** function **taskName** and **entityName** input parameters are listed in the following table.

Task Name	Entity Name	Description
addForm	*Schema-name::locale-name::superform-name*	Adds a form in Painter
copyForm	*Schema-name::locale-name::original-form-name*	Saves the Painter form as another name
deleteForm	*Schema-name::locale-name::form-name*	Deletes the Painter form
saveForm	*Schema-name::locale-name::form-name*	Saves the Painter form
subclass	*Schema-name::locale-name::form-name*	Saves the Painter form as a subclass name

Administration Functions

The JADE development environment administration tasks and entities that you can define in the **jadeDevelopmentFunctionSelected** function **taskName** and **entityName** input parameters are listed in the following table.

Task Name	Entity Name	Description
backupDataBase	None	Backs up database
menuAdmin	None	Administration menu
menuAdminRemove	None	Removes sources
menuAdminReset	None	Resets preferences
menuEnable	None	Enables patch versioning
menuExtractPatch	None	Extracts a patch version

Task Name	Entity Name	Description
menuFileSaveAndExit	None	Saves changes and exits
menuFileSaveAndLogoff	None	Saves changes and logs off
menuPatch	None	Patch versioning menu
menuRecreate	None	Recreates patch history
menuRemoveHistory	None	Removes a patch history
menuSetNumber	None	Sets up patch number

JADE Smart Thin Client Security

The JADE thin client mode provides optional encryption support for the presentation client and application server communication. As the application server can run only one of the following types of security at any time, you must change the security settings in the JADE initialization file on the application server and presentation client nodes if you want to change to another type of security.

- None
- Internal
- User-supplied external Dynamic Link Library (DLL)
- Secure Sockets Layer (SSL)

Enabling JADE Smart Thin Client Security Encryption

The [JadeAppServer] and [JadeThinClient] sections of the JADE initialization file contain the **RPCEncryptionEnabled** and **RPCEncryptionHookDLL** parameters.

- If the **RPCEncryptionEnabled** parameter in the [JadeAppServer] section is set to **true**, the defined encryption is enforced on all presentation clients attached to that application server.
- The **RPCEncryptionHookDLL** parameter identifies the JADE or user-supplied encryption library. The available types of encryption are:
  - **RPCEncryptionHookDLL=Internal**, which uses Windows-supplied 40 bit encryption.
  - **RPCEncryptionHookDLL=SSL_TLS**, which enables the SSL security feature. When SSL security is enabled, the parameters whose names start with **SSL** are used. For details, see "Secure Sockets Layer (SSL) Security", later in this chapter.
  - **RPCEncryptionHookDLL=<*name*>**, which calls a user-supplied DLL with the name specified in the **<*name*>** value.

Note The RPC encryption DLL must be thread-safe; that is, it must be able to handle multiple threads calling this library simultaneously.

The settings of the **RPCEncryptionEnabled** and **RPCEncryptionHookDLL** parameters in the [JadeAppServer] section override any setting in the [JadeThinClient] section if the combination of these two parameters are not values listed in the following table, except that the connection fails if the **RPCEncryptionEnabled** parameter is not set to **true** on both the application server and the presentation client when the **RPCEncryptionHookDLL** parameter is set to **SSL_TLS**.

Application Server	Presentation Client
False	**False**, or **True** when **RPCEncryptionHookDLL** = ***<dll-name>*** or **Internal**
True, with **RPCEncryptionHookDLL** = **Internal**	**True**, with **RPCEncryptionHookDLL** = **Internal**
True, with **RPCEncryptionHookDLL** = **SSL_TLS**	**True**, with **RPCEncryptionHookDLL** = **SSL_TLS**
True, with **RPCEncryptionHookDLL** = ***<dll-name>***	**True**, with **RPCEncryptionHookDLL** = ***<dll-name>***

For more details, see "Application Server Section [JadeAppServer]" and "JADE Presentation Client Section [JadeThinClient]" under "JADE Thin Client Sections", in the *JADE Initialization File Reference*. See also "Network Message Encryption", earlier in this chapter, for details about encrypting network messages.

Controlling JADE Smart Thin Client Application Execution

You can control the applications that can be executed from any presentation client attached to an application server.

The [JadeAppServer] section of the JADE initialization file contains the **EnableAppRestrictions** and **AllowSchemaAndApp*<n>*** parameters.

When the **EnableAppRestrictions** parameter is set to **false** (the default value), all applications can be executed by presentation clients. If this parameter is set to **true**, only the schema and optional applications specified in the **AllowSchemaAndApp*<n>*** parameters can be executed.

For details, see "Application Server Section [JadeAppServer]" under "JADE Thin Client Sections", in the *JADE Initialization File Reference*.

Secure Sockets Layer (SSL) Security

The Secure Sockets Layer (SSL) provides an alternative form of secure communication between the presentation client and the application server. The secure connection is available between the presentation client and the application server.

This secure connection uses the Secure Sockets Layer (SSL) protocol to achieve authentication and encryption. This implementation is based upon open source code from the **OpenSSL** project. (For more information, see http://www.openssl.org.)

For details about the valid combinations of the **RPCEncryptionEnabled** parameter and **RPCEncryptionHookDLL** parameter on both the application server and presentation client nodes, see "Enabling JADE Smart Thin Client Security Encryption", earlier in this chapter.

You can use the sample certificates supplied with the JADE release, which allow the presentation client to authenticate the application server, optionally allow the application server to authenticate the presentation client, and start an encrypted connection. These certificates are self-signed samples, which you should *not* use in a production environment where full trust is required.

It is the responsibility of the system administrator in a production environment to obtain and manage certificates; that is, security of these private keys and signing certificates.

It is your responsibility to obtain or generate the appropriate certificates. The physical security of private key files is also your responsibility.

Note As the **OpenSSL** libraries upon which this implementation is based support only hard-coded English single-byte character set messages, they are not translatable. The Unicode version of JADE converts any messages into Unicode before they are logged to the **jommsg.log** file.

The sample self-signed certificates are located in the directory in which your JADE binary files are installed and are named **server.pem** and **client.pem** for the application server and presentation client certificates, respectively.

The **SSLProxyHost** and **SSLProxyPort** parameter values are used if they are defined. If no values are specified for these parameters, the Windows registry is checked to see if a proxy server has been specified. The **"same proxy for all protocols"** value is used if it is specified or the **"secure"** proxy server entry if the **"same proxy for all protocols"** value is not specified. If neither value is present, a direct connection is attempted. See also "Automatically Detecting Proxy Settings" under "Presentation Client Considerations", later in this section.

Tip Encryption software adds processing overheads (and therefore time) to the operation of both the presentation client and the application server. You should therefore take care to balance the requirements of secure communications with the extra processing load on the application server, as it must encrypt and decrypt data sent between it and all connected presentation clients.

Specifying JADE Initialization File Parameters for SSL Security

To support SSL security, the [JadeAppServer] and [JadeThinClient] sections of the JADE initialization file can contain the parameters summarized in the following table.

Parameter	Value Type	JadeAppServer	JadeThinClient
RPCEncryptionEnabled	Boolean	false	false
RPCEncryptionHookDLL	See description	Internal	Internal
SSLCertificateAuthorityFile	File name	Not specified	Not specified
SSLCertificateAuthorityPaths	Directory list	Not specified	Not specified
SSLCertificateFile	File name	Not specified	Not specified
SSLCipherNames	See description	Not specified	Not specified
SSLMethodName	See description	SSLv23	SSLv23
SSLPrivateKeyFile	File name	Not specified	Not specified
SSLProxyDetect	Boolean	Not applicable	true
SSLProxyHost	TCP/IP host address	Not applicable	Not specified
SSLProxyPort	TCP/IP port number	Not applicable	Not specified
SSLRemoteCertCheck	Boolean	false	true
SSLRemoteVerify	Boolean	true	true
SSLSecurePort	TCP/IP port number	443	443
SSLVerifyDepth	Integer	9	9

If a parameter value is invalid, an appropriate message is logged on the node that contains the invalid value and the connection between the presentation client and the application server fails. In addition, a "fail to connect message" is usually logged on the other end of the connection.

The **SSLProxyHost** and **SSLProxyPort** parameter values are used if they are defined. If no values are specified for these parameters, the Windows registry is checked to see if a proxy server has been specified. As the verify depth (specified in the **SSLVerifyDepth** parameter) is honored, you should not set the value of this parameter to zero (**0**).

The **"same proxy for all protocols"** value is used if it is specified or the **"secure"** proxy server entry if the **"same proxy for all protocols"** value is not specified. If neither value is present, a direct connection is attempted.

For details about these JADE initialization file parameters, see "JADE Thin Client Sections", in the *JADE Initialization File Reference*.

JADE Initialization File Examples

The following is an example of the minimum entries that are required in the [JadeAppServer] and [JadeThinClient] sections of the JADE initialization file to have an authenticated and encrypted connection from the presentation client to the application server. In this example, the presentation client authenticates the application server, ensuring that the application server has a valid certificate, but the application server does not authenticate the presentation client.

Example 1

```
[JadeAppServer]
RPCEncryptionEnabled=true
RPCEncryptionHookDLL=SSL_TLS
SSLCertificateFile=C:\Jade\bin\server.pem

[JadeThinClient]
RPCEncryptionEnabled=true
RPCEncryptionHookDLL=SSL_TLS
```

To get a mutually authenticated and encrypted connection, the following entries are required in the [JadeAppServer] and [JadeThinClient] sections. Each end of the connection ensures that the other node has a valid signed certificate.

Example 2

```
[JadeAppServer]
RPCEncryptionEnabled=true
RPCEncryptionHookDLL=SSL_TLS
SSLCertificateFile=C:\Jade\bin\server.pem
SSLCertificateAuthorityFile=C:\Jade\bin\server.pem
SSLRemoteVerify=true

[JadeThinClient]
RPCEncryptionEnabled=true
RPCEncryptionHookDLL=SSL_TLS
SSLCertificateFile=C:\Jade\bin\client.pem
SSLCertificateAuthorityFile=C:\Jade\bin\client.pem
```

These examples assume that the **server.pem** and **client.pem** certificates are self-signed and authorize themselves.

Application Server Considerations

You do not have to alter the application server command line parameter to enable SSL support, but you must define or update the appropriate parameters in the [JadeAppServer] section of the JADE initialization file on the application server node.

When the application server is started, it listens for JADE thin client connections on the port number specified by the **SSLSecurePort** parameter (defaulting to port number **443**) on all network interfaces. You can restrict the application server to listen on a specific network interface, if required, by setting the **AppServer** parameter in the [JadeAppServer] section to the appropriate value.

If the presentation client is connected to the application server through a proxy server, the GUI version of the application server shows that the JADE thin client has the address of the proxy server and not the real address of the JADE thin client.

If the GUI version of the application server (**jadapp**) is used, the command line displays the secure port number on which it is listening.

Tip If you require both normal (non-secure) and secure connections from presentation clients, you must start a second JADE application server. Each application server must use a different JADE initialization file that has the **RPCEncryptionEnabled** parameter in the [JadeAppServer] section set to **false** in one JADE initialization file and to **true** in the other.

Proxy Server Consideration

To allow the creation of an end-to-end tunnel, the proxy server needs to support the **HTTP/1.1 CONNECT** method (which is described in the "draft-luotonen-web-proxy-tunneling-01" document on the Internet).

Ensure that the proxy server does not require client authentication.

Presentation Client Considerations

You do not have to alter the presentation client command line parameter to enable SSL support, but you must define or update the appropriate parameters in the [JadeThinClient] section of the JADE initialization file on the appropriate presentation client nodes.

If your connection to the Internet is through a proxy server, ensure that the **SSLProxyHost** and **SSLProxyPort** parameters in the [JadeThinClient] sections of the JADE initialization file on each presentation client node are set correctly. (These values are likely to be the same as those that you defined in your Web browser settings dialog, but you should contact your network administrator for exact details.)

When the application is starting on the presentation client, a splash screen may display the host and port to which the network connection is being made. This will be the proxy server if that is enabled, or the application server host name or port number defined by the **SSLSecurePort** parameter (which defaults to **443**) in the [JadeThinClient] section of the JADE initialization file on each presentation client node.

Automatically Detecting Proxy Settings

The presentation client can attempt automatic detection of proxy settings. If the **SSLProxyHost** and **SSLProxyPort** parameters are set, these values are used. If no values are specified for these parameters, automatic detection reads the Windows registry and looks for the proxy settings used by Microsoft Internet Explorer.

If the **Address** and **Port** values are defined (in the appropriate fields of the Local Area Network (LAN) Settings dialog accessed from the Internet Options dialog **Connections** sheet), these values are used.

If the **Address** and **Port** values are not defined, it uses the values from the proxy address and port fields of the **HTTP** type row if the **Use same proxy server for all protocols** check box is selected. If this is not selected, it uses the proxy address and port fields of the **Secure** type row. These are located on the Proxy Setting dialog (accessed from the Local Area Network (LAN) Settings dialog, which is accessed from the Internet Options dialog **Connections** sheet).

If no values are detected, a direct connection to the application server is made. (See also "Secure Sockets Layer (SSL) Security", earlier in this chapter.)

Certificate Considerations

Your certificates must be structurally correct and be signed by a valid Certificate Authority. Checks are not performed on the validity of the certificate date range or that the connection is to whom the certificate says it should be.

The certificate files must be in PEM-encoded format, and they cannot require pass phrases.

Using an SSL Security Library Protocol

You can connect to external systems from JADE by using an SSL library protocol instead of the TCP/IP protocol when the **TcpIpConnection** class **sslContext** property contains a reference to a **JadeSSLContext** transient object.

JadeSSLContext connections use digital certificates in X509 format, which are written to disk in Privacy-Enhanced Electronic Mail (PEM)-encoded certificate (PEM) format. The transient **JadeX509Certificate** class stores the digital certificates in X509 format.

SSL is a secure communication protocol on top of an already established TCP/IP connection.

SSL libraries are generated from publicly available third-party sources, maintained by the OpenSSL Group (www.openssl.org). JADE supports TLS version 1, TLS version 1.1, and TLS version 1.2.

For details about the **JadeSSLContext** and **JadeX509Certificate** classes and the properties and methods defined in these classes, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

Deployed JADE Database Inspection Security

To restrict access to class instance inspection in a deployed (run time) database, you can implement the **allowedToInspect** security filter method for the class or classes to which you want to restrict access.

The **allowedToInspect** method must have the following signature.

```
allowedToInspect(userName: String;
                 password: String): Boolean;
```

If the **allowedToInspect** method is present on a class, the Schema Inspector prompts for a user name and password, and calls the **allowedToInspect** method, passing these values as parameters. If the **allowedToInspect** method returns **true**, the instances of that class are displayed. If this method returns **false**, a message box is displayed, informing the user that he or she does not have sufficient security rights to inspect instances of that class.

Note As the **allowedToInspect** method defaults to **false** in a runtime application, no instances can be inspected if you do not implement the **allowedToInspect** method to return **true** in a runtime system. (This ensures that any user of the Schema Inspector in a runtime application can inspect only those classes to which access has been specifically permitted by the application software developer.)

The **allowedToInspect** method applies to the class in which it is implemented and to any subclasses of that class. To restrict deployment inspection access for all classes, implement the **allowedToInspect** method for the **Object** class.

The following example shows the implementation of the **allowedToInspect** method in a user-defined **Employee** class to restrict deployment inspection access to that class by all users other than **wilbur2**, who has a password of **taonga**.

```
allowedToInspect(userName, password: String): Boolean;
begin
    if userName = "wilbur2" and password = "taonga" then
        return true;
    else
        return false;
    endif;
end;
```

For details about the Schema Inspector, see Chapter 1, "Inspecting a Deployed Database", in the *JADE Schema Inspector Utility User's Guide*.

Schema Source File Security

The encryption hooks provided by the schema extract and load facilities enable you to incorporate data encryption algorithms of your choice in your schema extract files. These hooks encrypt the JADE method source code in your schema extract files, enabling you to release schema extract files without making their source code easily visible.

Caution If you extract an encrypted schema for deployment by a third-party, ensure that you extract the encrypted schema to a location different from that of your source schema. If you subsequently load the encrypted schema (for testing purposes, for example), your method source code is lost if you load it into the same database that contains your original source files, which are not saved during the decryption process.

Example hook routines are included in the **demodll.cpp** demonstration file, provided in the **demodll** folder in the JADE **examples** directory. For further information if your JADE licence includes a support agreement, contact JADE Support.

When extracting method source code with encryption enabled, the following actions are performed.

1. JADE looks in the **SchemaEncryptionHookLibrary** parameter in the [JadeSecurity] section of the JADE initialization file for the name of your user-defined encryption library (DLL). JADE uses a default encryption algorithm if you do not supply your own library. For details, see "JADE Security Section [JadeSecurity]", in the *JADE Initialization File Reference*.
2. The compiler attempts to load the encryption library by calling **LoadLibrary** and gets the address of your predefined encryption hook routine by calling **GetProcAddress**. An error is raised if the library cannot be located in the JADE executable (binary) directory or in the current path.
3. The source code is passed to your user-supplied encryption routine. The encryption routine can encrypt the source in any way and can change its length. For example, the encryption routine could embed an identifier at the beginning of the source to identify the type of encryption algorithm that is used.
4. The encrypted source is written to the extract file.

When loading a schema extract file containing encrypted source code, the following actions are performed.

1. JADE looks in the **SchemaEncryptionHookLibrary** parameter in the [JadeSecurity] section of the JADE initialization file for the name of your user-defined encryption library (DLL). JADE uses a default decryption algorithm if you do not supply your own library.
2. JADE attempts to load the encryption library by calling **LoadLibrary** and gets the address of your predefined decryption hook routine by calling **GetProcAddress**. An error is raised if the library cannot be located in the JADE executable (binary) directory or in the current path.

3. The compiler passes the encrypted source to your user-supplied decryption routine, which restores the source to its original form.

4. The method source is compiled.

To retain confidentiality, the source code is not saved by the load process. (Any load or compile error that is raised during the load process displays only the error code and message, as the source is not available. The developer of the deployed schema must take the appropriate action, using the source of that schema.) For details about extracting and loading encrypted schemas, see "Encrypting Schema Source Files", in Chapter 10 of the *JADE Development Environment User's Guide*.

Schema Encryption and Decryption Hooks

The **encryptSchemaSource** user-supplied Encryption hook function is called when creating schema extract files with the **Encrypt Sources** check box in the Extract dialog **Schema Options** sheet checked (that is, it is enabled).

The **encryptSchemaSource** function has the following format.

```
int JOMAPI encryptSchemaSource(const BYTE    *inData,
                               Size          inLength,
                               BYTE          **pOutData,
                               Size          *pOutLength);
```

The **decryptSchemaSource** user-supplied Decryption hook function is called when loading schema extract files containing encrypted method sources. The **decryptSchemaSource** function has the following format.

```
int JOMAPI decryptSchemaSource(const BYTE    *inData,
                               Size          inLength,
                               BYTE          **pOutData,
                               Size          *pOutLength);
```

The **encryptSchemaSource** and **decryptSchemaSource** functions return the following values.

Value	Description
0	The encryption or decryption routine was successful
Non-zero	The encryption or decryption routine failed

inData

The **inData** input parameter specifies the input data stream for encryption or decryption.

inLength

The **inLength** input parameter specifies the length of the input data stream.

pOutData

Use the **pOutData** output parameter to specify the location of the output data stream after encryption or decryption.

pOutLength

Use the **pOutLength** output parameter to specify the length of the encrypted or decrypted output data stream.

JADE Report Writer Security

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.

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 provided by instances of that class to use the security mechanism to return an integer value that indicates the type of access that the user has.

The constants provided by this class and listed in the following table are returned by the methods in subclasses of the **JadeReportWriterSecurity** class.

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

For more details, see the **JadeReportWriterManager** and **JadeReportWriterSecurity** classes, in Chapter 1 of the *JADE Encyclopaedia of Classes*.

Internet Access Support

Security when accessing a JADE application from the Internet can be defined at the levels described in the following subsections.

- Windows Security
- Web Server Security
- Application Security
- JadeHttp.ini File Security
- Network Device Security
- Suppressing the Logging of Messages
- Uploading Text Input Files to the Web
- Web Form Security

As unknown XML Web service consumers can access an XML Web service provider, the Web server must provide basic security services at the protocol level. The **Web Services** sheet on the Define Application dialog enables you to specify that the application is secure. In addition, the **Web Services** sheet of the Define Class dialog enables you to specify whether a Web services class uses the default security of the application or you can override this value and select whether the class is secure (that is, it uses the HTTPS protocol) or not secure (that is, it uses the HTTP protocol).

In addition, you can use the TCP communication protocol for *direct* Web services messages between JADE systems. For more details, see "Building Web Services Applications", in Chapter 10 of the *JADE Developer's Reference*.

Windows Security

Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, and Windows Server 2008 security enables you to protect your computer and its resources when accessing your JADE applications from the Internet, by requiring assigned user accounts and passwords. To control access to computer resources, limit the user rights of these accounts.

To ensure that an application specified in a Web browser cannot cause an attachment to a non-JADE environment within Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, or Windows Server 2008, you must specify the name of the Web application to which users can connect as a section name within the **jadehttp.ini** file, as follows.

```
[<application-name>]
```

An attempt to attach to an application is made only to the applications whose names are specified as section names in the **jadehttp.ini** file. (See also "JadeHttp.ini File Security", later in this section.)

Web Server Security

In addition to Windows 10, Windows 8, Windows 7, Windows Server 2016, Windows Server 2012, and Windows Server 2008 security, the Internet Information Server (IIS) or Apache HyperText Transfer Protocol (HTTP) Server Web server provides additional security by restricting Internet Protocol (IP) addresses or by checking user names and passwords.

It also provides a Secure Sockets Layer (SSL) security protocol. This protocol is layered between its service protocols HyperText Transfer Protocol (HTTP) and Transfer Control Protocol / Internet Protocol (TCP/IP). The SSL provides data encryption, server authentication, and message integrity for a TCP/IP connection. (For more details, see "Secure Sockets Layer (SSL) Security" under "JADE Smart Thin Client Security", earlier in this chapter.)

Connections from the Internet Information Server (IIS) to the JADE application can be through a named pipe or a TCP/IP connection. Connections from the Apache HTTP Server can be through a TCP/IP connection only.

Notes The main advantage of a TCP/IP connection over a named pipe connection is that the machine hosting the Web server can be different from the machine that is running the JADE application, to provide greater security by using firewalls.

A TCP/IP connection is also slightly faster than a named pipe connection.

For details about implementing a TCP/IP connection, see "Connecting to JADE Applications from Internet Information Server (IIS)" under "JADE Configurations", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Application Security

Your JADE user application can have its own defined security scheme; for example, you can write your own sign-on screen.

JadeHttp.ini File Security

The **jadehttp.dll** derives a path for the **jadehttp.ini**, **jadehttp.log**, and transfer files, ensuring that they are completely secure. The directories are derived from the **jadehttp.dll** directory for these files, as follows.

1. If security permits it, **jadehttp** attempts to create the required directories if they do not exist. The required directories, based on the example **c:\jade\bin\jadehttp.dll** file, are listed in the following table.

Directory	Description	Example
dll-path_dll-name\ini\	For the initialization file and the default error ***dll-name*.htm** file	c:\jade\bin_jadehttp\ini\jadehttp.ini
dll-path_dll-name\logs\	For all ***dll-name*** logs	c:\jade\bin_ jadehttp\logs\jadehttp.log
dll-path_dll-name\transfer\	For the location of any temporary files created during file transfer	c:\jade\bin_jadehttp\transfer\...

2. If the initialization file option to specify the log path is set, the logs directory is not created or used.
3. If the initialization file option to specify the transfer path is set, the transfer directory is not created or used.
4. As the DLL can be renamed, its name is included. The DLL path therefore does not need to contain any other files, and users cannot access any of the files listed in the previous table.

For details about implementing a TCP/IP connection from the IIS to JADE applications and firewall security, see "Connecting to JADE Applications from Internet Information Server (IIS)" and "Configuring JadeHttp for Remote Connections", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Network Device Security

You can use a network device to restrict access to and protect data in your JADE applications.

Suppressing the Logging of Messages

To ensure the security of data, set the value of the **Trace** parameter in the [Jadehttp Logging] section of the **jadehttp.ini** file for IIS or the JadeHttp_Trace directive in the JADE **mod_ jadehttp** module for Apache to **true**.

When the value is **true**, messages logged to the **jadehttp.log** file do not include any of the text sent or received from the client, as this text could contain personal information, passwords, credit card details, and so on.

Logged messages then only acknowledges that a message has been received or sent, because it is not possible to distinguish what is sensitive data and what is not.

Uploading Text Input Files to the Web

To provide increased security for applications running in HTML thin client mode, a text file input by using the **TextBox** class **webInputType** property with a value of **Web_InputType_File** must be processed in the event that resulted in the file upload occurring (for example, in the **click** method of a **Completed** button).

Caution To prevent malicious use of files uploaded to Web-enabled applications, the files are removed as soon as the event that resulted in their upload has completed. You should therefore process the file immediately or move it into a directory that is not available from the Web if you require that file for future processing.

For details about the **webInputType** property, see Chapter 2 of your *JADE Encyclopaedia of Classes*. For details about specifying the directory to which the file is written, see "Controlling the Location of Files Uploaded via a Web Application", in Chapter 2 of the *JADE Installation and Configuration Guide*.

Web Form Security

You can use the **Form** class **secureForm** property to specify that a Web form is secure and the **webEncodingType** property to contain a reference to the content type used to submit the Web form to the JADE application. (For details, see Chapter 2 of the *JADE Encyclopaedia of Classes*.)

JADE Monitor Security

If JADE development security is enabled, the JADE Monitor sets up menus via calls to the **jadeDevelopmentFunctionSelected** user-supplied hook function (for details, see "JADE Development Environment Security", earlier in this chapter) to provide accessibility to JADE Monitor activities based on one of the following conditions.

- If the **Monitor_SuperUser**, **Monitor_NormalUser**, or **Monitor_ReadOnlyUser** security level task returns **true**, that security profile is applied.
- If none of the security level calls returns **true**, the user is initially assumed to be a read-only user. However, you can use the JADE Monitor tasks and the **JadeMonitorSchema** entity defined in the **jadeDevelopmentFunctionSelected** function **taskName** and **entityName** input parameters to fine-tune access to JADE Monitor individual menus and activities.

For details about JADE Monitor menus and activities, see Chapter 2 of the *JADE Monitor User's Guide*.

Setting Up JADE Monitor Security

When a user has signed on to the JADE Monitor, the **MonitorSecurity** and **MonitorDefaultUser** parameters in the [JadeMonitorSecurity] section of the JADE initialization file on the database server control access to monitor functionality. (For more details, see the *JADE Initialization File Reference*.)

An example of the [JadeMonitorSecurity] section is shown in the following example.

```
[JadeMonitorSecurity]
MonitorDefaultUser=ReadOnlyUser
MonitorSecurity=Monitor|DevelopmentSecurity|None
wilbur=SuperUser
testuser=NormalUser
```

If the **MonitorSecurity** parameter is not defined when a JADE application is launched, the following values are written to the JADE initialization file on the database server and all users will all have **SuperUser** access. (Although the **MonitorDefaultUser** parameter is included in the following example, it is not used when the **MonitorSecurity** parameter is set to **None**.)

```
[JadeMonitorSecurity]
MonitorDefaultUser=ReadOnlyUser
MonitorSecurity=None
```

Default JADE Monitor Security Levels

The JADE Monitor security levels are described in the following subsections.

SuperUser Level

The **SuperUser** security level has access to all JADE Monitor activities and menus.

This is the default security level for systems running without JADE development security. The **NormalUser** and **ReadOnlyUser** security levels are enabled only when a user-defined library is specified in the **DevelopmentSecurityLibrary** parameter in the [JadeSecurity] section of the JADE initialization file.

NormalUser Level

The **NormalUser** security level provides access to all JADE Monitor activities and menus except for interrupting a user, forcing off a user, and forcing off a node.

ReadOnlyUser Level

The **ReadOnlyUser** security level provides access to all JADE Monitor activities other than those with significant system impact. This is the default security level when development security is enabled if no other security level or menu items are enabled. The following menu commands or activities are disabled for this security level.

- **Force Off User** command
- **Force Off Node** command
- **Dump Selected Node** and **Dump All Nodes** commands
- **Interrupt User** command
- Process request statistics activities in the Process Information activity group
- Persistent Object Activity group activities (for example, the **Most Accessed Objects** activity)
- **Method Analysis** and **Selective Analysis** activities
- **Lock Summary By Class**, **Oid**, **Chronology**, and **Contention** activities in the Lock Analysis activity group
- Activities in the Node Sampling activity group

JADE Monitor Tasks and Entities

The JADE Monitor tasks that you can define in the **jadeDevelopmentFunctionSelected** function **taskName** input parameter for the **JadeMonitorSchema** value of the **entityName** input parameter are listed in the following table. (The **Monitor_** tasks in the first three rows of the following table override use of any of the tasks in subsequent rows of the table, which assume an initial value of **Monitor_ReadOnlyUser**.)

Task Name	Allows access to...
Monitor_SuperUser	All JADE Monitor functions
Monitor_NormalUser	Most JADE Monitor functions other than those under "NormalUser Level", earlier in this chapter
Monitor_ReadOnlyUser	Read-only access only (for details, see "ReadOnlyUser Level", earlier in this chapter)
Setup Process Statistics	The **Setup Process Statistics** activity
Local Request Statistics	The **Local Request Statistics** activity
Remote Request Statistics	The **Remote Request Statistics** activity
Method Analysis	The **Method Analysis** and **Selective Analysis** method profiling activities
Persistent Object Activity	Class and object access activities in the Persistent Object Activity group

Task Name	Allows access to...
Lock Analysis	Activities in the Lock Analysis activity group
Node Sampling	The capture and analysis activities in the Node Sampling activity group
mnuUserForceOffUser	**Force Off User** menu command
mnuUserForceOffNode	**Force Off Node** menu command
mnuUserDumpNode	**Dump Selected Node** menu command
mnuUserDumpAllNodes	**Dump All Nodes** menu command
mnuUserShutdownMonitor	**Shutdown Selected Monitor Process** menu command
mnuUserInterrupt	**Interrupt User** menu command
mnuUserRegLocal	**Register for Process Stats (Local)** menu command
mnuUserRegRemote	**Register for Process Stats (Remote)** menu command
mnuUserDeregProcess	**Deregister for Process Stats (Local)** and **Deregister for Process Stats (Remote)** menu commands

Menus, menu commands, and Navigator pane items not listed in this table can be accessed by all JADE Monitor users.

Synchronized Database Environment (SDE) Security

As you are likely to want to employ the public Internet Protocol (IP) network infrastructure as an economic means to network geographically separated sites, the Synchronized Database Environment (SDE) enables you to specify by name or IP address a list of hosts from which the node is willing to accept in-bound SDS connections. For details, see "Security and Authentication", in Chapter 1 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

Relational Population Service (RPS) Security

The RPS Datapump application **JadeRpsDataPump**, which runs on the RPS node from the schema in which the RPS mapping is defined, connects to the RDBMS database and applies transactions to it, implementing the incremental object replication. To allow the Datapump application to be started without user intervention, it must be able to log on to the SQL Server with a valid account. You can accomplish this by performing one of the following actions.

- Run the RPS node under a Windows account that has the required access rights and database permissions.
- Configure an SQL Server account in the RPS mapping. (For details, see "Setting Up the RPS Options", in Chapter 15 of the *JADE Development Environment User's Guide*.)

Note The first of these actions is more secure, as it avoids the need to save password information in the RPS mapping meta data.

For details, see "RPS Datapump Application", in Chapter 2 of the *JADE Synchronized Database Service (SDS) Administration Guide*.

Chapter 3 JADE Application Programming Interface (API)

The JADE Application Programming Interface (API) calls are listed in the following table. Function prototypes for these calls are defined in the **jomcalls.h** header in the **include** directory on the JADE release medium.

API Call	For details, see…	Description
jomAbortTransaction	Aborting a Transaction	Aborts a transaction
jomArmExceptionHandler	Arming an Exception Handler	Requests a message when an exception occurs
jomBeginLoad	Beginning a Load	Indicates the beginning of an atomic read transaction
jomBeginLock	Beginning a Transaction Lock	Ensures that objects referenced are the latest editions
jomBeginNotification	Beginning a Notification	Requests a notification message from the server
jomBeginTransaction	Beginning a Transaction	Indicates the beginning of an atomic transaction
jomCallPrimMethod	Calling a Primitive Method	Invokes a primitive instance method
jomCallPrimTypeMethod	Calling a Primitive Type Method	Invokes a primitive type method
jomCauseEvent	Causing an Event	Requests a user notification event from the server
jomCauseSDEEvent	Causing an SDE Event	Achieves inter-system event notifications in a Synchronized Database Environment (SDE)
jomCauseSDSEvent	Causing an SDS Server Event	Achieves inter-system event notifications in a Synchronized Database Service (SDS)
jomChangeAccessMode	Changing the Database Access Mode	Indicates the mode in which the user wants to use the database
jomClearBuffers	Clearing JADE Object Manager Buffers	Marks the object buffer as obsolete
jomCloneObject	Creating a Shallow Copy of an Object	Creates a shallow copy of an object
jomCreateObject	Creating an Object	Creates an instance of the specified class
jomDeleteObject	Deleting an Object	Deletes the specified object
jomDeregisterTerminationCallBack	Canceling the Termination Method Invocation	Canceling invocation of the termination callback method
jomDisarmExceptionHandler	Disarming an Exception	Disarms an armed exception handler
jomEndLoad	Ending a Load	Indicates the end of a read transaction

API Call	For details, see…	Description
jomEndLock	Ending a Transaction Lock	Ends a read-only transaction lock bracket
jomEndNotification	Ending a Notification	Cancels a begin notification request from the server
jomEndTransaction	Ending a Transaction	Indicates the end of a transaction
jomFinalize	Closing a Node	Flushes object buffers and closes all open classes and the database
jomFinalizeApplication	Finalizing an Application	Finalizes an application
jomGetBufferEdition	Getting a Buffer Edition	Loads the edition of the object buffer in this node
jomGetExecInstances	Getting Execution Instances	Returns the object identifiers (oids) of the environmental objects for the process
jomGetLatestEdition	Checking an Edition	Loads the edition of the object in the database
jomGetLockStatus	Getting a Lock Status	Gets the status of an object lock
jomGetNodeContextHandle	Getting a Node Context Handle	Gets the current node context handle for the process
jomGetNotificationInfo	Getting Notification Information	Requests additional notification information
jomGetObject	Getting an Object	Copies object buffer contents to the specified area
jomGetProperty	Getting a Property	Gets the value of a property of the specified object
jomGetPropertyDesc	Getting a Property Description	Gets the description of a property of the specified object
jomGetSystemStatus	Getting System Status	Requests the general state of a process
jomImportMethod	Calling an Imported Method	Causes the execution of an imported method
jomInheritMethod	Inheriting a JADE Object Manager Method	Causes the execution of a superclass method of the same name
jomInitialize	Initializing a Node	Initializes a node
jomInitializeApplication	Initializing an Application	Initializes an application
jomInvokeMethod	Invoking a Method on a Saved Object	Sends the specified target method containing a variable list of parameters to the receiver
jomIsKindOf	Checking an Object is a Valid Class Instance	Checks that an object is a valid instance of a class
jomLockObject	Locking an Object	Places a lock on an object
jomRegisterTerminationCallBack	Cleaning Up Resources	Specifying a method for cleaning up resources on termination

API Call	For details, see…	Description
jomRaiseException	Raising an Exception	Executes the exception handling methods for a specified exception
jomRespondsTo	Checking an Object Implements an Interface	Checks that an object implements an interface
jomSendMsg	Sending a Message	Sends normal messages to objects
jomSendEventMsg	Sending an Event Message	Sends event messages to objects
jomSendTypeMsg	Sending a Type Message	Sends normal messages to types
jomSetProperty	Setting a Property	Sets the value of a property of a specified object
jomSignOff	Closing a Process	Finalizes the process associated with the handle
jomSignOn	Opening a Process	Initializes a process in the current mode
jomTerminate	Terminating Applications in a Node	Terminates all active methods in the current node
jomUnlockObject	Unlocking an Object	Removes a lock placed on an object
jomUpdateEdition	Updating an Edition	Forces an increment in the object edition

For all calls other than the **jomInitialize**, **jomSignOn**, and **jomFinalize** calls, the **pHandle** parameter is the **pProcessHandle** parameter returned from the **jomSignOn** call. (For more details, see "Opening a Process", later in this chapter.)

Note When you set the **TerminateProcessOnDisconnect** parameter in the [JadeClient] section of the JADE initialization file to **true**, unwanted side-effects may occur when the connection to the JADE database server came from a JADE Object Manager API connection in multiuser mode, as the underlying program that opened the connection is also terminated.

The JADE Object Manager also provides the **jomGetMessageText** API call, whose function prototype is defined in the **jomerrs.h** header in the **include** directory on the JADE release medium. For details, see "Getting Text for a System or Error Message", later in this chapter.

JADE Object Manager API calls that use a **pNodeHandle** parameter have a null (**""**) or zero (**0**) node handle parameter value for the main node. Otherwise, the value should be that as returned by the **jomInitialize** API call for the node.

For details about version checking, see "Checking the Dynamic Link Library Version", later in this chapter.

Initializing a Node

The **jomInitialize** call, shown in the following example, must be the first call made to the JADE Object Manager module.

```
int jomInitialize(DskHandle  *pNodeHandle,
                  DskParam   *pDirectory,
                  DskParam   *pServerType,
                  DskParam   *pIniFile)
```

This call opens the database and initializes the buffers for a new node. The Windows process can call this API once only, unless a call to the **jomFinalize** API call is issued before the subsequent call.

The code in the following example uses the C++ API to sign on to the JADE Object Manager. (If you use the code in this example, make sure that you change variable values so that they are appropriate for connecting to your system.)

```
DskParam directoryParam;
paramSetCString(directoryParam, "<dbPath>");
DskParam serverTypeParam;
paramSetCString(serverTypeParam, JOM_MULTIUSER_TAG);
ShortDskParam iniParam;
paramSetCString(iniParam, "<iniFile>");

// open the jom database - once per node
DskHandle nodeHandle;
int result = ::jomInitialize(&nodeHandle, &directoryParam, &serverTypeParam,
&iniParam);
CHECK_RESULT;
```

The parameters for the **jomInitialize** call are described in the following subsections.

pNodeHandle

The **pNodeHandle** parameter is a pointer to a handle structure that is loaded by the JADE Object Manager.

This handle refers to a node and must be passed later as the node handle in the **jomSignOn** call. All of the objects referenced in JADE Object Manager entry points always belong to a process. You can write your own program to initialize several processes.

The maximum number of processes for each node is limited only by the memory on that node; that is, it is hardware-dependent. Processes in a node share the same object buffers.

pDirectory

The **pDirectory** parameter is the path name of the location of the database schema description files. These file names start with an underscore (_) character; for example, the **_control.dat** file. Other database files can reside in other directories. Such directories must have been defined as the file path in the class map for the file.

pServerType

The **pServerType** parameter is the database server type. There are two server types, which have predefined names of **singleUser** and **multiUser**.

- **singleUser** specifies that the server program and database run in the single user mode.
- **multiUser** specifies that the program will open a client connection to the database server. The connection is specified in the [JadeClient] section **ServerNodeSpecifications** parameter in the JADE initialization file.

For details about the JADE initialization file, see the *JADE Initialization File Reference*, and for details about the JADE Remote Node Access utility, see the *JADE Remote Node Access Utility User's Guide*.

pIniFile

The **pIniFile** parameter is a pointer to the path and name of the file that contains a section describing extended client options for this process. For more details about these parameters, see the *JADE Initialization File Reference*.

Initializing an Application

The **jomInitializeApplication** call, shown in the following example, initializes the application by calling the global **initialize** method (if any) and the start-up method of the application (for example, an **initialize** event method), including those from any imported packages.

```
int jomInitializeApplication(const DskHandle   *pHandle,
                             bool               callInitialize,
                             const DskObjectId *pOid,
                             UInt32             lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

callInitialize

If you set the **callInitialize** parameter to **false**, the start-up method of the main application is not called but only those from any imported packages.

pOid

The **pOid** parameter is the optional parameter for the **initialize** event method of a non-GUI application. If the method has no parameter, it should be set to **null**.

For details, see the **Application** class **startApplicationWithParameter** method, in Chapter 1 of the *JADE Encyclopaedia of Classes*.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Finalizing an Application

The **jomFinalizeApplication** call, shown in the following example, finalizes the application by calling the close request method of the application (for example, a **finalize** event method), including those from any imported packages.

```
int jomFinalizeApplication(const DskHandle   *pHandle,
                           bool               callFinalize,
                           UInt32             lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

callFinalize

If you set the **callFinalize** parameter to **false**, the close request method of the main application is not called but only those from any imported packages.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting a Node Context Handle

The **jomGetNodeContextHandle** call, shown in the following example, is used to get the current node context handle for the process.

```
int jomGetNodeContextHandle(const DskHandle *pProcessHandle,
                            DskHandle       *pNodeContextHandle)
```

The parameters for this call are described in the following subsections.

pProcessHandle

The **pProcessHandle** parameter points to a **DskHandle** structure that identifies a specific process. This structure is populated by the call to **jomSignOn**.

This parameter can also be null, in which case the process for the calling thread is used.

pNodeContextHandle

The **pNodeContextHandle** parameter points to a **DskHandle** structure where this function copies the node handle for the process specified by the **pProcessHandle** parameter.

Closing a Node

The **jomFinalize** call, shown in the following example, performs the following actions.

- Flushes the object buffers
- Closes all open classes
- Closes the database for the node

```
int jomFinalize(const DskHandle *pNodeHandle)
```

Following the **jomFinalize** call, the **pNodeHandle** parameter for this context is made invalid for future calls.

The parameter for this call is described in the following subsection.

pNodeHandle

The **pNodeHandle** parameter is a pointer to the handle structure that is loaded in the **jomInitialize** call or the **jomInitializeAux** call by the JADE Object Manager. (For details about the **jomInitialize** call, see "Initializing a Node", earlier in this chapter.)

Opening a Process

The **jomSignOn** call, shown in the following example, initializes a process in the current node.

```
int jomSignOn(const DskHandle *pNodeHandle,
              DskHandle       *pProcessHandle,
              DskHandle       *pSecurityHandle,
              DskParam        *pSchemaName,
              DskParam        *pAppName,
              DskParam        *pUserName,
              DskParam        *pPassword,
              const DskParam  *pDbMode,
              const DskParam  *pDbUsage,
              UInt32          lineNo)
```

This call starts an application process in the specified subschema and creates the required application and process objects.

The **jomSignOn** API call must be called to start each new application, which has an associated JADE process.

Caution Every new **jomSignOn** API call must be made from a different operating system thread. Unpredictable results may be obtained if a **jomSignOn** call is done within the same thread in which a current process was initialized.

If the requested mode or usage conflicts with the current mode or usage of the database by existing users, the request fails and an exception is raised.

For user validation, the **jomSignOn** call operates in the following modes.

1. A user code (and optional password) is supplied. The JADE Object Manager directly invokes the **isUserValid** method on the schema **Global** instance, passing the **userName** and **password** properties received from the **jomSignOn** API call.

2. No user code is supplied (**null** parameter). The JADE Object Manager invokes the **getAndValidateUser** method on the schema global instance. If this succeeds, it then invokes the **isUserValid** method on the schema global instance, passing the **userCode** and **password** properties returned by the user method. (For more details, see "Global Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.)

JADE applications controlled by the JADE application controller (**jade.exe**) use the second mode; that is, no user code is supplied. Non–JADE applications can choose between either mode, but they would normally use the first mode; that is, a user code and optional password are supplied. The JADE ODBC interface uses the first mode.

The address of a **null**-valued security handle must be passed on the first call to the **jomSignOn** API call.

The security handle returned in the **pSecurityHandle** parameter following a successful sign-on call can be used in subsequent **jomSignOn** calls by the application controller or by other JADE tools, to avoid the need for users to log on multiple times when starting multiple applications in a session.

If the security handle is valid for the node and subschema combination, the JADE Object Manager permits the sign-on without invoking the user validation methods. A specified security handle is valid only for the subschema for which it was first obtained. (For more details, see "User-Validation Support", in Chapter 2.)

When an application is terminated, the application controller (or other tool) should make a **jomSignOff** API call to terminate the JADE process. A process (Windows thread) can call this API once only, unless a call to the **jomSignOff** API call is issued. (For details about the **jomSignOff** call, see "Closing a Process", later in this chapter.)

The code in the following example uses the C++ API to start an application process. (If you use the code in this example, make sure that you change variable values so that they are appropriate for your system.)

```
DskParam dbUsageParam;
paramSetInteger(dbUsageParam, DB_UPDATE);
DskParam dbModeParam;
paramSetInteger(dbModeParam, DB_SHARED);

DskParam schemaNameParam;
paramSetCString(schemaNameParam, "<schemaName>");
DskParam appNameParam;
paramSetCString(appNameParam, "<applicationName>");
DskParam userParam;
paramSetCString(userParam, "<myUser>");
DskParam passwordParam;
paramSetCString(passwordParam, TEXT(""));

// sign on to the database - once per process/thread
// note that nodeHandle was passed to the call to jomInitialize()
DskHandle dbHandle;
DskHandle securityHandle = { 0 };
result = ::jomSignOn(&nodeHandle, &dbHandle, &securityHandle, &schemaNameParam,
&appNameParam,
&userParam, &passwordParam, &dbModeParam, &dbUsageParam, __LINE__);
CHECK_RESULT;
```

The parameters for this call are described in the following subsections.

pNodeHandle

The **pNodeHandle** parameter specifies the handle obtained from the **jomInitialize** call. (For details about the **jomInitialize** call, see "Initializing a Node", earlier in this chapter.)

pProcessHandle

The **pProcessHandle** parameter points to a **DskHandle** structure that identifies a specific process. This structure is populated by the call to **jomSignOn**.

This parameter can also be null, in which case the process for the calling thread is used.

pSecurityHandle

The **pSecurityHandle** parameter is a handle returned following the successful log on to a node.

pSchemaName

The **pSchemaName** parameter is the name of the schema in which the application is defined.

pAppName

The **pAppName** parameter is the name of the application within the specified schema.

pUserName

The **pUserName** parameter is the user name (user code) obtained from the end-user.

pPassword

The **pPassword** parameter is the password obtained from the end-user and is optionally encrypted.

pDbMode

The **pDbMode** parameter specifies if the database process mode is shared or exclusive.

The database mode constants and their integer values are listed in the following table.

Constant	Value	Description
DB_SHARED	0	Enables multiple current users to open the database.
DB_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.

pDbUsage

The **pDbUsage** parameter specifies the database usage. The database usage constants and their integer values are listed in the following table.

Constant	Value	Description
DB_UPDATE	0	Enables applications to update the database.
DB_READ_ONLY	1	Specifies that your applications have read-only access to the database.
DB_NO_AUDIT	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.
DB_ODBC_LOGIN	3	Specifies that the login is from the JADE ODBC driver (that is, it is read-only). This value is for internal use only.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Closing a Process

The **jomSignOff** call, shown in the following example, finalizes the process associated with the **handle** parameter.

```
int jomSignOff(const DskHandle *pHandle,
               UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Cleaning Up Resources

When a process terminates, the **jomRegisterTerminationCallBack** call, shown in the following example, cleans up or de-allocates operating system resources that have been allocated.

```
int jomRegisterTerminationCallBack(const DskHandle *pHandle,
                                   const JadeTerminationCallBack cb,
                                   const void      *pParam,
                                   TerminationCallBackType callBackType,
                                   LineNumber       lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

cb

The **cb** parameter specifies a C++ method to be called when the current JADE node terminates or when the JADE process currently associated with the thread terminates (depending on the value specified for the **callBackType** parameter).

pParam

The **pParam** parameter is an arbitrary value to be made available to the termination method that is called.

callBackType

The **callBackType** parameter causes the specified termination method to be called when the current JADE node terminates. The value can be one of the following.

- **NodeTerminationCallBack** causes the method to be called when the current JADE node terminates.
- **ProcessTerminationCallBack** causes the method to be called when the JADE process currently associated

with the thread terminates.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Specifying a Termination Method

The specified termination method must have a signature in the following format.

```
int JOMAPI termination-method-name(DskHandle *pHandle,
                                   void      *pParam,
                                   bool      nodeTermination)
```

Multiple calls to **jomRegisterTerminationCallBack** with identical parameter values (except for the **lineNo** parameter) are ignored.

When the method is invoked, the parameters passed to the method are as follows.

pHandle

The **pHandle** parameter is an internal handle value.

pParam

The **pParam** parameter is the value of the **pParam** parameter specified when the method was registered by the **jomRegisterTerminationCallBack** call.

nodeTermination

The **nodeTermination** parameter is **true** if the method was invoked as a result of the JADE node being terminated or **false** if the method was invoked as a result of the JADE process being terminated.

Canceling the Termination Method Invocation

To cancel invocation of the termination method so that it is not invoked when termination occurs, use the **jomDeregisterTerminationCallBack** call, shown in the following example.

```
int jomDeregisterTerminationCallBack(const DskHandle *pHandle,
                                     const JadeTerminationCallBack cb,
                                     const void      *pParam,
                                     TerminationCallBackType callBackType,
                                     LineNumber       lineNo)
```

If the parameters do not match a previous call to the **jomRegisterTerminalCallBack** call, the call is ignored.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

cb

The **cb** parameter specifies a C++ method previously specified in a call to the **jomRegisterTerminationCallBack** call.

pParam

The **pParam** parameter is the value previously specified in a call to the **jomRegisterTerminationCallBack** call.

callBackType

The **callBackType** parameter is the value specified on the previous call to the **jomRegisterTerminationCallBack** call that is to be deregistered.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Allocated File Resource Cleanup Method Examples

The following example shows the use of the **jomRegisterTerminationCallBack** and **jomDeregisterTerminationCallBack** API calls for a method to clean up an allocated file resource.

```
result = jomRegisterTerminationCallBack((DskHandle *)0,
                                cleanUpFile,  // this is the method to invoke
                                (void*)pFile, // pointer to the file resource
                                ProcessTerminationCallBack,
                                __LINE__);
```

The method to be called has the following format.

```
int JOMAPI cleanUpFile(DskHandle *pHandle,
                       void      *pParam,
                       bool      nodeTermination)
{
      int result = 0;
      //...
      // use value in pParam to clean up the file resource
      //
      return result;
}
```

The method is deregistered by the **jomDeregisterTerminationCallBack** call shown in the following example.

```
result = jomDeregisterTerminationCallBack((DskHandle *)0,
                                          cleanUpFile,
                                         (void *)pFile,
                                          ProcessTerminationCallBack,
                                          __LINE__);
```

Beginning a Transaction

The **jomBeginTransaction** call, shown in the following example, indicates the beginning of an atomic transaction for the calling process.

```
int jomBeginTransaction(const DskHandle *pHandle,
                        BYTE            type,
                        UInt32          lineNo)
```

The buffers of all created, deleted, or updated objects in a specific transaction are held in the logical objects cache until it is full and are then written to the server.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

All changes made by the current process to objects in the database are done in an "all or nothing" fashion, depending on a subsequent **jomEndTransaction** or **jomAbortTransaction** call. (For more details, see "Aborting a Transaction" or "Ending a Transaction", later in this chapter.)

type

The **type** parameter specifies the type of transactions, whose values are listed in the following table.

Value	Description
0	Persistent transaction
1	Transient transaction

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Ending a Transaction

The **jomEndTransaction** call, shown in the following example, indicates the end of a transaction.

```
int jomEndTransaction(const DskHandle *pHandle,
                      BYTE            type,
                      UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

This call causes the commit of all changes made by the current process since the last **jomBeginTransaction** call. (For details, see "Beginning a Transaction", earlier in this chapter.)

All transaction locks held by the process are released automatically at this point.

type

The **type** parameter specifies the type of transactions, whose values are listed in the following table.

Value	Description
0	Persistent transaction
1	Transient transaction

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Beginning a Transaction Lock

The **jomBeginLock** call, shown in the following example, is used to ensure that objects referenced after the **jomBeginLock** call are the latest editions of the objects.

As objects are referenced, an implicit shared lock is acquired on each object, which causes the latest edition to be fetched from the server, if required.

```
int jomBeginLock(const DskHandle *pHandle,
                 UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Ending a Transaction Lock

The **jomEndLock** call, shown in the following example, is used to end the automatic lock state initiated previously by using the **jomBeginLock** call. (For details, see "Beginning a Transaction Lock", earlier in this chapter.)

```
int jomEndLock(const DskHandle *pHandle,
               UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Aborting a Transaction

The **jomAbortTransaction** call, shown in the following example, is used to abort a transaction.

```
int jomAbortTransaction(const DskHandle *pHandle,
                        BYTE            type,
                        UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**. This call causes all changes made by the process since the last **jomBeginTransaction** call to be rolled back. (For details, see "Beginning a Transaction", earlier in this chapter.)

type

The **type** parameter specifies the type of transactions, whose values are listed in the following table.

Value	Description
0	Persistent transaction
1	Transient transaction

As this parameter has been introduced for future implementation, use of the **type** parameter with a value of **1** is ignored.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Terminating Applications in a Node

The **jomTerminate** call, shown in the following example, terminates all active methods in the current process and terminates the corresponding application.

```
int jomTerminate(const DskHandle *pHandle,
                 BYTE            mode,
                 UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

mode

The **mode** parameter specifies the type of termination that is to be performed. The values that you can specify and their constant definitions are listed in the following table.

Constant	Value	Description
CONDITIONAL_TERMINATE	0	Causes the result "APPLICATION_TERMINATE_REQUEST" to be returned to the method currently executing on the top of the process stack. The method may relay the result to the method that caused its execution.
UNCONDITIONAL_TERMINATE	1	Causes the result "APPLICATION_TERMINATE" to be returned to the method currently executing on the top of the process stack. The method is required to pass the same result to the method that caused its execution.
TERMINATE_ABORT_OPERATION	2	Causes the unwinding of the process execution stack, due to a conditional or unconditional **jomTerminate** call, to stop. The application continues its execution.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Creating an Object

The **jomCreateObject** call, shown in the following example, creates an object of the specified class and returns the full object identifier (oid) of the created object in the **pOid** parameter.

```
int jomCreateObject(const DskHandle *pHandle,
                    DskObjectId     *pOid,
                    BYTE            type,
                    DskParam        *pParam1,
                    DskParam        *pParam2,
                    UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

Before making this call, the application must assign the correct class number (obtained by direct inquiries to the schema definition objects) to the **classId** field of the **pOid** parameter. Exclusive subobjects are created automatically.

Shared references to other objects from this object are the responsibility of your application and must be created by your application.

Any constructor method specified for this class of object is automatically executed. (For details, see "Using Constructors and Destructors", in Chapter 4 of the *JADE Development Environment User's Guide*.)

If the left-most bit of the **classId** field of the **pOid** parameter is on, a transient instance of the object is created; otherwise, a persistent object instance is created (that is, when the bit is set to **off**).

type

The **type** parameter specifies the lifetime of the object to be created, whose values are listed in the following table.

Type	Description
0	Indicated by the class specified in the **pOid** parameter
1	Transient object
2	Persistent object
3	Shared transient object

Types **1**, **2**, and **3** take precedence over type **0** (specified by the class of the **pOid** parameter).

pParam1 and pParam2

The **pParam1** and **pParam2** parameters are reserved for internal use only.

The **pParam1** and **pParam2** parameters are not required for normal user object creations and should be set to zero (**0**).

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Deleting an Object

The **jomDeleteObject** call, shown in the following example, deletes the object specified by the oid.

```
int jomDeleteObject(const DskHandle   *pHandle,
                    const DskObjectId *pOid,
                    DskParam          *pParam1,
                    DskParam          *pParam2,
                    UInt32            lineNo)
```

Exclusive subobjects are deleted automatically. Any subobject defined as shared is the responsibility of your application and must be deleted by your application.

Any destructor method specified for this class of object is automatically executed. (For details, see "Using Constructors and Destructors", in Chapter 4 of the *JADE Development Environment User's Guide*.)

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object to be deleted.

pParam1 and pParam2

The **pParam1** and **pParam2** parameters are reserved for internal use only.

The **pParam1** and **pParam2** parameters are not required for normal user object creations and should be set to zero (**0**).

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting a Property

The **jomGetProperty** call, shown in the following example, is used to get the value of a property of a specified object.

```
int jomGetProperty(const DskHandle   *pHandle,
                   const DskObjectId *pOid,
                   DskParam          *pProp,
                   DskParam          *pValue,
                   UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object to be deleted.

pProp

The **pProp** parameter is the name (or class element identifier) of the property.

pValue

The value of the property is returned in the **pValue** parameter.

The type of the property is returned in the header format part of the corresponding parameter.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting a Property Description

The **jomGetPropertyDesc** call, shown in the following example, is used to get the description of a property of a specified object.

```
int jomGetPropertyDesc(const DskHandle   *pHandle,
                       const DskObjectId *pOid,
                       DskParam          *pProp,
                       DskParam          *pValue,
                       UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object to be deleted.

pProp

The **pProp** parameter is the name (or class element identifier) of the property.

pValue

The description of the property is returned in the **pValue** parameter. The format of this parameter corresponds to the **JomPropertyDesc** type.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Setting a Property

The **jomSetProperty** call, shown in the following example, is used to set the value of a property of a specified object.

```
int jomSetProperty(const DskHandle   *pHandle,
                   const DskObjectId *pOid,
                   DskParam          *pProp,
                   const DskParam    *pValue,
                   UInt32            lineNo)
```

When objects are modified in transaction state, they must be locked exclusively. If the user has not done so, the JADE Object Manager locks the object automatically. The object remains locked until the end of the transaction.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object to be deleted.

pProp

The **pProp** parameter is the name (or class element identifier) of the property.

pValue

The **pValue** parameter contains the value that is to be assigned to the property. The type of the **pValue** parameter must be consistent with the **Property** type.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Locking an Object

The **jomLockObject** call, shown in the following example, is used to place a lock on an object.

```
int jomLockObject(const DskHandle   *pHandle,
                  const DskObjectId *pOid,
                  BYTE              type,
                  BYTE              duration,
                  Int32             waitTime,
                  unsigned int      invokeHandler,
                  unsigned int      *pSuccess,
                  UInt32            lineNo)
```

When an object is locked, the logical buffer is automatically loaded with the latest edition, to guarantee that a locked object buffer is always current for the duration of the lock.

Locks issued after a **jomBeginTransaction** or **jomBeginLoad** call are associated with the process specified by that call. (For details, see "Beginning a Transaction" or "Beginning a Load", elsewhere in this chapter.)

For more details about locking objects, see the **Object** class **lock** method, in Chapter 1 of the *JADE Encyclopaedia of Classes*.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter passed to this call is the oid of the object to be locked. If the object has any exclusive subobjects, these subobjects are not locked.

type

The **type** parameter specifies the type of the lock operation. The types of lock that can be specified in the **type** parameter are listed in the following table.

Constant	Value	Description
UNLOCKED	0	Result of a **jomGetObject** operation
SHARED_LOCK	1	Shared lock
RESERVED_LOCK	2	Reserve lock
EXCLUSIVE_LOCK	3	Exclusive lock

More than one shared lock can coexist for the object at any time but only one exclusive lock is allowed. Additionally, a reserve lock can coexist with any number of shared locks but is incompatible with another reserve lock or with an exclusive lock.

If the object to be locked is already locked in an incompatible manner by another process, a lock exception may be raised. For details, see the **invokeHandler** parameter, later in this section.

duration

The **duration** parameter specifies if the object is automatically unlocked at the end of transaction or at the end of the current session if no manual unlocks are issued.

The constant definitions for lock durations are listed in the following table.

Constant	Value
TRANSACTION_LOCK	0
SESSION_LOCK	1

waitTime

The **waitTime** parameter specifies the length of time the lock request waits if the object is currently locked by another user before a lock exception is reported back. The wait time values are listed in the following table.

Value	Action
< 0	No wait
= 0	Wait for the system-default wait time (from the JADE initialization file)
> 0	Actual time to wait (in milliseconds)

invokeHandler

If the **invokeHandler** parameter is set to zero (**0**), no exception is raised and no exception handler is executed if the lock cannot be obtained. If this parameter is set to a non-zero value, an exception is raised and the corresponding exception handler is invoked.

pSuccess

The **pSuccess** parameter is loaded with zero (**0**) if the lock operation failed. If the lock operation was successful, this parameter is loaded with a non-zero value.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Unlocking an Object

The **jomUnlockObject** call, shown in the following example, is used to remove the lock placed on the object whose object identifier is specified in the **pOid** parameter.

```
int jomUnlockObject(const DskHandle   *pHandle,
                    const DskObjectId *pOid,
                    UInt32            lineNo)
```

The **jomUnlockObject** call has no effect inside transaction load or lock states. All locked objects for a specified process are automatically unlocked at the corresponding end transaction, end load, or end lock time.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object whose lock is to be removed.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting a Lock Status

The **jomGetLockStatus** call, shown in the following example, is used to get the status of a lock on an object. If the process doing the call has the object locked, this call returns the lock information.

```
int jomGetLockStatus(const DskHandle   *pHandle,
                     const DskObjectId *pOid,
                     BYTE              *type,
                     BYTE              *duration,
```

```
                                DskObjectId       *pLockedBy,
                                UInt32            lineNo)
```

The parameters passed to this call from the lock placed on the object are listed in the following table.

Parameter	Description
pHandle	Pointer to a handle structure that is loaded by the JADE Object Manager
pOid	Oid of the object that has the lock
type	Type of lock operation
duration	Duration of the lock; that is, if the object is automatically unlocked at the end of transaction or at the end of the current session if no manual unlocks are issued
pLockedBy	Pointer to the process object that has the lock
lineNo	Line number in which an exception occurred

For details about the lock parameters, see "Locking an Object", earlier in this chapter.

Getting a Buffer Edition

The **jomGetBufferEdition** call, shown in the following example, loads the edition field of the **pOid** parameter with the value corresponding to the object buffer in this node.

```
int jomGetBufferEdition(const DskHandle *pHandle,
                        DskObjectId     *pOid,
                        UInt32          lineNo)
```

If no buffer was in memory, a buffer is loaded as a result of this call.

Use this call with the **jomGetLatestEdition** call to determine if the current object buffer is the latest edition for the object. (For details about the **jomGetLatestEdition** call, see "Checking an Edition", later in this chapter.)

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object whose edition is to be loaded.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Checking an Edition

The **jomGetLatestEdition** call, shown in the following example, loads the **pEdition** parameter for the object specified in the **pOid** parameter with the edition of the object held in the database physical buffer.

```
int jomGetLatestEdition(const DskHandle   *pHandle,
                        const DskObjectId *pOid,
                        Edition           *pEdition,
                        UInt32            lineNo)
```

The JADE Object Manager maintains a set of logical object buffers for each client node. These buffers are additional to the buffers provided by the underlying database management system. In a multiuser environment, these logical buffers may not necessarily reflect the current information in the database.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object whose edition is to be loaded.

pEdition

The **pEdition** parameter is updated with the latest edition corresponding to the object referenced in the **pOid** parameter. This edition could be different from the edition corresponding to the object in the client node.

To initially load the client node, a previous **jomGetBufferEdition** call must have been issued. (For details about the **jomGetBufferEdition** call, see "Getting a Buffer Edition", earlier in this chapter.)

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Updating an Edition

The **jomUpdateEdition** call, shown in the following example, forces an increment in the edition for the specified object, regardless of whether it has been modified.

```
int jomUpdateEdition(const DskHandle   *pHandle,
                     const DskObjectId *pOid,
                     UInt32            lineNo)
```

The JADE Object Manager automatically updates the edition of an object whenever it is modified.

This call can be used by your application to indicate a change in the edition of the object in which the application may be interested but that has not been caused by a direct updating of that object. For example, if a dictionary is being used to obtain the contents of a list and the name of an object in the list changes, this does not cause an update to the dictionary as such.

However, your application may mark the edition of the dictionary as having changed, in order to easily detect the need to update the list.

Note If this call is issued within transaction state, the object must be locked manually or automatically.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object whose edition is to be updated.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Sending a Message

The **jomSendMsg** call, shown in the following example, is used to send normal messages to objects.

```
int jomSendMsg(const DskHandle   *pHandle,
               const DskObjectId *pOid,
               DskParam          *pMessage,
               DskParam          *pParams,
               DskParam          *pReturn,
               UInt32            lineNo)
```

JADE provides a set of messages that can be sent to instances of system classes such as dictionaries. For a description of the messages available for the instances of each system class, see Volume 1 and Volume 2 (system classes) and Chapter 2, "Window Classes", of the *JADE Encyclopaedia of Classes*.

This call is also used to send user-defined messages to objects.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object to which this message is to be sent.

pMessage

The **pMessage** parameter is the message name.

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method or condition. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method or condition back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Sending a Type Message

The **jomSendTypeMsg** call, shown in the following example, sends a message to the type of the receiver, which is the root type identified by the **ClassNumber** parameter.

```
int JOMAPI jomSendTypeMsg(const DskHandle *pHandle,
                          ClassNumber     classNumber,
                          DskParam        *pMessage,
                          DskParam        *pParams,
                          DskParam        *pReturn,
                          LineNumber      lineNo)
```

If a type method is defined on an imported class, the receiver is the class of the exported class (*not* the imported class).

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

classNumber

The **classNumber** parameter is used to identify the class containing the root type instance to which this message is to be sent.

pMessage

The **pMessage** parameter is the message name.

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Calling a Primitive Method

The **jomCallPrimMethod** call, shown in the following example, is used to send a message to a primitive type.

```
int jomCallPrimMethod(const DskHandle *pHandle,
                      UINT            primNo
                      DskParam        *pValue,
                      DskParam        *pMessage,
                      DskParam        *pParams,
                      DskParam        *pReturn,
                      UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

primNo

The **primNo** parameter is the number of the primitive type of the value to which this message is to be sent.

pValue

The **pValue** parameter is the primitive type value to which the message is to be sent.

pMessage

The **pMessage** parameter is the message name (or class element identifier).

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method or condition. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method or condition back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Calling a Primitive Type Method

The **jomCallPrimTypeMethod** call, shown in the following example, invokes a primitive type; for example, from an external method.

```
int JOMAPI jomCallPrimTypeMethod(const DskHandle *pHandle,
                                 TypeNum         primitiveNumber,
                                 DskParam        *pMessage,
                                 DskParam        *pParams,
                                 DskParam        *pReturn,
                                 LineNumber      lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

primitiveNumber

The **primitiveNumber** parameter is the number of the primitive type of the value to which this message is to be sent.

pMessage

The **pMessage** parameter is the message name (or class element identifier).

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Calling an Imported Method

The **jomImportMethod** call, shown in the following example, is used within the method code to cause the execution of an imported method when there is a local method with the same name in the class (or superclass) to which the currently executing method belongs.

```
int jomImportMethod(const DskHandle   *pHandle,
                    const DskObjectId *pOid,
                    DskParam          *msg,
                    DskParam          *pParams,
                    DskParam          *pReturn,
                    UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object to which this message is to be sent.

msg

The **msg** parameter is the message name.

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method or condition. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method or condition back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Invoking a Method on a Saved Object

The **jomInvokeMethod** call, shown in the following example, is used to send the specified target method containing a variable list of parameters to the receiver, after switching to the specified target execution context.

```
int jomInvokeMethod(const DskHandle   *pHandle,
                    const DskObjectId *pTargetOid,
                    const DskObjectId *pParamTargetContext,
                    const DskObjectId *pTargetMethod,
                    DskParam          *pInParam,
```

```
                DskParam            *pOutParam,
                UInt32              lineNo)
```

After the method has finished, the execution context switches back to the current context. (This switch will typically be from the context of the package to the context of the user of the package he second optional parameter.) For more details about details about calling user methods from packages, see "Calling User Methods from Packages", in Chapter 8 of the *JADE Developer's Reference*.

As the application context used by **jomInvokeMethod** is transient, it can switch to a context only within the same process. The mechanism is *not* designed to call a method running in another process in the node or in another node. In addition, as the context is transient, any connection between a context and a method to be invoked must be set up again if an application is stopped and then restarted.

If you want to save events to be called persistently so that methods would still be called if the application stops and restarts (for example, in a scheduler application), you would have to re-supply a context when the application restarts and events are loaded. The target method and object could be persistent but the context is not.

Although the callback mechanism is designed with packages in mind, you can also use it to allow a method to be invoked from within the same context.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pTargetOid

The **pTargetOid** parameter is the target oid of the object to which this message is to be sent.

pParamTargetContext

The **pParamTargetContext** parameter is the execution context to which the invoked method switches. After the method has finished, the execution context switches back to the current context.

pTargetMethod

The **pTargetMethod** parameter is the valid method that is executed when the **jomInvokeMethod** API is called.

If the context in the **pTargetMethod** call is **null**, the current context is used. This therefore enables you to invoke a specific saved method rather than calling **jomSendMsg**, which allows you to provide only the name of the method to which the message is sent. For details, see "Sending a Message", earlier in this chapter.

pInParam

The **pInParam** parameter is a pointer that is used to pass a variable list of parameters of any type to the method specified in the **pTargetMethod** parameter. If there are no parameters, a **null** pointer can be passed.

Note If the number or type of the actual parameters passed to a method by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result may occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

For details about the **ParamListType** pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1 of the *JADE Developer's Reference*. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1 of the *JADE Developer's Reference*.

---

pOutParam

The **pOutParam** parameter is a pointer that is used to pass a variable list of parameters of any type from the invoked method back to the sender when the call switches back to the current context.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting an Object

The **jomGetObject** call, shown in the following example, copies the object buffer contents into the area specified in the **pObjectBuffer** parameter.

```
int jomGetObject(const DskHandle   *pHandle,
                 const DskObjectId *pOid,
                 DskParam          *pObjectBuffer,
                 UInt32            lineNo)
```

The format of this buffer is internally determined by the JADE Object Manager.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object whose buffer is to be copied to the **pObjectBuffer** parameter.

pObjectBuffer

The **pObjectBuffer** parameter is updated with the object buffer specified in the **pOid** parameter.

Note This is just a copy of the buffer as it was at that moment. The JADE Object Manager does not have any responsibility for keeping the user informed about possible changes that could make the copy obsolete.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Creating a Shallow Copy of an Object

The **jomCloneObject** call, shown in the following example, is used to create a shallow copy (that is, clone) of the object specified in the **pOid** parameter.

```
int jomCloneObject(const DskHandle   *pHandle,
                   const DskObjectId *pOid,
                   DskBuffer         **ppBuffer,
```

```
                    ClassID         classId,
                    BYTE            type,
                    Boolean         construct,
                    UInt32          lineNo)
```

All common attributes (that is, all properties of primitive types that exist on the source and target object classes) are copied from the source object to the target object.

References are left with a **null** value. Exclusive subobject collections are left empty.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object that is to be cloned (shallow-copied).

ppBuffer

The **ppBuffer** parameter is loaded with a pointer to **DskBuffer** corresponding to the target object.

classId

The **classId** parameter is the oid of the class that is to be created.

type

The **type** parameter specifies the lifetime of the object to be cloned. The types of lifetime that can be specified by using this parameter are listed in the following table.

Type	Description
0	Indicated by the class specified in the **pOid** parameter
1	Transient object
2	Persistent object
3	Shared transient object

Types **1**, **2**, and **3** take precedence over type **0** (specified by the class of the **pOid** parameter.)

construct

The **construct** parameter specifies if the constructor methods, if any, should be called before starting the copy of the common attributes.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Clearing JADE Object Manager Buffers

The **jomClearBuffers** call, shown in the following example, marks the object buffer as obsolete.

```
int jomClearBuffers(const DskHandle   *pHandle,
                    const DskObjectId *pOid,
                    UInt32            lineNo)
```

When an obsolete object buffer is referenced, the JADE Object Manager client module asks the server program to reload it with the contents of a new buffer, if necessary.

This call guarantees that the next time this object is referenced, its latest copy is used.

Note A newly loaded object buffer can be guaranteed to be current only when an object is locked.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

A **pOid** parameter value of zero (**0**) clears all object buffers for the node. A partial oid, a **classId** with a valid class number, and an **instId** value equal to zero (**0**) clear all object buffers for the specified class.

A complete **pOid** parameter clears the specified object only.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Changing the Database Access Mode

The **jomChangeAccessMode** call, shown in the following example, indicates the database access mode for a process, as opposed to the access mode specified in the **jomSignOn** call for the process. (For details about the **jomSignOn** call, see "Opening a Process", earlier in this chapter.)

```
int jomChangeAccessMode(const DskHandle *pHandle,
                        const DskParam  *pMode,
                        const DskParam  *pUsage,
                        UInt32          lineNo)
```

If the requested mode or usage conflicts with the current mode or usage of the database by existing users, the request fails and an exception is raised.

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.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pMode

The **pMode** parameter specifies if the process mode is shared, exclusive, snapshot, or archive.

The mode constants and their integer values are listed in the following table.

Constant	Value	Description
DB_ARCHIVE	4	Quiesces the database for archive backup.
DB_DEFAULT	9	Restores the database to the initial mode and usage values.
DB_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.
DB_SHARED	0	Enables multiple current users to open the database.
DB_SNAPSHOT	6	Conditions the database for external third-party snapshot backup.

For details about external third-party snapshot backups, see "Non-JADE Backups" in the *Developing a Backup Strategy White Paper*.

pUsage

The **pUsage** parameter specifies the database usage. The usage constants and their integer values are listed in the following table.

Constant	Value	Description
DB_UPDATE	0	Enables applications to update the database.
DB_READ_ONLY	1	Specifies that your applications have read-only access to the database.
DB_NO_AUDIT	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.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Beginning a Load

The **jomBeginLoad** call, shown in the following example, indicates the beginning of an atomic read transaction.

```
int jomBeginLoad(const DskHandle  *pHandle,
                 UInt32           lineNo)
```

All unlock operations are ignored within a load state to guarantee that a number of object buffers remain consistent for a period.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Ending a Load

The **jomEndLoad** call, shown in the following example, indicates the end of a read transaction.

```
int jomEndLoad(const DskHandle *pHandle,
               UInt32          lineNo)
```

All transaction locks held by the process are released automatically at this point.

Note As a **jomEndTransaction** or a **jomAbortTransaction** call performs an implicit **jomEndLoad**, a load state cannot be sustained across transactions. (For details about the **jomEndTransaction** and **jomAbortTransaction** calls, see "Ending a Transaction" and "Aborting a Transaction", earlier in this chapter.)

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Inheriting a JADE Object Manager Method

The **jomInheritMethod** call, shown in the following example, is used within method code to cause the execution of a method of the same name in the most-immediate superclass from the class to which the currently executing method belongs.

```
int jomInheritMethod(const DskHandle   *pHandle,
                     const DskObjectId *pOid,
                     DskParam          *pParams,
                     DskParam          *pReturn,
                     UInt32            lineNo)
```

An error occurs if no superclass has a method with the specified name.

This call cannot be used from within a constructor or destructor method.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object whose method code is being executed.

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method or condition. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method or condition back to the sender.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Sending an Event Message

The **jomSendEventMsg** call, shown in the following example, is used to send event messages to objects.

```
int jomSendEventMsg(const DskHandle   *pHandle,
                    const DskObjectId *pOid,
                    DskParam          *pMessage,
                    DskParam          *pParams,
                    DskParam          *pReturn,
                    unsigned int      raiseNoReceiverException,
                    int               noReceiverError,
                    unsigned int      raiseInvalidMessageException,
                    int               invalidMessageError,
                    UInt32            lineNo)
```

This call is similar to the **jomSendMsg** call. (For details, see "Sending a Message", earlier in this chapter.) The difference between the two calls is that if the receiver object class does not have a method corresponding to the event message in this call, no exception is raised by the JADE Object Manager.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object to which this message is to be sent.

pMessage

The **pMessage** parameter is the message name (or class element identifier).

pParams

The **pParams** parameter is a pointer to the first optional parameter. This pointer is used to pass one or more parameters to the method or condition. If there are no parameters, a **null** pointer can be passed.

pReturn

The **pReturn** parameter is a pointer to the second optional parameter. This pointer is used to pass a return value from the invoked method or condition back to the sender.

raiseNoReceiverException

When the **raiseNoReceiverException** parameter is set to **true**, the system will not raise an exception if the receiver object does not exist, but it will pass an error result back to the caller.

noReceiverError

If the receiver object does not exist and the system is to raise an exception, the error number specified in the **noReceiverError** parameter is used instead of the normal error code.

raiseInvalidMessageException

When the **raiseInvalidMessageException** parameter is set to **true**, the system will not raise an exception if the message is invalid, but it will pass an error result back to the caller.

invalidMessageError

If the message is invalid and the system is to raise an exception, the error number specified in the **invalidMessageError** parameter is used instead of the normal error code.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Checking an Object is a Valid Class Instance

The **jomIsKindOf** call, shown in the following example, is used to check if the receiver is a valid instance of the class specified in the **superClassNo** parameter, or any of its subclasses.

```
int jomIsKindOf(const DskHandle   *pHandle,
                const DskObjectId *pOid,
                ClassNumber       superClassNo,
                unsigned int      *pResult,
                UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object that is to be checked to ensure that it is a valid class instance.

superClassNo

The **superClassNo** parameter is the number of the class whose object instance validity is to be checked.

pResult

The **pResult** parameter is a pointer to the result of the class instance validation check.

A value of zero (**0**) indicates that the object is not an instance of the specified class number or any of its subclasses. Any other value indicates that the object is an instance of the specified class number or any of its subclasses.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Checking an Object Implements an Interface

The **jomRespondsTo** call, shown in the following example, is used to check if the receiver implements (responds to) the interface specified in the **interfaceNo** parameter or any interface that extends that interface.

```
int jomRespondsTo(const DskHandle   *pHandle,
                  const DskObjectId *pOid,
                  InterfaceNumber   interfaceNo,
                  unsigned int      *pResult,
                  UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter is the oid of the object that is to be checked to determine if it implements the interface.

interfaceNo

The **interfaceNo** parameter is the number of the interface whose implementation is to be checked (that is, whether the receiver responds to the interface or to any interface that extends the interface).

pResult

The **pResult** parameter is a pointer to the result of the interface implementation check.

A value of zero (**0**) indicates that the object does not implement the specified interface number. Any other value indicates that the object implements the specified interface or any interface that extends it.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Beginning a Notification

The **jomBeginNotification** call, shown in the following example, requests the JADE Object Manager server to send a notification message whenever the specified object is affected by the specified event.

```
int jomBeginNotification(const DskHandle   *pHandle,
                         DskParam          *pSubscriber,
                         DskParam          *pFeature,
                         const DskObjectId *pTarget,
                         const UInt32      eventType,
                         UInt32            responseType,
                         UInt32            subscriberResponseType,
                         UInt32            userTag,
                         UInt32            lineNo)
```

The notification is sent at the end transaction point for JADE Object Manager events and instantly for USER events if the **immediate** parameter of the **jomCauseEvent** call is set to a non-zero value. (For details about the **jomCauseEvent** or **jomCauseSDSEvent** call, see "Causing an Event" or "Causing an SDS Server Event", later in this chapter.)

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pSubscriber

The **pSubscriber** parameter specifies a pointer to the **DskParam** of the subscribing object. The **DskParam** can be of type **DSKOBJECTID** for the default object notification mechanism or of type **DSKWINPARAM** to invoke the Windows-based delivery mode.

Note The **DSKWINPARAM** type provides compatibility with previous notification delivery mechanisms.

pFeature

Set the **pFeature** parameter to **NULL**, as it is for internal system use only.

pTarget

The **pTarget** parameter specifies a pointer to the object id for which notification is required. All fields except the **edition** field are required.

Setting either or both the **classId** and **instId** fields to zero (**0**) causes a masking effect. For example, setting **instId** to zero (**0**) and specifying a **classId** results in any instance of that class returning a notification when it receives a matching event.

Setting the **classId** field to zero (**0**) (the **instId** field must also be zero (**0**)) results in any instance of any class returning a notification when it receives a matching event.

eventType

The **eventType** parameter specifies the event that invokes a notification response. These are divided into two categories: **JADE** and **USER** events. The **JADE** event constant definitions are listed in the following table.

Constant	Value	Description
JOM_UPDATE_EVENT	3	Update event
JOM_CREATE_EVENT	4	Create event
JOM_DELETE_EVENT	6	Delete event
JOM_EVENT	0	Update, create, or delete event

USER-defined events can be in the range 16 through 4 G bytes, where 4G bytes represents any **USER** event.

responseType

The **responseType** parameter values are listed in the following table.

Value	Meaning	Sends a notification …
0	Continuous	Whenever the object receives a matching event.
1	Cancel	When the object receives a matching event and then cancels the notification request.
2	Suspend	When the object receives a matching event and then suspends notifications until the registering process acts upon the notification object by resynchronizing its cache copy with that of the server.

subscriberResponseType

The **subscriberResponseType** parameter values are listed in the following table.

Value	Sends …
0	A response to the subscriber for each notification in a transaction (the default).
1	Only one response (the first object notification) to the subscriber for each transaction.

userTag

The **userTag** parameter can contain any user information and is returned in the Windows **lParam** field.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Ending a Notification

The **jomEndNotification** call, shown in the following example, requests the JADE Object Manager server to cancel a previous begin notification request for the specified object.

```
int jomEndNotification(const DskHandle   *pHandle,
                       const DskObjectId *pTarget,
                       UInt32            eventType,
                       DskParam          *pSubscriber,
                       DskParam          *pFeature,
                       UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pTarget

The **pTarget** parameter specifies a pointer to the object id for which the end notification is required. The value specified in this parameter must match that of the **pTarget** parameter of the **jomBeginNotification** call. (For details, see "Beginning a Notification", earlier in this chapter.)

eventType

The **eventType** parameter specifies the event specified on the matching begin notification request.

pSubscriber

The **pSubscriber** parameter specifies the subscriber to the notification, which was supplied by the **jomBeginNotification** call.

pFeature

Set the **pFeature** parameter to **NULL**, as it is for internal system use only.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Causing an Event

The **jomCauseEvent** call, shown in the following example, requests the JADE Object Manager server to cause a USER notification event on all matching USER notification requests.

```
int jomCauseEvent(const DskHandle   *pHandle,
                  const DskObjectId *pTarget,
                  UInt32            eventType,
                  UInt16            immediate,
                  DskParam          *pInfo,
                  DskObjectId       *pCausedBy
                  UInt32            lineNo)
```

This call can be made from an execution thread that does not correspond to a process of the current node. For threads that do not correspond to a process of the current node, the request is executed by the background process thread.

Events are processed at the time of execution, and the notifications are sent immediately or at the end of transaction time to those users who have a matching subscriber.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pTarget

The **pTarget** parameter specifies a pointer to the object id for which the notification event is required. The **jomBeginNotification** request call masking rules apply to this call. (For details, see "Beginning a Notification", earlier in this chapter.)

eventType

The **eventType** parameter specifies the user event that is to cause the notification response.

immediate

The **immediate** parameter is used to determine when the event is actioned. Set this parameter to zero (**0**) to post the event at the end of transaction.

If it is not set to zero (**0**), the event is posted immediately.

pInfo

The **pInfo** parameter specifies a pointer to the **DskParam** that can contain the user information to be associated with the notification. It is also delivered to all subscribers of the notification.

pCausedBy

The **pCausedBy** parameter is intended for future use, to specify an optional object id indicating the object that is to be associated with the event causation.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Causing an SDE Event

The **jomCauseSDEEvent** call, shown in the following example, is used in a Synchronized Database Environment (SDE) to achieve inter-system event notifications by allowing an event to be caused on the local system as well as on the reciprocal SDS system.

```
int jomCauseSDEEvent(const DskHandle   *pHandle,
                     const DskObjectId *pTarget,
                     UInt32            eventType,
                     UInt16            immediate,
                     DskParam          *pInfo,
                     DskObjectId       *pCausedBy
                     UInt32            lineNo)
```

The **jomCauseSDEEvent** call combines the **jomCauseEvent** and **jomCauseSDSEvent** calls. For details, see "Causing an Event" and "Causing an SDS Server Event", elsewhere in this chapter.

When used on a SDS primary system, subscribers of the event are notified on the primary server as well as the attached secondary database servers. On an SDS secondary system, subscribers are notified on the secondary system as well as on the primary system. On a primary system, subscribers are notified only on the secondary systems if the target object is persistent, the value of the **immediate** parameter is **false**, and the process is currently in transaction state.

On a secondary database server, the event is caused only on the primary database server if the target object is persistent. The value of the **immediate** parameter is immaterial. However, subscribers of the event are always notified immediately on the primary, even when the value of the **immediate** parameter is **false**, to defer the event on the secondary database server.

The following table lists the contexts in which the event is caused.

Database Role	Transient Target Object	Persistent Target Object, Immediate	Persistent Target Object, Deferred, in Transaction State	Persistent Target Object, Deferred, not in Transaction State
Undefined	Process only	Local system	Local system	Local system
Primary	Process only	Primary system	Primary system and secondary systems	Primary system only
Secondary	Process only	Primary system and secondary system	Secondary system and immediately on the primary system	Secondary system and immediately on the primary system

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pTarget

The **pTarget** parameter specifies a pointer to the object id for which the notification event is required. The **jomBeginNotification** request call masking rules apply to this call. (For details, see "Beginning a Notification", earlier in this chapter.) If this parameter specifies a transient object, no events are posted.

eventType

The **eventType** parameter specifies the user event that is to cause the notification response.

immediate

The **immediate** parameter is used to determine when the event is actioned. Set this parameter to zero (**0**) on a primary system. As events caused on a secondary are assumed to be immediate, this parameter is ignored.

pInfo

The **pInfo** parameter specifies a pointer to the **DskParam** that can contain user information to be associated with the event. It is delivered to subscribers of the notification in the corresponding parameter of the user event notification callback.

pCausedBy

The **pCausedBy** parameter is intended for future use, to specify an optional object id indicating the object that is to be associated with the event causation.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Causing an SDS Server Event

The **jomCauseSDSEvent** call, shown in the following example, is used in a Synchronized Database Service (SDS) to achieve inter-system event notifications.

```
int jomCauseSDSEvent(const DskHandle   *pHandle,
                     const DskObjectId *pTarget,
                     UInt32            eventType,
                     UInt16            immediate,
                     DskParam          *pInfo,
                     DskObjectId       *pCausedBy
                     UInt32            lineNo)
```

The role-dependent usage scenarios are as follows.

- From a primary system, to cause persistent events audited by the primary database for replay by secondary database servers.
- From a secondary system, to cause events that are notified to event subscribers on the primary system.

The behavior of the **jomCauseSDSEvent** function is database role-dependent. The three database role categories are listed in the following table.

Role	Action
Primary	When called from a thread executing in the primary system, the **USER** event is audited by the primary for subsequent replay by SDS secondary databases. The event is not notified to event subscribers in the primary system. When using **jomCauseSDSEvent** in a primary system, **USER** notification events are only posted if the target is a persistent object and the request is not specified as being immediate. In other words, the request has no effect if the target is a transient object or the request is specified to be immediate.
Secondary	When called from a thread executing in the secondary system, if the secondary system is connected to its primary server, a corresponding **USER** event on the same receiver object is triggered in the primary system. The user event is not notified on the secondary system. When using **jomCauseSDSEvent** in a secondary system, **USER** notification events are only posted if the target is a persistent object. In other words, the request has no effect if the target is a transient object.
None	When invoked within a non-SDS-capable system, the behavior is the same as the **Object**::**causeEvent** method.

This call can be made from an executing thread that does not correspond to a process of the current node. For threads that do not correspond to a process of the current node, the request is executed by the background process thread.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pTarget

The **pTarget** parameter specifies a pointer to the object id for which the notification event is required. The **jomBeginNotification** request call masking rules apply to this call. (For details, see "Beginning a Notification", earlier in this chapter.) If this parameter specifies a transient object, no events are posted.

eventType

The **eventType** parameter specifies the user event that is to cause the notification response.

immediate

The **immediate** parameter is used to determine when the event is actioned. Set this parameter to zero (**0**) on a primary system. As events caused on a secondary are assumed to be immediate, this parameter is ignored.

pInfo

The **pInfo** parameter specifies a pointer to the **DskParam** that can contain user information to be associated with the event. It is delivered to subscribers of the notification in the corresponding parameter of the user event notification callback.

pCausedBy

The **pCausedBy** parameter is intended for future use, to specify an optional object id indicating the object that is to be associated with the event causation.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting Notification Information

The **jomGetNotificationInfo** call, shown in the following example, requests the JADE Object Manager client to return the additional notification information for the specified Windows handle and a count of the number of currently queued messages for this handle.

```
int jomGetNotificationInfo(const DskHandle  *pHandle,
                           DskObjectId      *pSubscriber,
                           NotificationInfo *pNoteInfo,
                           int              *pNotesLeft,
                           UInt32           lineNo)
```

The return value from this call can be zero (**0**) whenever a message is returned or a **-1** (negative one) value when there are no messages. The format of this message is:

```
NotifyHdr   header;
DskParam    data;
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pSubscriber

The **pSubscriber** parameter specifies the subscriber of the notification, supplied by the **jomBeginNotification** call. (For details, see "Beginning a Notification", earlier in this chapter.)

pNoteInfo

The **pNoteInfo** parameter contains the **NotifyHdr** values. The **NotifyHdr** values that are returned are:

```
UInt32        eventType;
UInt32        notifyParam;
UInt32        userTag;
UInt32        serialNo;
DskObjectId   target;
DskObjectId   subscriber;
DskHandle     hProcess;
UInt16        length;
```

These values are described in the following subsections.

eventType

The **eventType** value is the event type that caused the notification event.

notifyParam

The **notifyParam** value is the Windows **wParam** value that accompanied the notification event. It consists of the event type with the left-most bit set.

userTag

The **userTag** value is your user information supplied with the **jomBeginNotification** request. (For details, see "Beginning a Notification", earlier in this chapter.)

serialNo

The **serialNo** value is a sequential number supplied by the server to each **jomBeginNotification** request. This number is initialized to zero (**0**) with the corresponding **jomBeginNotification** request and is incremented after each message is posted, to enable you to detect if messages have been lost due to an overflow of the message queue.

target

The **target** value is the oid supplied with the **jomBeginNotification** request.

subscriber

The **subscriber** value is the oid supplied with the **jomBeginNotification** request.

hProcess

The **hProcess** value is an internal value.

length

The **length** value is the length of the data supplied with this message.

pNotesLeft

The **pNotesLeft** parameter contains a count of the remaining queued notification information messages.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Checking the Dynamic Link Library Version

The **include** directory on the JADE release medium contains the header file **jombuild.h**, which provides the defines that enable you to perform build and runtime version checking.

The define **MAKEMODULEVERSIONFUNC** is used to construct a module version-specific function name.

Each JADE Dynamic Link Library (DLL) exports a function named using **MAKEMODULEVERSIONFUNC**; for example, the function name exported by **jomutil.dll** is **"moduleVersion_jomutil_7_1_02"**.

Importers of the JADE DLL can use the following to ensure that the importing executable, the **jomutil.lib**, and the **jomutil.dll** are consistent.

Declaration:

```
extern "C" DllImport int JOMAPI MAKEMODULEVERSIONFUNC(jomutil)();
```

Invocation:

```
int jomutilpatch = MAKEMODULEVERSIONFUNC(jomutil)();
```

If the **jombuild.h** file version included by the user source does not match the **jomutil.lib**, the compile-time linker raises an error; for example:

```
mysource.obj : error LNK2019: unresolved external symbol __imp_moduleVersion_
jomutil_7_0_7 referenced in function MyVersionChecker
```

At run time, if the importer version and the **jomutil.lib** version do not match **jomutil.dll**, the Dynamic Linker raises the following.

```
The procedure entry point moduleversion_jomutil_7_0_8 could not be located in the
dynamic library jomutil.dll.
```

The importing executable should contain a version-check function invocation and declaration for each statically-linked JADE DLL; for example, **jomutil.dll** and **jom.dll**.

Exception Handling

The JADE Object Manager calls for exception handling are described in the following subsections.

Arming an Exception Handler

The **jomArmExceptionHandler** call, shown in the following example, requests the JADE Object Manager kernel that if an exception of the specified exception class occurs before the handler is disarmed, it should then send the message in the parameter to the receiver object.

```
int jomArmExceptionHandler(const DskHandle   *pHandle,
                           const DskObjectId *pOid,
                           const DskObjectId *pExceptionClass,
                           DskParam          *pMessage,
                           DskParam          *pInputParams,
                           BYTE              scope,
                           UInt32            lineNo)
```

Notes As the default exception handler is not invoked if you intercept an exception with your own handler, the Unhandled Exception dialog is not displayed, regardless of the value returned for your exception handler. If you want to invoke the default handler and therefore display the Unhandled Exception dialog, you must send the message **"defaultHandler"** to the exception object.

For more details about arming an exception handler, see Chapter 3 of the *JADE Developer's Reference*.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pOid

The **pOid** parameter specifies the object to which the message specified in the **pMessage** parameter is sent when the exception occurs. (As the receiver must be an object, you cannot arm the exception handler when the receiver is a primitive type.)

pExceptionClass

The **pExceptionClass** parameter specifies the exception class that is to be handled; for example, the **FileException** class handles all file exceptions and the **Exception** class handles all exceptions (including those in subclasses). For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

pMessage

The **pMessage** parameter specifies the exception handler method to be invoked when an exception of a specified exception class is raised.

The **exceptionHandler** call that corresponds to the exception message must have the following syntax.

```
int exceptionHandler(DskBuffer *pReceiverBuffer,
                     DskParam  *pExceptionParam,
                     DskParam  *pResultParam)
```

This method is executed to handle the exception, by accessing the transient exception instance using the reference provided in the **pExceptionParam** parameter.

pResultParam

The method should set the **pResultParam** parameter, to indicate the action to be followed by the method that caused the exception. The actions are:

- Continue execution
- Resume execution
- Abort execution
- Execute the next available exception handler

The **continue** execution action attempts to continue the execution (in the next instruction) of the method that caused the exception. The **resume** execution action attempts to resume the execution (in the next instruction) of the method that caused the exception or of the method that armed the exception handler.

The **abort** execution action aborts all methods currently executing in the kernel stack; that is, all methods invoked by the JADE Object Manager kernel. The outermost caller receives the **JOM_METHOD_ABORTED** result to the last call to the JADE Object Manager that caused the error.

The exception handler can also indicate that the next available handler for this class of exception should be executed, leaving the decision of resuming or aborting the method that caused the exception to that handler.

If the last global exception handler indicates the execution of the next available exception handler, the error code is returned as the result of the JADE Object Manager call that caused the exception.

The **pResultParam** parameter values are listed in the following table.

Value	Constant	Action
0	E_CONTINUE	Processing continues as if no error had occurred. The JADE Object Manager API call that caused the exception ultimately returns a zero (**0**) result to the user application. **Caution** Use this return mode only in circumstances when you are certain that continuing the code execution will still be correct after ignoring the exception. For lock exceptions, use this return mode only if the lock has been successfully retried. If the lock exception occurred while updating, ensure that the transaction has not been aborted by the exception handler before using **E_CONTINUE**.
1	E_ABORT_ACTION	Aborts the current action. The JADE Object Manager message stack is cut back, and control returns to the user application, with the JADE Object Manager API call that caused the exception returning a **JOM_METHOD_ABORTED** result.
2	E_RESUME_NEXT	The JADE Object Manager message stack is cut back to the method that armed the exception handler, and the JADE Object Manager API call that caused the exception returns a zero (**0**) result. If there were no JADE Object Manager messages on the stack when the handler was armed, the effect of the **E_RESUME_NEXT** is identical to that of **E_ABORT_ACTION**.
-1	E_PASS_BACK	Passes control back to any previously armed exception handler for this type of exception. If there are no other applicable handlers, the JADE Object Manager message stack is cut back, and control returns to the user application, with the JADE Object Manager API call that caused the exception returning a non zero result representing the error condition.

Your external methods should make provision for an abort action. The JADE Object Manager kernel cuts the C++ stack, by issuing a *throw* statement. Your resource clean up code should be given guaranteed execution, by means of destructors or by catching all C++ exceptions and *rethrowing* them on clean up completion.

Exceptions can occur automatically because of error conditions found by the kernel or they can be created manually and passed to the **jomRaiseException** entry point. (For details, see "Raising an Exception", later in this chapter.) It is the responsibility of the exception handler to abort the current transaction, if required.

pInputParams

The **pInputParams** parameter is a placeholder intended for future use, and should be set to zero (**0**).

scope

The **scope** parameter values are listed in the following table.

Constant	Value	Action
ES_LOCAL	0	Local exception automatically disarmed when current method execution finishes
ES_GLOBAL	1	Global exception armed for the duration of the session

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Disarming an Exception

The **jomDisarmExceptionHandler** call, shown in the following example, requests the JADE Object Manager kernel to disarm the most-recent armed exception handler for exceptions of the specified class and within the specified scope.

```
int jomDisarmExceptionHandler(const DskHandle   *pHandle,
                              const DskObjectId *pExceptionClass,
                              BYTE              scope,
                              UInt32            lineNo)
```

Note All non-global exception handlers are disarmed when the method within which they were armed finishes.

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pExceptionClass

The **pExceptionClass** parameter specifies the class whose exceptions are to be disarmed.

scope

The **scope** parameter values are listed in the following table.

Define	Value	Action
ES_LOCAL	0	Disarm the local exception handler
ES_GLOBAL	1	Disarm the global exception handler

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting Text for a System or Error Message

The **jomGetMessageText** call, shown in the following example, is an architecture-independent call that returns a character string of the text of a system or error message from the **jadmsgs** file.

```
int jomGetMessageText(int       num,
                      Character *msgString,
                      Size      length)
```

The parameters for this call are described in the following subsections.

num

The **num** parameter specifies the error number whose text you want returned.

msgString

The **msgString** parameter is a buffer for the text that you want returned.

length

The **length** parameter specifies the size of the text buffer. If the value specified in the **msgString** parameter is 50 characters and you specify a **length** of 30, only thirty characters of text are returned.

Raising an Exception

The **jomRaiseException** call, shown in the following example, requests the JADE Object Manager kernel to execute the exception handling selection algorithm with the exception instance referenced in the **pException** parameter.

```
int jomRaiseException(const DskHandle   *pHandle,
                      const DskObjectId *pException,
                      BYTE              cause,
                      UInt32            lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pException

The **pException** parameter specifies the exception that is to be handled.

The exception handling exception algorithm looks for an exception handler that can handle this exception instance. The search starts at the current method and proceeds with all the methods in the stack until it reaches the outermost method.

If no method level handler is found, the search continues with the existing global exception handlers. If no handler is found, the **defaultHandler** message is sent to the exception instance.

When the handler execution finishes, the exception instance is automatically deleted by the kernel. Execution of nested exception handlers is allowed, as long as the new exception error code is different from that of the exception that is already being processed.

cause

The cause parameter specifies the method against which the exception is to be reported. The constant definitions for the cause of the event are listed in the following table.

Constant	Value	Description
EC_PRECONDITION	0	Reported against the method that called the current method
EC_INTERNAL	1	Reported against the current method itself

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting System Status

The **jomGetSystemStatus** call, shown in the following example, requests the state of a process.

```
int jomGetSystemStatus(const DskHandle *pHandle,
                       unsigned int    *pInTransactionState,
                       unsigned int    *pInTransientTransactionState,
                       unsigned int    *pInLoadState,
                       unsigned int    *pInLockState,
                       unsigned int    *pInExceptionState,
                       unsigned int    *pUnicodeVersion,
                       DskParam        *pCurrentMth,
                       UInt32          lineNo)
```

The current system status information is returned in parameters described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pInTransactionState

The **pInTransactionState** parameter specifies that the system is in persistent transaction state.

pInTransientTransactionState

The **pInTransientTransactionState** parameter specifies that the system is in transient transaction state.

pInLoadState

The **pInLoadState** parameter specifies that the system is in load state.

pInLockState

The **pInLockState** parameter specifies that the system is in lock state.

pInExceptionState

The **pInExceptionState** parameter specifies that the system is executing an exception handler method or the epilog of a method is being executed while cutting back the stack as the result of an exception handler exit code.

pUnicodeVersion

The **pUnicodeVersion** parameter specifies that the current kernel and libraries version is Unicode.

pCurrentMth

The **pCurrentMth** parameter contains the name of the method that is currently executing.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Getting Execution Instances

The **jomGetExecInstances** call, shown in the following example, returns the object ids of the environmental objects associated with the process.

```
int jomGetExecInstances(const DskHandle *pHandle,
                        DskObjectId     *pGlobal,
                        DskObjectId     *pApp,
                        DskObjectId     *pRootSchema,
                        DskObjectId     *pSchema,
                        DskObjectId     *pSystem,
                        DskObjectId     *pNode,
                        DskObjectId     *pProcess,
                        DskObjectId     *pSession,
                        UInt32          lineNo)
```

The parameters for this call are described in the following subsections.

pHandle

The **pHandle** parameter is maintained for backward compatibility and is ignored. It should be set to **null**.

pGlobal

The pGlobal parameter specifies the global instance of the process.

pApp

The **pApp** parameter specifies the application transient instance of the process.

pRootSchema

The **pRootSchema** parameter specifies the instance of the Root Schema.

pSchema

The **pSchema** parameter specifies the instance of the current schema; that is, the schema specified in the **jomSignOn** call for the process. (For details about the **jomSignOn** call, see "Opening a Process", earlier in this chapter.)

pSystem

The **pSystem** parameter specifies the system environmental object instance.

pNode

The **pNode** parameter specifies the instance corresponding to the node where the current process is executing.

pProcess

The **pProcess** parameter specifies the instance corresponding to the current process.

pSession

The **pSession** parameter specifies the transient instance that corresponds to the current session.

lineNo

If an exception directly caused by the call occurs, the line number that is reported is that passed in the **lineNo** parameter.

Chapter 4 JADE System Instrumentation and Diagnosis

This chapter covers the following topics.

Overview

The JADE Object Manager enables you to:

- Use system instrumentation and diagnosis methods in your JADE applications to examine your JADE system, nodes, or processes. These methods, which are summarized in the following subsection, are used by the JADE Monitor and when node sampling.

  For details about using the JADE Monitor to monitor and examine your JADE environment, see the *JADE Monitor User's Guide*. For details about using the Web services JADE Monitor, see "Monitoring Your Web Sessions", in Chapter 2 of the *JADE Web Application Guide*.

- Sample JADE node activities and gather extensive details and statistics in addition to those provided by the statistics windows in the JADE Monitor or by calling the **System** class **getStatistics** or **getStatistics64** method. For details, see "Node Sampling", later in this chapter.

  Tip You can use the JADE Monitor to analyze the contents of your node sampling files.

Returned method values include cumulative counters that do not get reset during the lifetime of the node. To work out the value differences, JADE applications that use a method that retrieves statistical information need to compare values from one call to the next.

Cumulative values are held as unsigned 64-bit Integer values, and are copied to the dynamic object as **Integer64** values. Although unlikely, you should be aware that the cumulative values can overflow to negative values. The maximum value before cumulative values overflow to a negative value is **2^63 – 1** (approximately 8 Exabytes).

Implementing System Instrumentation and Diagnosis Methods

In addition to the node sampling methods documented under "Node Sampling", later in this chapter, JADE provides an Application Programming Interface (API) framework whose methods, summarized in the following table, you can incorporate into your JADE applications to examine your JADE system, nodes, or processes.

Class	Method	Description
DbFile	getFileStatus	Returns the status of a physical database file during the backup process
DbFile	getOpenPartitions	Populates the input **partitionList** array with references to **JadeDbFilePartition** instances; one for each open partition of the associated database file
DbFile	getPartitions	Populates the input **partitionList** array with **JadeDbFilePartition** instances; one for each partition of the associated database file
DbFile	isPartitioned	Returns **true** if the database file is partitioned
DbFile	getStatistics	Returns statistics on all reads and writes to the database file
JadeDatabaseAdmin	getDbFiles	Populates an array with references to files of selected kinds
JadeDbFilePartition	getFileStatus	Returns the status of a single physical database file partition during the backup process
JadeDbFilePartition	getStatistics	Returns statistics on reads of single database partition activity
JadeDbFilePartition	isFrozen	Returns **true** if the associated partition is frozen
JadeDbFilePartition	isOffline	Returns **true** if the associated partition is offline
Node	getCacheSizes	Retrieves the cache sizes of the node on which the method is executing
Node	getCacheSizes64	Retrieves the cache sizes of the node on which the method is executing when running in a 64-bit JADE environment
Node	getLocks	Populates the specified array with transient instances of the current locks for shared transient instances
Node	getNotes	Reserved for future use
Node	getMutexCounts	Retrieves the number of contentions on mutexes (locking mechanism used to ensure thread safety when executing critical sections of code) used internally by JADE for the node
Node	getObjectCaches	Retrieves node sampling values relating to cache activity

Class	Method	Description
Node	getQueuedLocks	Populates the specified array with transient instances of the lock requests waiting for shared transient objects to be unlocked
Node	getRequestStats	Returns node statistics relating to persistent database requests from the receiver
Node	getRpcServerStatistics	Retrieves statistics relating to RPC activity between the database server node and the receiver node
Node	processDump	Invokes a non-fatal operating system process dump of a specific node
Node	setCacheSizes	Sets the cache size values on the node on which the method is executing and retrieves the current values after the operation
Node	setCacheSizes64	Sets the cache size values on the node on which the method is executing and retrieves the current values after the operation when running in a 64-bit JADE and environment
Node	wbemListClasses	Retrieves a list of the WBEM classes that can be queried for the node of the receiver object
Node	wbemListInstanceNames	Retrieves the names of all instances of a specified WBEM class for the node of the receiver object
Node	wbemQueryQualifiers	Retrieves the name, type, and scale factor for each property of a specified WBEM class
Node	wbemRetrieveData	Retrieves instances and property values for a specified WBEM class for the node of the receiver object
Process	adjustObjectCachePriority	Changes how long an object is to be retained in persistent or transient object cache
Process	beginMethodProfiling	Starts method profiling for the receiving process
Process	classAccessFrequenciesStatus	Returns the status of class access counters, including the UTC time
Process	enableClassAccessFrequencies	Enables or disables class access counters
Process	endMethodProfiling	Stops method profiling for the receiving process
Process	extractRequestStatistics	Extracts local or remote request statistics from a notification sent in response to a **sendRequestStatistics** method request
Process	extractWebStatistics	Extracts performance statistics relating to Web activity from a notification sent in response to a **sendWebStatistics** method request
Process	getBufferStatistics	Returns cache-related information about a specified object
Process	getCallStackInfo	Retrieves information about the call stack of the executing process

Class	Method	Description
Process	getMethodProfileInfo	Retrieves method profiling information for the receiving process
Process	getRpcServerStatistics	Retrieves statistics relating to RPC activity between the database server node and the receiver process
Process	getTimers	Retrieves timer-related information
Process	profileMethod	Selects or deselects a method to be profiled for the receiving process
Process	removeMethodProfileInfo	Removes all method profiling information for the receiving process
Process	sendCallStackInfo	Requests a target process to send one or more notifications containing information about the call stack of the receiver process
Process	sendRequestStatistics	Requests a process to send a notification containing local or remote request statistics
Process	sendTransientFileAnalysis	Requests a target process to send one or more notifications containing detailed analysis of the transient database file
Process	sendTransientFileInfo	Requests a target process to send a notification containing information about its transient database file
Process	sendWebStatistics	Requests a process to send a notification containing performance statistics for Web activity
Process	setObjectCachePriority	Specifies how long an object is to be retained in persistent or transient object cache
System	beginLockContentionStats	Starts recording lock contentions for persistent objects
System	beginObjectTracking	Starts recording in a file database server persistent object activity
System	clearLockContentionStats	Removes all existing lock contention data and restarts recording of lock contentions
System	endLockContentionStats	Stops recording lock contentions and removes all lock contention data
System	endObjectTracking	Ends recording in a file of database server persistent object activity
System	getClassAccessFrequencies	Returns access counts for specified classes
System	getDatabaseStats	Returns statistics relating to persistent database activity
System	getDbDiskCacheStats	Returns statistics relating to the persistent database disk cache
System	getLockContentionInfo	Retrieves lock contention information for the specified object
System	getLockContentionStats	Retrieves lock contention for all contended locks

Class	Method	Description
System	getLocks	Fills an array with a specified number of instances of current locks in the system
System	getMostAccessedClasses	Returns access counts for the classes that have been most frequently accessed
System	getNotes	Fills an array with a specified number of instances of current notification requests in the system
System	getQueuedLocks	Fills an array with the specified number of instances of lock requests in the system that are waiting for a locked object
System	getRequestStats	Returns system statistics relating to requests carried out by the database server node
System	getRpcServerStatistics	Retrieves RPC statistics relating to activity between the database server node and all client nodes
System	processDumpAllNodes	Invokes a near-simultaneous operating system process dump of all nodes and the server node itself
System	queryLockContentionStats	Retrieves information about the current recording of lock contentions

For details, see the following subsections. For examples of the use of and output from these methods, see Volume 1 and Volume 2 of the *JADE Encyclopaedia of Classes*.

Adjusting the Retention Priority of Objects in Cache

A set of methods defined for the **Process** class enables you to adjust and interrogate the retention characteristics of an object in cache. The retention characteristics affect the relative priority of cached objects in respect to how long they are retained in cache.

When the object cache is full and a new object is to be copied into cache, one or more objects must be removed from the cache to make enough room. By default, the least-recently referenced object is selected for removal from cache.

The methods in the following subsections enable you to adjust this policy for selected objects in cache, by using a concept of *lives*. By default, an object has a single life. If it is selected to be removed from cache because it is the object that was least-recently accessed, it is removed because it has used its one life.

The **Process** class **adjustObjectCachePriority** and **setObjectCachePriority** methods allow the number of lives an object in cache has to be altered. When an object is a candidate to be removed from cache but its number of lives is greater than **1**, the object is not removed from cache; instead its number of lives is decremented and it is treated as if it had just been referenced. It therefore remains in cache for an additional period.

You can therefore use the **adjustObjectCachePriority** and **setObjectCachePriority** methods to increase the number of lives of an object in cache, causing it to be retained longer in the object cache. This can suit objects that are repeatedly accessed but potentially have time gaps between accesses.

You can also use these methods to lower the number of lives for an object in cache; in particular, if you set the number of lives to zero (**0**), the object is removed from cache immediately. This suits objects that are accessed once only.

After being accessed, you can use the **setObjectCachePriority** method to set the lives of an object to zero (**0**) so that it is immediately removed from cache. This prevents objects that are not likely to be accessed again from filling the cache at the expense of other more-regularly accessed objects.

Note The number of lives an object has applies only while the object is in cache. When an object is first loaded into cache, it is always assigned one life. Lives are not recorded for objects that are not in the cache.

Although String Large Objects (slobs) and Binary Large Objects (blobs) are removed when the cache priority of an object is set to zero (**0**), exclusive collections are not removed and you must remove these individually, as shown in the following examples.

```
// remove obj1 from cache as well as its slobs and blobs, but not its
// exclusive collections
process.setObjectCachePriority(obj1, 0);

// remove the exclusive collection obj2.allObj3s from cache and any of
// its collection blocks
process.setObjectCachePriority(obj2.allObj3s, 0);
```

A process must use its own **Process** instance as the method receiver. Using any other **Process** instance raises exception 1265 (*Environmental object operation is out of scope for process*).

For details, see the following subsections.

Process::*setObjectCachePriority* Method

Signature `setObjectCachePriority(obj: Object; priority: Integer);`

The **Process** class **setObjectCachePriority** method alters how long a specified object is to be retained in persistent or transient object cache. The greater the value of the **priority** parameter, the longer an object remains in the object cache.

Use the **priority** parameter to give an object a number of *lives* in cache. The range of values for the **priority** parameter is zero (**0**) through **255**. A negative value is treated as zero (**0**) and a value greater than **255** is treated as **255**.

If the value of the **priority** parameter is greater than zero (**0**), the **setObjectCachePriority** method loads the object into cache if it is not already present, before setting the number of lives. For values greater than **1**, this results in an extension to the length of time the object is retained in cache. If the value of the **priority** parameter is zero (**0**), the **setObjectCachePriority** method removes the object and its subobjects (that is, slobs, blobs, and collection blocks) from cache immediately, if it is currently in cache. If it is not in cache, the method has no effect.

Note An object is not removed from the cache if it is currently being updated by another process (that is, it contains uncommitted updates).

You can use the **setObjectCachePriority** method with transient objects as well as persistent objects.

With transient objects, a process can affect only shared transient objects and its own non-shared transient objects.

Process::*adjustObjectCachePriority* Method

Signature `adjustObjectCachePriority(obj: Object; delta: Integer): Boolean;`

The **Process** class **adjustObjectCachePriority** method changes how long a specified object is to be retained in object cache.

If the object is in cache, this method returns **true**, to indicate that the retention of the object in cache has been changed. If the object is not in cache or it is being updated by another process, this method returns **false**.

Use the **delta** parameter to change the number of *lives* in cache of the object specified by the **obj** parameter. The number of lives for an object in cache is **1** through **255**. A positive value for the **delta** parameter increases the number of lives up to the upper limit of **255**. A negative value decreases the number of lives to the lower limit of zero (**0**) lives, at which point the object and its subobjects (that is, slobs, blobs, and collection blocks) are removed from cache.

The **adjustObjectCachePriority** method is a variation of the **setObjectCachePriority** method, which enables you to adjust the number of lives relative to the current value, rather than specifying the exact number of lives.

You can use the **adjustObjectCachePriority** method with transient objects as well as persistent objects. With transient objects, a process can affect only shared transient objects and its own non-shared transient objects.

Process::*getBufferStatistics* Method

Signature
```
getBufferStatistics(obj: Object;
                    jdo: JadeDynamicObject): Boolean;
```

The **Process** class **getBufferStatistics** method returns cache-related information about the object specified by the **obj** parameter.

The cache information is returned as properties inserted into the **JadeDynamicObject** instance specified by the **jdo** parameter.

The calling process is responsible for creating and deleting this instance. Any existing properties in the **JadeDynamicObject** instance are cleared when the **getBufferStatistics** method is called.

The method returns **true** if the object was in cache at the time of the call, and **false** if the object was not in cache. It does not load an object into cache if it was not already present.

The properties inserted into the **JadeDynamicObject** instance are listed in the following table.

Property	Type	Description
object	Object	A reference to the specified object.
size	Integer	The size of the object in bytes.
lives	Integer	The current number of lives for the object; that is, its cache retention priority value.
cycles	Integer	The number of times the object has been retained instead of removed from cache due to the number of lives being greater than 1.
flag	Integer	The status of the object cache, which can be one of the following values.
		0 Empty; that is, contains no data.
		1 Active; that is, contains data.
		2 Temporary; that is, an empty exclusive collection.
		3 Updated; that is, contains uncommitted updates.
		4 Created; that is, a newly created object that has not yet been committed.
		5 Permanent; that is, a newly created exclusive collection that has not yet been committed.
		6 Deleted; that is, an object that has been deleted but the deletion has not yet been committed.
		7 Not applicable.
		8 Being read; that is, the object is currently being loaded into cache.

Property	Type	Description
		9 Dirty; that is, contains updates done by another process that have not yet been committed.
		10 Obsolete; that is, contains obsolete data that will not be used. When next accessed, the data is refreshed by reloading the object.
operations	Integer64	The number of times the object has been accessed since being loaded into the cache.
age	Integer64	The number of node ticks that have elapsed since the object was loaded into cache. (Node ticks record the total number of accesses for all objects in all caches.)

You can use the **getBufferStatistics** method with persistent and transient objects. With transient objects, a process can examine only shared transient objects and its own non-shared transient objects. For an example of this method, see the **getBufferStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Cache Information

The **Node** class provides methods that enable you to obtain and set cache information. For details, see the following subsections.

Node::*getCacheSizes* Method

Signature

```
getCacheSizes(persistentCache:      Integer output;
              transientCache:       Integer output;
              remoteTransientCache: Integer output);
```

The **Node** class **getCacheSizes** method retrieves the persistent, transient, and remote transient cache values of the node on which the method is executing. These values, which are in bytes, represent the maximum amount of memory that is allocated by the JADE Object Manager library for caching objects in the node.

Node::*getCacheSizes64* Method

Signature

```
getCacheSizes64(persistentCache:      Integer64 output;
                transientCache:       Integer64 output;
                remoteTransientCache: Integer64 output);
```

The **Node** class **getCacheSizes64** method retrieves the persistent, transient, and remote transient cache values of the node on which the method is executing. These values, which are in bytes, represent the maximum amount of memory that is allocated by the JADE Object Manager library for caching objects in the node.

Node::*getObjectCaches* Method

Signature

```
getObjectCaches(dynObj:    JadeDynamicObject input;
                cacheType: Integer);
```

The **Node** class **getObjectCaches** method retrieves information about cache activity for the node specified as the method receiver. You can call this method using any node as the receiver.

The values are returned as properties of a **JadeDynamicObject** object, relating to the specified node.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

An example that shows the use of a method to handle the wrapping of negative values when calculating differences is provided later in this subsection. See also the **Node** class **getObjectCaches** method, in Volume 2 of the *JADE Encyclopaedia of Classes*.

The **cacheType** parameter specifies whether information is retrieved from the persistent, transient, or remote transient cache. The retrieved values are listed in the following table.

Value	Description
1	Persistent cache
2	Transient cache
3	Remote transient cache (applicable only on server nodes)

You should use a blank dynamic object the first time you call this method, which adds properties to the object. You can then use this object in subsequent calls.

If the dynamic object passed to the method already contains properties but they do not match the properties to be returned, the existing dynamic object properties are removed and then replaced with the appropriate properties.

Notes The **getObjectCaches** method is most efficient when the properties match those to be returned.

The properties that are returned to the dynamic object specified in the **dynObj** parameter are listed in the following table.

All Cache Types	Persistent Cache Only	Primitive Type	Value
clockTicks		Integer64	
nodeCPUTime		Integer64	
nodeTicks		Integer64	
cacheType		Integer64	
hits		Integer64	
misses		Integer64	
topOfLRUHits (no longer maintained)		Integer64	0
createdBuffers		Integer64	
cleanSwappedBuffers		Integer64	
dirtySwappedBuffers		Integer64	
resizedBuffers		Integer64	
maximumBufferSize		Integer64	
totalNumberOfBuffers		Integer64	
availableBufferSize		Integer64	
maximumOverdraftBufferSize		Integer64	
overdraftBufferSize		Integer64	
deadBuffers		Integer64	
	totalOperations	Integer64	
	currentOperations	Integer64	

All Cache Types	Persistent Cache Only	Primitive Type	Value
	currentBuffers	Integer64	
	deletedBuffers	Integer64	
	copiedBuffers	Integer64	
	newBuffers	Integer64	
	fetches	Integer64	
	duplicateFetches	Integer64	
	totalSwaps	Integer64	
	totalOpsWhenSwapped	Integer64	
	minOpsWhenSwapped	Integer64	
	maxOpsWhenSwapped	Integer64	
	totalAgeWhenSwapped	Integer64	
	minAgeWhenSwapped	Integer64	
	maxAgeWhenSwapped	Integer64	
	lruTraversals	Integer64	
	totalLruTraversalTicks	Integer64	
	latestLruTraversalTicks	Integer64	
	totalCacheCoherencyNotifications	Integer64	
	cacheCoherencyNotificationHits	Integer64	
	cacheCoherencyUpdatedObjects	Integer64	
	cacheCoherencyObjectHits	Integer64	
	cacheCoherencyObjectMisses	Integer64	
	cacheCoherencyRangeRequests	Integer64	
	nodeLockRemoveRequestsSent	Integer64	
	nodeLockRemoveRequestsRcvd	Integer64	
	nodeLockSwapOutRequestsSent	Integer64	

For explanations about these direct node statistics, see "Cache Statistics" under "Statistics File Format", later in this chapter.

The following example that shows the use of the **getObjectCaches** method to return the number of node ticks over a specified period.

```
getNodeTicks(seconds: Integer): Integer64;
vars
    sample              : JadeDynamicObject;
    cumulativeNodeTicks : Integer64;
    nodeTicks           : Integer64;
begin
    create sample transient;
    node.getObjectCaches(sample, 1);    // persistent cache
    cumulativeNodeTicks := sample.getPropertyValue("nodeTicks").Integer64;
```

```
        process.sleep(seconds*1000);
        node.getObjectCaches(sample, 1);
        nodeTicks := sample.getPropertyValue("nodeTicks");
        if nodeTicks >= cumulativeNodeTicks then
            nodeTicks := nodeTicks - cumulativeNodeTicks;
        else // wrapped to a negative value
            nodeTicks := (Max_Integer64 - cumulativeNodeTicks) + 1 +
                     (nodeTicks - Min_Integer64);
        endif;
        delete sample;
        return nodeTicks;
    end;
```

Node::*setCacheSizes* Method

Signature `setCacheSizes(persistentCache: Integer io;`
`transientCache: Integer io;`
`remoteTransientCache: Integer io);`

The **Node** class **setCacheSizes** method changes the sizes of the persistent, transient, and remote transient cache on the node on which the method is executing to be set to the specified values.

The cache size cannot be set lower than the minimum for that type of cache. In addition, the cache size sometimes cannot be reduced because of current usage of objects on it. If the cache size cannot be set to the requested value, it is increased or reduced as much as possible at that time. No exception is raised.

The cache size on 32-bit systems cannot exceed 4G bytes.

Note The values are set for the current JADE session only.

When you next initiate JADE, the **ObjectCacheSizeLimit**, **TransientCacheSizeLimit**, and **RemoteTransientCacheSizeLimit** parameter values in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file are those that are used for the persistent, transient, and remote transient cache sizes, respectively. For details about cache sizes, see "Cache Sizes", in Chapter 1, and the appropriate parameters in "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*.

Node::*setCacheSizes64* Method

Signature `setCacheSizes64(persistentCache: Integer64 io;`
`transientCache: Integer64 io;`
`remoteTransientCache: Integer64 io);`

The **Node** class **setCacheSizes64** method changes the sizes of the persistent, transient, and remote transient cache on the node on which the method is executing to be set to the specified values.

The cache size cannot be set lower than the minimum for that type of cache. In addition, the cache size sometimes cannot be reduced because of current usage of objects on it. If the cache size cannot be set to the requested value, it is increased or reduced as much as possible at that time. No exception is raised.

The cache size on 32-bit systems cannot exceed 4G bytes.

Note The values are set for the current JADE session only.

When you next initiate JADE, the values in the **ObjectCacheSizeLimit**, **TransientCacheSizeLimit**, and **RemoteTransientCacheSizeLimit** parameters in the appropriate [JadeClient] or [JadeServer] section of the JADE initialization file are those that are used for the persistent, transient, and remote transient cache sizes, respectively. For details about cache sizes, see the appropriate parameters in "JADE Object Manager Client Module Section [JadeClient]" or "JADE Object Manager Server Section [JadeServer]", in the *JADE Initialization File Reference*.

Method Profiling

Method profiling enables you to monitor JADE and external methods called by a specified JADE process. Information recorded for each method includes the method name, the number of calls, as well as CPU time and clock time consumed by the method.

Method profiling differs from the JADE Interpreter method profiling documented under "Profiling an Application", in Chapter 1 of the *JADE Runtime Application Guide*, as you can initiate it by any process for any process in an application without requiring code changes, and it can record external methods as well as internal methods.

The **Process** instance used as the receiver indicates the process for which method profiling is to be applied. You can specify any current process.

Notes A process can initiate method profiling on a target process and a different process can clear or end the profiling.

It is recommended that when investigating application performance, only one of the JADE Profiler, JADE Monitor, or method profiling is used at any one time, as the results reported when any of these are combined is undefined.

Methods specified as **serverExecution** are not profiled unless they are executed from server applications or in single user mode. For more details, see the following subsections.

Process::*beginMethodProfiling* Method

Signature `beginMethodProfiling(option: Integer);`

The **Process** class **beginMethodProfiling** method starts method profiling for the receiving **Process** instance, which can be any current process including processes running on other nodes.

The values for the **option** parameter and the corresponding range of methods to be profiled are listed in the following table.

Value	Profiles …
1	All called methods whether nominated or not
2	Nominated methods and their nested method calls
3	Nominated methods only

Methods are nominated using the **profileMethod** method of the **Process** class.

The following actions occur if the **beginMethodProfiling** method is called when profiling is already in effect for the target process.

1. Profiling information is reset
2. The profiling option is adjusted to match the **option** parameter

3. The list of nominated methods is retained

Methods specified as **serverExecution** are not profiled, unless executed from server applications or in single user mode.

Process::*endMethodProfiling* Method

Signature `endMethodProfiling();`

The **Process** class **endMethodProfiling** method stops method profiling for the target process used as the method receiver. The profiling information is retained until the **removeMethodProfileInfo** method is called or the target process terminates.

The **endMethodProfiling** method has no effect if it is called when profiling is not in effect for the target process.

Process::*profileMethod* Method

Signature
```
profileMethod(m: Method;
              b: Boolean);
```

The **Process** class **profileMethod** method selects or deselects a method to be profiled for the target process used as the method receiver.

You would usually call this method before method profiling is started for the target process using the **beginMethodProfiling** method with the **option** parameter set to **2** or **3**, to indicate a subset of methods are to be profiled.

The **m** parameter is a **Method** object reference, which can be a JADE method or an external method.

The **b** parameter indicates if the method is to be added to or removed from the list of nominated methods. If the value of the **b** parameter is **true**, the method is added. If it is **false**, the method is removed.

You can call the method when method profiling is already in effect for the target process. Changes to the list of nominated methods take immediate effect.

The method has no effect if you call it to add a method that is already in the list of nominated methods. Similarly, it has no effect if removing a method that is not in the nominated list. The list of nominated methods is retained until the **removeMethodProfileInfo** method is called or the target process terminates.

Process::*getMethodProfileInfo* Method

Signature
```
getMethodProfileInfo(jdo:       JadeDynamicObject input;
                     truncated: Boolean output);
```

The **Process** class **getMethodProfileInfo** method retrieves method profiling information for the process specified as the method receiver.

The retrieved information is presented as a set of dynamic objects in the **children** collection of the dynamic object instance specified by the **jdo** parameter; that is, the information is contained in the collection **jdo.children**.

The calling process is responsible for creating and deleting the **JadeDynamicObject** passed to the method. It is also responsible for deleting the **JadeDynamicObject** instances inserted into the **jdo.children** collection (for example, by purging the collection).

If the **JadeDynamicObject** instance used as the **jdo** parameter is persistent, the **JadeDynamicObject** instances added to the **children** collection are also persistent. Similarly, if the object is transient, the child dynamic objects are transient.

The properties of the **JadeDynamicObject** instances in the **children** collection are listed in the following table.

Property	Type	Description
method	Object	Reference to the **Method** instance for the profiled method
calls	Integer64	Number of times the method was called
totalCpuTime	Integer64	Total CPU time in microseconds consumed by the method and the methods it called
cpuTimeInMethod	Integer64	Total CPU time in microseconds consumed by the method only; that is, excluding time consumed by the methods it called
totalClockTime	Integer64	Total clock time in microseconds elapsed while executing the method and the methods it called
clockTimeInMethod	Integer64	Total clock time in microseconds elapsed while executing the method only; that is, excluding time elapsed while executing the methods it called

Time spent in recursive method calls is correctly accounted for as time spent executing the method.

The **truncated** parameter indicates if the amount of method profile information to be retrieved exceeded an internal buffer limit so was truncated. If the **truncated** parameter is returned as **false**, all method profiling information is present in the **JadeDynamicObject** instance specified in the **jdo** parameter. If it is returned as **true**, some entries have been truncated. When truncation occurs, entries with the lowest "total calls" values are omitted.

Notes Truncation occurs when the amount of profiling information to be returned exceeds 1,000 entries.

Truncation occurs only when method profiling information is retrieved for processes running on remote nodes.

The CPU time has a granularity of 10 or 15 milliseconds, which means that the CPU time figures for methods of short duration are subject to inaccuracy due to the large granularity. However, the clock times have a much smaller granularity and are therefore more accurate.

Note Clock times may fluctuate depending on other activity on the same machine. The total clock times include time spent waiting; for example, to wait for a Window event, to lock an object, or for a user response to a modal form.

The **children** collection of the **jdo** parameter passed to the **getMethodProfileInfo** method is purged each time the method is called.

When displaying method profiling results, you can use the **qualifiedName** method of the corresponding **Method** instance (specified in the **method** parameter) to obtain the name of a profiled method, and the **Object** class **isKindOf** method to determine if the method is an external method or a JADE method.

Any process can call the **getMethodProfileInfo** method to retrieve method profiling information, if it is available. However, any process can also call the **removeMethodProfileInfo** method to remove profiling information.

If you call the **getMethodProfileInfo** method on a target process that has terminated, an exception of type 4 (*Object not found*) is raised.

For usage and output examples of this method, see the **Process** class **getMethodProfileInfo** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Process::*removeMethodProfileInfo* Method

Signature `removeMethodProfileInfo();`

The **Process** class **removeMethodProfileInfo** method removes all method profiling information and any list of nominated methods for the target process specified as the method receiver.

If method profiling is in effect for the target process, it is stopped before the profiling information is removed. If method profiling is not active for the target process or there is no method profiling information, this method has no effect.

Getting Database File Statistics

The **DbFile** class provides a method that enables you to retrieve persistent database file statistics. For details, see the following subsection. (For details about methods that enable you to retrieve database file read and write activity, see "Analyzing Database Files", later in this chapter.)

DbFile::*getStatistics* Method

Signature `getStatistics(jdo: JadeDynamicObject input);`

The **DbFile** class **getStatistics** method 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
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. To work out the differences, you must compare values from one execution of the **getStatistics** method with the previous values.

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.

For examples of the use of and output from this method, see the **DbFile** class **getStatistics** method in Volume 1 of the *JADE Encyclopaedia of Classes*.

Analyzing Database Files

The **DbFile**, **JadeDatabaseAdmin**, and **JadeDbFilePartition** classes provide methods that enable you to retrieve database file read and write activity. For details, see the following subsection. (For details about retrieving persistent database file statistics, see "Getting Database File Statistics", earlier in this chapter.)

JadeDatabaseAdmin::*getDbFiles* Method

Signature `getDbFiles(fileKinds: Integer;`
`dbfiles: DbFileArray input);`

The **JadeDatabaseAdmin** class **getDbFiles** method populates a **DbFile** array with references to database files of the kinds specified in the **fileKinds** parameter, by searching all schemas from the Root Schema 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", in Volume 1 of the *JADE Encyclopaedia of Classes*.)

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
```

For an example of this method, see the **JadeDatabaseAdmin** class **getDbFiles** method in Volume 1 of the *JADE Encyclopaedia of Classes*.

DbFile::*getfileStatus* Method

Signature `getFileStatus(): Integer;`

The **DbFile** class **getFileStatus** method 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
Status_NotCreated	File deleted or not yet created	2
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.

DbFile::*isPartitioned* Method

Signature `isPartitioned(): Boolean;`

The **DbFile** class **isPartitioned** method returns **true** if the associated database file is partitioned; otherwise it returns **false**. (See also the **DbFile** class **freeze** and **thaw** methods, in Volume 1 of the *JADE Encyclopaedia of Classes*.)

DbFile::*getOpenPartitions* Method

Signature
```
getOpenPartitions(partitionList: JadeDbFilePartitionArray input;
                  maxEntries:    Integer);
```

The **DbFile** class **getOpenPartitions** method populates the **partitionList** parameter with references to **JadeDbFilePartition** instances; one for each open partition of the associated database file.

If the value of the **maxInstances** parameter is non-zero, the array contains an entry for the specified number of open partitions.

DbFile::*getPartitions* Method

Signature
```
getPartitions(partitionList: JadeDbFilePartitionArray input;
              maxEntries:    Integer);
```

The **DbFile** class **getPartitions** method populates the **partitionList** parameter with references to **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.

JadeDbFilePartitions::*getFileStatus* Method

Signature `getFileStatus(): Integer;`

The **JadeDbFilePartition** class **getFileStatus** method returns the status of a dropped single physical database file partition during the backup process. The status of the database file partition is represented by **DbFile** class constants listed in the following table.

Constant	Description	Integer Value
Status_Deleted	File partition deleted from control file	6
Status_InvalidPath	Invalid database file partition path in control file	7
Status_Missing	File partition defined in the schema but does not exist in the database	3
Status_NotAssigned	File partition not defined in control file	1
Status_NotCreated	File partition deleted or not yet created	2
Status_Resident	File partition is resident on disk	4
Status_Unmapped	File partition 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 file partitions prior to commencing the backup or to determine the status of database file partitions 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.

JadeDbFilePartitions::*isFrozen* Method

Signature `isFrozen(): Boolean;`

The **JadeDbFilePartition** class **isFrozen** method returns **true** if the associated database file has been frozen (that is, the partition has been converted to read-only mode after which any object update, delete, or create operations is not permitted); otherwise it returns **false**. (See also the **JadeDbFilePartition** class **freeze** and **thaw** methods, in Volume 1 of the *JADE Encyclopaedia of Classes*.)

JadeDbFilePartitions::*isOffline* Method

Signature `isOffline(): Boolean;`

The **JadeDbFilePartition** class **isOffline** method returns **true** if the associated database partition has been marked offline (that is, the file partition is not accessible from the file system); otherwise it returns **false**. (See also the **JadeDbFilePartition** class **markOffline** and **markOnline** methods, in Volume 1 of the *JADE Encyclopaedia of Classes*.)

JadeDbFilePartitions::*getStatistics* Method

Signature `getStatistics(jdo: JadeDynamicObject input);`

The **JadeDbFilePartition** class **getStatistics** method returns statistics relating to read and write operations on the persistent database file 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).

For an example of this method, see the **JadeDbFilePartition** class **getStatistics** method in Volume 1 of the *JADE Encyclopaedia of Classes*.

Getting Database Statistics

The **System** class provides methods that enable you to retrieve database statistics. The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. For details, see the following subsections.

System::*getDatabaseStats* Method

Signature `Signature getDatabaseStats(jdo: JadeDynamicObject input);`

The **System** class provides the **getDatabaseStats** method returns statistics relating to persistent database activity.

The values returned in the dynamic object specified by the **jdo** parameter are described in the following table. Properties are null at the initiation of the database server node and are applicable only to the running database instance for the lifetime of the database server node. Where properties hold cumulative values (fileOpens, checkPoints, and so on), JADE applications will need to compare values from one call to the next, to work out the value differences.

The properties returned in the dynamic object are listed in the following table.

Property	Type	Description
fileOpens	**Integer64**	The number of database file open operations performed
fileCloses	**Integer64**	The number of database file close operations performed
committedTrans	**Integer64**	The number of transactions committed to the database
abortedTrans	**Integer64**	The number of transactions aborted
checkPoints	**Integer64**	The number of database checkpoints that have taken place
lastCheckPointDate	**Date**	The date of the last checkpoint taken
lastCheckPointTime	**Time**	The time of the last checkpoint taken
lastCheckPointDuration	**Integer64**	The time in milliseconds for the last checkpoint taken
maxCheckPointDuration	**Integer64**	The maximum time in milliseconds for any checkpoint taken
avgCheckPointDuration	**Integer64**	The average time in milliseconds for any checkpoint taken
editionGets	**Integer64**	The number of getEdition operations performed
objectGets	**Integer64**	The number of getObject operations performed
objectCreates	**Integer64**	The number of objects created
objectUpdates	**Integer64**	The number of objects updated
objectDeletes	**Integer64**	The number of objects deleted
dirtyReads	**Integer64**	The number of dirty reads
osmReads	**Integer64**	The number of reads of a previously committed edition
priorEditionReads	**Integer64**	The number of reads requiring fetch of a previously committed edition from the journal

Property	Type	Description
absentCollGets	**Integer64**	The number of osmReads for collection headers
overflowDeleteGets	**Integer64**	The number of osmReads by the deleting transaction

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.

For an example of the use of and output from this method, see the **System** class **getDatabaseStats** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

System::*getDbDiskCacheStats* Method

Signature `getDbDiskCacheStats(jdo: JadeDynamicObject input);`

The **System** class **getDbDiskCacheStats** method returns statistics relating to the persistent database cache. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The returned values are **Integer64** values representing counts of actions pertaining to the persistent database cache, most of which are cumulative, starting from when the database was opened. JADE applications that use the **getDbDiskCacheStats** method defined in the **System** class therefore need to compare values from one call to the next, to work out the value differences.

The properties returned in the dynamic object are listed in the following table.

Property	Description
cacheMisses	The number of cache misses; that is, where the address referenced a block that was not found in the cache
gets	The number of read requests received
puts	The number of write requests received
blockReads	The number of block reads from disk performed
getsWithFetch	The number of read requests with additional block prefetch received
putsWithFetch	The number of write requests with additional block prefetch received
blocksFetched	The number of blocks prefetched with read from disk
blockReadsMultiple	The number of prefetch reads performed
bufferReassigns	The number of times buffers were reassigned for use with a different address
bufferSteals	The number of times the buffer reassigned was dirty, requiring writing
maxHashCollisions	The maximum number of hash collisions
maxConcFlushIos	The maximum number of concurrent flush write operations
blockWrites	The number of single block writes to disk performed
blockWritesMultiple	The number of multiple block writes to disk performed

For further explanation of these values, refer to the JADE Monitor knowledge base.

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.

For an example of the use of and output from this method, see the **System** class **getDbDiskCacheStats** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Locks

The **Node** and **System** classes provide methods that enable you to retrieve the current locks and lock requests waiting for a shared transient object to be unlocked. For details, see the following subsections.

Node::*getLocks* Method

Signature
```
getLocks(locks:      LockArray input;
         maxEntries: Integer);
```

The **Node** class **getLocks** method populates the array specified in the **locks** parameter with transient instances of the current locks for the shared transient objects in the node specified as the method receiver.

Any current node can be used as the method receiver.

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

Parameter	Specifies the…
locks	Locks array that is to be populated with the lock instances
maxEntries	Maximum number of lock instances that are to be included in the array

For an example of the use of and output from this method, see the **Node** class **getLocks** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Node::*getQueuedLocks* Method

Signature
```
getQueuedLocks(locks:      LockArray input;
               maxEntries: Integer);
```

The **Node** class **getQueuedLocks** method populates the array specified in the **locks** parameter with transient instances of the lock requests that are waiting for shared transient objects in the node specified as the method receiver to be unlocked by the processes that currently have them locked.

Any current node can be used as the method receiver.

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

Parameter	Specifies the…
locks	Locks array that is to be populated with the lock request instances
maxEntries	Maximum number of lock instances that are to be included in the array

The calling process is responsible for creating and deleting the **LockArray** instance used with this method, as well as deleting the **Lock** instances inserted into the array.

For an example of the use of and output from this method, see the **Node** class **getQueuedLocks** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

System::*getLocks* Method

Signature `getLocks(locks: LockArray input;`
`maxEntries: Integer);`

The **System** class **getLocks** method populates the array specified in the **locks** parameter with transient instances of the current persistent object locks held by all the processes in the system.

The parameters of the **getLocks** method are listed in the following table.

Parameter	Specifies the…
locks	Locks array that is to be populated with the lock instances
maxEntries	Maximum number of lock instances that are to be included in the array

For an example of the use of this method, see the **System** class **getLocks** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

System::*getQueuedLocks* Method

Signature `getQueuedLocks(locks: LockArray input;`
`maxEntries: Integer);`

The **System** class **getQueuedLocks** method is similar to the **System** class **getObjectLockProcesses** and **getLocks** methods, but it includes only the lock requests that are waiting for objects to be unlocked by the processes that currently have them locked.

The value of the **maxEntries** parameter specifies the maximum number of entries to be inserted into the array specified by the **locks** parameter. Entries are inserted in no particular order.

Lock objects returned in the **locks** parameter can have lock entries in the array that have the **Lock** class **lockedBy** property set to **null** if the lock request is still waiting to be processed in the lock queue.

When this occurs, the process that caused the lock request to be queued has already released it but because of high activity on the executing node, the lock request has not been retried.

For an example of the use of this method, see the **System** class **getQueuedLocks** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Notification Statistics

In addition to the information retrieved in request statistics (for details, see "Getting Request Statistics", later in this chapter), the **Node** and **System** classes provide the **getNotes** method, which enables you to retrieve the number of current notification requests in the system.

You can obtain notification statistics at the following levels.

- Node, representing totals for notifications from a specified node
- System, representing totals for all notifications from all nodes

For details, see the following subsections.

Node::*getNotes* Method

Signature
```
getNotes(notes:      NotificationArray input;
         transients: Boolean;
         maxEntries: Integer);
```

The **Node** class **getNotes** method is not yet implemented. It is reserved for future use.

System::*getNotes* Method

Signature
```
getNotes(notes:      NotificationArray input;
         transients: Boolean;
         maxEntries: Integer);
```

The **System** class **getNotes** method populates the array specified in the **notes** parameter with transient instances of the current notification requests by all the processes in the system.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

Note As this method creates transient instances of the **Notification** class, it is the responsibility of the method caller to purge the collection used by the method to delete these transient instances. The collection should be purged *before* the deletion of the notification array passed to the method in the **notes** parameter.

For an example of the use of this method, see the **System** class **getNotes** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Object Access Information

The **System** class provides methods that enable you to retrieve information about the access of objects. Access counts for classes are held on the database server node, and are incremented every time an instance of that class or one of its subobjects is written to the database or fetched from the database. Persistent object accesses only are counted.

Note The access counts do not directly indicate how many times applications have used objects. If an object resides in the persistent object cache, it may not have to be fetched from the database when used. The access counts reflect database activity, rather than application activity.

When dealing with classes, the class number can be retrieved using the **number** property of the **Class** class, and the class with a particular number can be found using the **getClassByNumber** method of the **Schema** class.

To reduce any overhead caused by maintaining class access frequency statistics, the **Process** class provides methods that enable you to turn class access frequency counters on and off and to query the status of classes access frequency statistics, including the UTC time at which the recording of process counters were enabled.

JADE maintains a list of the processes that have class access frequency counters enabled. When a process requests the disabling of class access frequency statistics, the process is removed from the list. Statistics are disabled only when no processes remain in the list. If a process terminates without having turned class access frequency statistics disabled, the process is removed from the list and statistics are turned off if there are no other processes in the list. For details, see the following subsections.

Process::*enableClassAccessFrequencies* Method

Signature `enableClassAccessFrequencies(enable: Boolean): Boolean serverExecution;`

The **Process** class **enableClassAccessFrequencies** method enables or disables class access frequency counters.

Enabling class access frequency counters when they have already been enabled for the current process is ignored, as is the disabling of class access frequency counters when they have already been disabled for the current process.

The return value indicates whether the current process has already enabled the frequency counters.

The frequency counters are maintained by JADE if at least one process enables them. Similarly, they are disabled when all processes disable them. If a process enables frequency counters and does not explicitly disable them before signing off, they are implicitly disabled during the sign-off process.

Process::*classAccessFrequenciesStatus* Method

Signature
```
classAccessFrequenciesStatus(processList: ProcessDict input;
                             startTime:  TimeStamp output): Boolean serverExecution;
```

The **Process** class **classAccessFrequenciesStatus** method returns the status of class access frequency counters.

The value of the **processList** parameter is populated with the process oids for all processes that enabled class access frequency counters. The value of the **startTime** parameter is the time the first process enabled frequency counters.

The return value indicates whether frequency counters are enabled; that is, the value of the **processList** parameter is greater than zero (**0**).

System::*getClassAccessFrequencies* Method

Signature
```
getClassAccessFrequencies(clsNumArray: IntegerArray;
                          freqArray:   Integer64Array input);
```

The **System** class **getClassAccessFrequencies** method returns access counts for specified classes. The class numbers of these classes are added to the array specified by the **clsNumArray** parameter before the method is called.

An exception is raised when you call this method and frequency counters are not enabled. (For details about enabling and obtaining the status of class access frequency counters, see the **Process** class **enableClassAccessFrequencies** and **classAccessFrequenciesStatus** methods, respectively.)

The access counts are returned as **Integer64** values in the **freqArray** parameter, which is an instance of the **Integer64Array** class, passed to this method. Each entry in the **freqArray** array contains information relating to the class number specified by the entry with the same index in the **clsNumArray** array. If a class number is invalid, the corresponding access count is set to zero (**0**). The **freqArray** array is cleared every time the method is called. The calling process is responsible for creating and deleting the two arrays used with this method.

For an example of the use of and output from this method, see the **System** class **getClassAccessFrequencies** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

System::*getMostAccessedClasses* Method

Signature `getMostAccessedClasses(clsNumArray: IntegerArray input;`
`freqArray: Integer64Array input;`
`maxWanted: Integer);`

The **System** class **getMostAccessedClasses** method returns access counts for the classes that have been most frequently accessed since the database server node was initialized.

The information is returned in a pair of arrays. The **clsNumArray** array contains a set of class numbers, and the **freqArray** array contains a matching set of access counts. Each entry in the **freqArray** array corresponds to the entry with the same index in the **clsNumArray** array. The entries in the array are sorted in descending order of access count; that is, the class with the highest access count is the first array member, the class with the second highest access count is second, and so on.

The **maxWanted** parameter specifies the maximum number of entries to be placed in the arrays.

The calling process is responsible for creating and deleting the **clsNumArray** array and the **freqArray** array.

When the **getMostAccessedClasses** method is called, the arrays passed as parameters are cleared of all entries.

System::*beginObjectTracking* Method

Signature beginObjectTracking(fileName: String);

The **System** class **beginObjectTracking** method starts recording persistent object activity on the database server.

The information is recorded in a file on the database server node specified by the **fileName** parameter.

The recorded information distinguishes between read, lock, create, update, delete, and unlock operations.

Note This does not necessarily record every time an application uses an object, because if the object resides in the persistent object cache, it may not have to be fetched from the database.

Object tracking is terminated by calling the **System** class **endObjectTracking** method.

The file is a standard text (.**txt**) file. The first line is a header record and the last line is a trailer record. The lines in-between represent individual object accesses. In each line, fields are separated by spaces.

The format of the three types of records is similar to that used for node sampling files, documented under "Statistics File Format", later in this chapter.

The format of the header record is shown in the following table.

Field	Description
Record type	21 (object tracking header record)
Node instance id	Instance id of the database server node
Node type	1 for the database server, 3 for the application server in single user mode, and 5 for standard client in single user mode
Computer name	The name of the machine on which the database server node is running
JADE version number	In the format *major-release-num.minor-release-num.build-num.patch-num*; for example, 7.1.03.3

Field	Description
Current date	Current date in Julian day format
Current time	Current time in milliseconds from midnight UTC
Current UTC bias	UTC bias for the location in which the database server node is executing, in minutes
Current date and time	Text representation of the date and time
Clock ticks	Microseconds elapsed since the database server node was started

The format of the trailer record is shown in the following table.

Field	Description
Record type	23 (object tracking trailer record)
Node instance id	Instance id of the database server node
Current date	Current date in Julian day format
Current time	Current time in milliseconds from midnight UTC
Current date and time	Text representation of the date and time
Current UTC bias	UTC bias for the location in which the database server node is executing, in minutes
Clock ticks	Microseconds elapsed since the database server node was started

The format of the data records is shown in the following table.

Field	Description
Record type	22 (object tracking data record)
Node instance id	Instance id of the node from which the access request initiated
Process number	The identifying number of the process making the request
Clock ticks	Microseconds elapsed since the database server node was started
Request type	1 for get object, 2 for lock object, 3 for unlock object, 20 for create object, 21 for update object, and 22 for delete object
Class id	Class id of the object being accessed
Instance id	Instance id of the object being accessed
Parent class id	Parent class id of the object being accessed
Sub level	Sub-level of the object being accessed
Sub id	Sub-id of the object being accessed
Edition	Edition of the object being accessed
Process instance id	Instance id of the process making the request
Lock type	The lock type, which is 1 for shared, 2 for reserved, and 3 for exclusive
Duration	The lock duration, which is zero (**0**) for transaction and 1 for session
Kind	The lock kind (transaction or session), which is zero (**0**) for normal and 2 for a node lock
Millisecond	The length of time the lock was held

The following is an example of the data record for the unlocking of a shared lock, transaction duration on object 127.206.5.4.14:2 held by process 187.3 running on node 186.2. The lock was held for at least 15 milliseconds.

```
22 2 2 169691499 3 127 206 5 4 14 2 3 1 0 0 15
```

For node locks, the process instance id is that of the background process of the associated node. When a server application unlocks a shared lock on a stable object, the unlock request does not normally appear in the file because only the release of node locks is recorded for stable objects.

Note The type, duration, kind, and millisecond values are present only for unlock object requests (that is, request type 3).

Only one object tracking session can be active at a time. If the **beginObjectTracking** method is called when object tracking is already active, exception 1138 (*Object tracking is already active*) is raised.

Caution Use the **beginObjectTracking** method judiciously. When object tracking is active, the tracking file can fill very rapidly. You should therefore use object tracking in short bursts.

System::*endObjectTracking* Method

Signature `endObjectTracking();`

The **System** class **endObjectTracking** method ends recording persistent object read operations from the database or write operations to the database.

An object tracking session is started using the **System** class **beginObjectTracking** method. If the **endObjectTracking** method is called when object tracking is not active, exception 1139 (*Object tracking is not active*) is raised.

Getting Process Information

The **Process** class provides methods that enable you to retrieve information about the process; for example, request statistics and timer information. For details, see the following subsections.

Getting Request Statistics

You can obtain request statistics at the following levels.

- Node, representing totals for requests from a specified node
- Process, representing totals for requests from a specified process
- System, representing totals for requests from all nodes

As the statistics are maintained on the database server node, they record requests and replies sent between the database server node and client nodes. You can retrieve the statistics as combined totals for all request types or as individual totals for each request type. See also "Getting Notification Statistics", earlier in this chapter.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

For details, see the following subsections.

Node Request Statistics Method

A node can use the **getRequestStats** method of the **Node** class to obtain its own request statistics. For details, see the following subsection.

Node::*getRequestStats* Method

Signature `getRequestStats(jdo: JadeDynamicObject input);`

The **Node** class **getRequestStats** method returns node statistics relating to persistent database requests from the receiving node that is the method receiver. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter. The properties returned in the dynamic object are listed in the following table.

Property	Description
committedTransactions	Total number of transactions committed
abortedTransactions	Total number of transactions aborted
getObjects	Total number of requests to retrieve an object from the database
createObjects	Total number of created objects
deleteObjects	Total number of deleted objects
updateObjects	Total number of object updates
lockObjects	Total number of lock requests
unlockObjects	Total number of unlock requests
beginNotifications	Total number of **beginNotification** instructions
endNotifications	Total number of **endNotification** instructions
serverMethodExecutions	Total number of **serverExecution** methods executed
causeEvents	Total number of **causeEvent** instructions

Properties are added to the object when the method is first called. The object can then be used in subsequent calls. If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties. The method is most efficient when the properties match those to be returned.

For examples of the use of and output from this method, see the **Node** class **getRequestStats** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Process Request Statistics Methods

A process can use the **getRequestStatistics** method of the **Process** class to obtain its own request statistics. To obtain *request* statistics for any other process, you must use the **Process** class:

- **sendRequestStatistics** method in conjunction with the **extractRequestStatistics** method
- **sendCallStackInfo** method in conjunction with the **getCallStackInfo** method
- **sendTransientFileAnalysis** method
- **sendTransientFileInfo** method

Each line of the output of these methods is delimited by the line feed (**Lf**) character.

Tip An easy way to view this output is to use the **writeString** method of the **File** class to write the string returned by the method to a file and then view it with WordPad or another text editor.

To obtain *process* statistics for any other process, call the:

1. **Object** class **beginNotification** method, to register to receive the notification containing the statistics
2. **Process** class **sendRequestStatistics** method, to request the target process to send a notification containing statistics
3. **Process** class **extractRequestStatistics** method in the **Object** class **userNotification** method, to retrieve the statistics

For details, see the following subsections.

Process::*getRequestStatistics* Method

Signature
```
getRequestStatistics(dynObj:        JadeDynamicObject input;
                     localOrRemote: Integer);
```

The **Process** class **getRequestStatistics** method retrieves node sampling values relating to the current process that is executing the method. The values are returned as properties of a **JadeDynamicObject** object.

If the **localOrRemote** parameter is set to **1**, the statistics for all requests invoked on the local node are returned as properties in the dynamic object specified in the **dynObj** parameter. If the **localOrRemote** parameter is set to **2**, the statistics for all requests from the local node to remote nodes are returned as properties in the specified dynamic object.

Most of the request statistics are collected only when node sampling is active on the node. The values that are reported independent of node sampling are thin client statistics that are part of the local request statistics.

The properties that are returned to the dynamic object specified in the **dynObj** parameter when the **localOrRemote** parameter is set to **1** (that is, local node sampling) are listed in the following table.

Property	Primitive Type
processId	Integer
clockTicks	Integer64
nodeCPUTime	Integer64
nodeTicks	Integer64
processCPUTime	Integer64
processTicks	Integer64
processLogicalClock	Integer64
transientObjectCreates	Integer64
transientObjectDeletes	Integer64
transientObjectClones	Integer64
transientObjectCopies	Integer64
transientObjectGetProperties	Integer64
transientObjectSetProperties	Integer64
transientObjectJadeMethods	Integer64
transientObjectExternalMethods	Integer64
transientObjectLocks	Integer64

Property	Primitive Type
transientObjectUnlocks	Integer64
transientObjectRemoveLocks	Integer64
transientObjectGets	Integer64
transientObjectPuts	Integer64
transientBufferSwaps	Integer64
persistentObjectCreates	Integer64
persistentObjectDeletes	Integer64
persistentObjectClones	Integer64
persistentObjectCopies	Integer64
persistentObjectGetProperties	Integer64
persistentObjectSetProperties	Integer64
persistentObjectJadeMethods	Integer64
persistentObjectExternalMethods	Integer64
persistentObjectLocks	Integer64
persistentObjectUnlocks	Integer64
persistentObjectRemoveLocks	Integer64
persistentObjectGets	Integer64
persistentObjectPuts	Integer64
persistentBufferSwaps	Integer64
primitiveJadeMethods	Integer64
primitiveExternalMethods	Integer64
transientBeginTransactions	Integer64
transientEndTransactions	Integer64
transientAbortTransactions	Integer64
transientBeginNotifications	Integer64
transientEndNotifications	Integer64
transientCauseEvents	Integer64
transientReturnNotes	Integer64
receivedNotesFromLocal	Integer64
userRequests	Integer64
persistentBeginTransactions	Integer64
persistentEndTransactions	Integer64
persistentAbortTransactions	Integer64
persistentBeginNotifications	Integer64
persistentEndNotifications	Integer64

Property	Primitive Type
persistentCauseEvents	Integer64
persistentReturnNotes	Integer64
thinClientMsgsSentByAppServer	Integer64
thinClientBytesSentByAppServer	Integer64
thinClientMsgsSentByClient	Integer64
thinClientBytesSentByClient	Integer64
thinClientLogicMsgWaitTime	Integer64

For explanations about these statistics, see "Local Request Statistics Format" under "Statistics File Format", later in this chapter.

The properties that are returned to the dynamic object specified in the **dynObj** parameter when the **localOrRemote** parameter is set to **2** (that is, remote node sampling) are listed in the following table.

Property	Primitive Type
processId	Integer
clockTicks	Integer64
nodeCPUTime	Integer64
nodeTicks	Integer64
processCPUTime	Integer64
processTicks	Integer64
processLogicalClock	Integer64
rpcNewBufferGetObjects	Integer64
rpcUpdatedBufferGetObjects	Integer64
rpcNonUpdatedBufferGetObjects	Integer64
rpcTemporaryBufferGetObjects	Integer64
rpcNewBufferGetObjectGroups	Integer64
rpcUpdatedBufferGetObjectGroups	Integer64
rpcNonUpdatedBufferGetObjectGroups	Integer64
rpcTemporaryBufferGetObjectGroups	Integer64
rpcGetObjectGroups	Integer64
rpcNewBufferLockObjects	Integer64
rpcUpdatedBufferLockObjects	Integer64
rpcNonUpdatedBufferLockObjects	Integer64
rpcTemporaryBufferLockObjects	Integer64
rpcNewBufferLockObjectGroups	Integer64
rpcUpdatedBufferLockObjectGroups	Integer64

Property	Primitive Type
rpcNonUpdatedBufferLockObjectGroups	Integer64
rpcTemporaryBufferLockObjectGroups	Integer64
rpcLockObjectGroups	Integer64
rpcCreateObjects	Integer64
rpcUpdateObjects	Integer64
rpcDeleteObjects	Integer64
rpcUnlockObjects	Integer64
rpcUnlockObjectGroups	Integer64
rpcRemoveLocks	Integer64
rpcGetEditions	Integer64
rpcGetOids	Integer64
rpcBeginTransactions	Integer64
rpcEndTransactions	Integer64
rpcAbortTransactions	Integer64
rpcBeginNotifications	Integer64
rpcEndNotifications	Integer64
rpcCauseEvents	Integer64
rpcReturnNotes	Integer64
receivedNotesFromRemote	Integer64
rpcServerExecutions	Integer64
rpcSendDatabaseMessages	Integer64
rpcNewBufferGetObjectsTime	Integer64
rpcUpdatedBufferGetObjectsTime	Integer64
rpcNonUpdatedBufferGetObjectsTime	Integer64
rpcTemporaryBufferGetObjectsTime	Integer64
rpcNewBufferGetObjectGroupsTime	Integer64
rpcUpdatedBufferGetObjectGroupsTime	Integer64
rpcNonUpdatedBufferGetObjectGroupsTime	Integer64
rpcTemporaryBufferGetObjectGroupsTime	Integer64
rpcGetObjectGroupsTime	Integer64
rpcNewBufferLockObjectsTime	Integer64
rpcUpdatedBufferLockObjectsTime	Integer64
rpcNonUpdatedBufferLockObjectsTime	Integer64
rpcTemporaryBufferLockObjectsTime	Integer64
rpcNewBufferLockObjectGroupsTime	Integer64

Property	Primitive Type
rpcUpdatedBufferLockObjectGroupsTime	Integer64
rpcNonUpdatedBufferLockObjectGroupsTime	Integer64
rpcTemporaryBufferLockObjectGroupsTime	Integer64
rpcLockObjectGroupsTime	Integer64
rpcLockQueueWaitsTime	Integer64
rpcPutObjectsTime	Integer64
rpcUnlockObjectTime	Integer64
rpcUnlockObjectGroupsTime	Integer64
rpcRemoveLocksTime	Integer64
rpcGetEditionsTime	Integer64
rpcGetOidsTime	Integer64
rpcBeginTransactionsTime	Integer64
rpcEndTransactionsTime	Integer64
rpcAbortTransactionsTime	Integer64
rpcBeginNotificationsTime	Integer64
rpcEndNotificationsTime	Integer64
rpcCauseEventsTime	Integer64
rpcReturnNotesTime	Integer64
rpcServerExecutionsTime	Integer64
rpcSendDatabaseMessagesTime	Integer64
allRequestBytesSent	Integer64
allRequestBytesReceived	Integer64
allRequestPacketsSent	Integer64
allRequestPacketsReceived	Integer64
allReceiptsBytesReceived	Integer64
allReceiptsPacketsReceived	Integer64

For explanations about these statistics, see "Remote Requests Statistics Format" under "Statistics File Format", later in this chapter.

The following example, which shows the use of the **getRequestStatistics** method, retrieves 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;
```

Process::*sendRequestStatistics* Method

Signature `sendRequestStatistics(localOrRemote: Integer);`

The **Process** class **sendRequestStatistics** method requests a process to send a notification containing local or remote request statistics. The process is indicated by the **Process** instance used as the method receiver.

You can specify any current process to send statistics, including the requesting process itself and processes executing on other nodes.

The target object for the notification is the **Process** instance of the process making the request.

To request local statistics that record information about requests made on the node on which the process is running, set the value of the **localOrRemote** parameter to **1**. To request remote statistics, which record information about requests made between the client node and the database server node, set the value of the **localOrRemote** parameter to **2**.

Most of the request statistics are collected only when node sampling is active on the node. The values that are reported independent of node sampling are thin client statistics that are part of the local request statistics.

The target process is temporarily activated or interrupted to send the notification. After sending the notification, the process resumes the inactive state or restarts the interrupted action.

Note This method is asynchronous; that is, the **sendRequestStatistics** method does not wait until the statistics have been received. The statistics are received as a notification some time after the **sendRequestStatistics** method has been called.

The information in the notification relating to the request statistics is shown in the following table.

Parameter	Contains…
eventType	**Process_Local_Stats_Event** or **Process_Remote_Stats_Event** global constant in the **JadeProcessEvents** category
target	Process instance of the process that made the request
userInfo	Statistical values encoded within a Binary value

The process making the request should register to receive type **Process_Local_Stats_Event** (local request statistics) or **Process_Remote_Stats_Event** (remote request statistics) notifications on its **Process** instance using the **beginNotification** method defined in the **Object** class before executing the **sendRequestStatistics** method, as shown in the following code fragment.

```
self.beginNotification(process, Remote_Proc_Stats_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains local or remote request statistical information, the **userNotification** method of its **Process** instance should test whether the value of the **eventType** parameter is **Process_Local_Stats_Event** or **Process_Remote_Stats_Event**, which indicates local or remote request statistics notification.

The **userInfo** parameter of the notification should then be passed as a parameter to the **extractRequestStatistics** method, to extract the statistics as properties in a **JadeDynamicObject** instance.

If the target process (the method receiver) is not a valid current process, exception 1128 (*The target process is not valid*) is raised.

Process::*extractRequestStatistics* Method

Signature

```
extractRequestStatistics(proc:          Process output;
                         jdo:           JadeDynamicObject input;
                         localOrRemote: Integer;
                         any:           Any);
```

The **Process** class **extractRequestStatistics** method extracts request statistics from the **userInfo** part of notifications sent in response to **sendRequestStatistics** method requests.

The extracted statistics are inserted as properties in a **JadeDynamicObject** instance. These properties are the same properties as those of the **getRequestStatistics** method.

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

Parameter	Description
proc	An output parameter that receives a reference to the **Process** instance to which the statistics relate
jdo	A **JadeDynamicObject** instance into which the statistics values are placed as properties
localOrRemote	Set to **1** to extract local request statistics (event type **Process_Local_Stats_Event**) or set to **2** to extract remote request statistics (event type **Process_Remote_Stats_Event**)
any	The value of the **userInfo** parameter of the notification that was received

Any existing properties of the **JadeDynamicObject** instance are cleared every time the method is called. For a list and explanations about the properties that are returned by this method, see "Direct Retrieval of Node Sampling Statistics", later in this chapter.

If the **any** parameter is not recognized as containing encoded process statistics values, exception 1000 (*Invalid parameter type*) is raised, exception 1002 (*Invalid parameter value*) is raised, or exception 1137 (*An internal data packet inconsistency was detected*) is raised. For example, this could happen if the **any** parameter is not the **userInfo** part of a notification received in response to a **sendRequestStatistics** request. Exception 1000 (*Invalid parameter type*) is also raised if the value specified in the **localOrRemote** parameter is invalid or does not match the information encoded in the **any** parameter.

For examples of the use of this method, see the **Process** class **extractRequestStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Process::*getCallStackInfo* Method

Signature `getCallStackInfo(): String;`

The **Process** class **getCallStackInfo** method retrieves information about the call stack of the executing process.

The return value contains environmental details, in addition to the local and remote call stacks for the executing process and method source lines. The location of execution for each method is signified at the end of the method line as **(C)** for a client node or **(S)** for a server node or single user mode, followed by the source code line for each method.

A process can use only its own **Process** instance as the method receiver. Using any other **Process** instance causes a 1265 exception (*Environmental object operation is out of scope for process*) to be raised. In addition, this exception is raised when some statistical functions (for example, getting the cache statistics of a node) cannot be carried out on the node stub of the secondary database server in an SDS environment. (The node stub represents the primary server node, and it does not have the full functionality of a normal client.)

You can use the value returned by the **Node** class **nodeRole** method to distinguish the node stub from standard client nodes. For the node stub, the returned value is the **Node** class **Role_Replay** constant (as opposed to the **Role_Standard** constant).

For an example of the information returned by the **getCallStackInfo** method, see the **Process** class **getCallStackInfo** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Process::*sendCallStackInfo* Method

Signature `sendCallStackInfo();`

The **Process** class **sendCallStackInfo** method requests a target process to send one or more notifications containing information about the call stack of the receiver process. (See also "Process::*getCallStackInfo* Method", in the previous section.)

The target process is indicated by the **Process** instance used as the method receiver. The target object for the notification is the **Process** instance of the process making the request.

You can specify any current process as the target, including the requesting process itself and processes executing on other nodes.

To retrieve the call stack information and send the notifications, the target process is temporarily activated or interrupted. After sending the notifications, the target process resumes from the point at which it was activated or interrupted.

Note This method is asynchronous; that is, the **sendCallStackInfo** method does not wait until the information has been received. The information is received as notifications some time after the **sendCallStackInfo** method has been called.

The information in the notification relating to the process call stack is listed in the following table.

Parameter	Contains …
eventType	**Process_Call_Stack_Info_Event** global constant in the **JadeProcessEvents** category
target	Process instance of the process that made the request
userInfo	Call stack information stored within a **String** value

The process making the request should register to receive type **Process_Call_Stack_Info_Event** notifications on its **Process** instance, using the **beginNotification** method defined in the **Object** class before executing the **sendCallStackInfo** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Call_Stack_Info_Event,
                       Response_Continuous, 0);
```

To determine whether a notification contains call stack information, the **userNotification** method receiving the notification should test whether the value of the **eventType** parameter is **Process_Call_Stack_Info_Event**.

The size of the call stack information collected can be greater than the maximum allowed for notifications. When this situation occurs, the information is broken into parts and sent via multiple notifications.

Each delivered notification records positional information at the end of the string, in the following the format.

```
[process-oid:current-notification:total-number-of-notifications]
```

The following is an example of the positional information.

```
[187.5:1:3]
```

If the target process (the method receiver) is not a valid current process, exception 1128 (*The target process is not valid*) is raised.

Process::*sendTransientFileAnalysis* Method

Signature `sendTransientFileAnalysis();`

The **Process** class **sendTransientFileAnalysis** method requests a target process to send one or more notifications containing detailed analysis of the transient database file, including counts of objects by class number and other useful information. (For details, see the **Process** class **analyzeTransientFileUsage** method.)

The target process is indicated by the **Process** instance used as the method receiver. The target object for the notification is the **Process** instance of the process making the request.

You can specify any current process as the target, including the requesting process itself and processes executing on other nodes.

To analyze the transient database file and send the notifications, the target process is temporarily activated or interrupted. After sending the notifications, the target process resumes from the point at which it was activated or interrupted.

Note This method is asynchronous; that is, the **sendTransientFileAnalysis** method does not wait until the information has been received. The information is received as notifications some time after the **sendTransientFileAnalysis** method has been called.

The information in the notification relating to the analysis of the transient database file of the process is listed in the following table.

Parameter	Contains …
eventType	**Process_TDB_Analysis_Event** global constant in the **JadeProcessEvents** category
target	Process instance of the process that made the request
userInfo	Detailed analysis of the transient database file of the process

The process making the request should register to receive type **Process_TDB_Analysis_Event** notifications on its **Process** instance, using the **beginNotification** method defined in the **Object** class before executing the **sendTransientFileAnalysis** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_TDB_Analysis_Event,
                       Response_Continuous, 0);
```

To determine whether a notification contains transient database file analysis, the **userNotification** receiving the notification should test whether the value of the **eventType** parameter is **Process_TDB_Analysis_Event**.

The size of the analyzed data collected can be greater than the maximum allowed for notifications. When this situation occurs, the information is broken into parts and sent via multiple notifications.

Each delivered notification records positional information at the end of the string, in the following the format.

```
[process-oid:current-notification:total-number-of-notifications]
```

The following is an example of the positional information.

```
[187.5:1:3]
```

If the target process (the method receiver) is not a valid current process, exception 1128 (*The target process is not valid*) is raised.

Process::*sendTransientFileInfo* Method

Signature `sendTransientFileInfo();`

The **Process** class **sendTransientFileInfo** method requests a target process to send a notification containing information about its transient database file.

The target process is indicated by the **Process** instance used as the method receiver. The target object for the notification is the **Process** instance of the process making the request.

You can specify any current process as the target, including the requesting process itself and processes executing on other nodes.

To retrieve the transient database file information and send the notification, the target process is temporarily activated or interrupted. After sending the notification, the target process resumes from the point at which it was activated or interrupted.

Note This method is asynchronous; that is, the **sendTransientFileInfo** method does not wait until the information has been received. The information is received as notifications some time after the **sendTransientFileInfo** method has been called.

The information in the notification relating to the transient database file information of the process is listed in the following table.

Parameter	Contains …
eventType	**Process_TDB_Info_Event** global constant in the **JadeProcessEvents** category
target	Process instance of the process that made the request
userInfo	Colon-delimited string containing the process oid, transient database file name, and the transient database file length

The process making the request should register to receive type **Process_TDB_Info_Event** notifications on its **Process** instance, using the **beginNotification** method defined in the **Object** class before executing the **sendTransientFileInfo** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_TDB_Info_Event,
                       Response_Continuous, 0);
```

To determine whether a notification contains transient database file information, the **userNotification** method receiving the notification should test whether the value of the **eventType** parameter is **Process_TDB_Info_Event**.

If the target process (the method receiver) is not a valid current process, exception 1128 (*The target process is not valid*) is raised

System Request Statistics Method

A JADE system can use the **System** class **getRequestStats** method to return statistics relating to persistent database activity for all nodes. For details, see the following subsection.

System::*getRequestStats* Method

Signature `getRequestStats(jdo: JadeDynamicObject input);`

The **System** class **getRequestStats** method returns statistics relating to persistent database activity. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The system statistics are held on the database server node.

The properties returned in the dynamic object are listed in the following table.

Property	Description	Example
committedTransactions	Total number of transactions committed	14
abortedTransactions	Total number of transactions aborted	0
getObjects	Total number of requests to retrieve an object from the database	41561
queuedLocks	Total number of lock requests that were queued	0
createObjects	Total number of created objects	433
deleteObjects	Total number of deleted objects	160
updateObjects	Total number of object updates	703
lockObjects	Total number of lock requests	22444
unlockObjects	Total number of unlock requests	12310
beginNotifications	Total number of **beginNotification** instructions	686
endNotifications	Total number of **endNotification** instructions	33
deliveredNotifications	Total number of notification deliveries	169
serverMethodExecutions	Total number of **serverExecution** methods executed	0
totalLockQueueWaitTime	Total amount of time processes spent waiting to obtain locks (in milliseconds)	0
causeEvents	Total number of **causeEvent** instructions	70

Properties are added to the object when the method is first called. You can then use the object 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.

For an example of the use of the **getRequestStats** method, see the **System** class **getRequestStats** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Timer Information

A process can use the **getTimers** method of the **Process** class to obtain timer information for any process. For details, see the following subsection.

Process::*getTimers* Method

Signature
```
getTimers(jdoa:  JadeDynamicObjectArray;
          count: Integer output);
```

The **Process** class **getTimers** method populates a user-supplied JADE dynamic object with timer-related information from the receiving process that is the method receiver.

The **count** parameter contains the total number of current timers for the process. If a process has more than 51 current timers, the count is the total number found but only the first 51 are returned.

The values are returned as properties of members inserted into the dynamic object array specified by the **jdoa** parameter. The properties returned in the JADE dynamic objects are listed in the following table.

Property	Type	Description
name	String	CurrentTimers.
type	Integer	117.
receiver	Object	Oid of the timer event method receiver.
eventTag	Integer	Event tag of the timer.
delay	Integer	Specified delay period of the timer, in milliseconds.
option	Integer	Option used when the timer was started; that is, **Timer_Continuous** (0) or **Timer_One_Shot** (1).
fired	Boolean	**true** if the timer has been fired and the timer event method is still in progress or has not yet been invoked.
inCallBack	Boolean	**true** if the timer event method is in progress.
exceptionState	Boolean	**true** if the target process is in exception state.
remainingTime	Integer	Number of milliseconds left until the timer is next fired.
Application	Object	Application of the process that was in effect when the timer was started.
interfaceNumber	Integer	Number of the **JadeInterface** specified when the timer was started (with the **Object** class **beginTimerForIF** method). Zero (**0**) indicates no interface.
serverExecution	Boolean	**true** if the timer was started in a **serverExecution** method.

Members are created and added to the object array when the method is first called. When used in subsequent class, the array is purged before new members are added. The calling process is responsible for purging and deleting the JADE dynamic object array when it is no longer required.

The JADE Monitor uses this method to display a table in the **Timers** view in the Process Information activity group.

Note If a timer was started in a **serverExecution** method, the timer event method is not called when the timer fires. The **serverExecution** attribute returned in the dynamic object array indicates this circumstance.

Timer event methods are not called while the process is in exception state but are deferred until the process is no longer in exception state. The **exceptionState** attribute returned in the dynamic object array indicates when this is the case.

Getting RPC Statistics

You can obtain Remote Procedure Call (RPC) statistics that relate to activity between the database server node and client nodes (standard clients and application servers). The statistics record the number of different requests and request data sizes, which can provide an indication of the amount of network activity between the database server node and client nodes.

You can retrieve RPC statistics from the following levels.

- Node, representing totals for requests from a specified node
- Process, representing totals for requests from a specified process
- System, representing totals for all requests from all nodes

As the statistics are maintained on the database server node, they record requests and replies sent between the database server node and client nodes. You can retrieve the statistics as combined totals for all request types or as individual totals for each request type.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance.

For details, see the following subsections.

Node::*getRpcServerStatistics* Method

Signature
```
getRpcServerStatistics(jdo:      JadeDynamicObject input
                       detailed: Boolean);
```

The **Node** class **getRpcServerStatistics** method retrieves RPC statistics relating to activity between the database server node and the client node represented by the **Node** instance used as the method receiver.

The values returned represent information about the connection to the specified node and totals for requests received and replies sent to it.

The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The **detailed** parameter specifies whether the values returned should include individual totals for each request type.

If the value of the **detailed** parameter is **false**, the properties listed in the following table are returned in the dynamic object.

Property	Type	Description
timeStarted	TimeStamp	The time at which the connection to the client node was established
connectionType	Integer	Zero (**0**) for TCP/IP, **1** for shared memory
lastInboundRequest	TimeStamp	The time at which the most recent request from the client node was received

Property	Type	Description
requestsFromClients	Integer64	The total number of requests from the client node
repliesToClients	Integer64	The total number of replies sent to the client node
requestPacketsFromClients	Integer64	The total number of separate request packets received from the client node
replyPacketsToClients	Integer64	The total number of separate reply packets sent to the client node
requestBytesFromClients	Integer64	The total number of bytes received for requests from the client node
replyBytesToClients	Integer64	The total number of bytes sent for replies to the client node
requestsToClients	Integer64	The total number of requests sent to the client node
repliesFromClients	Integer64	The total number of replies received from the client node
requestPacketsToClients	Integer64	The total number of separate request packets sent to the client node
replyPacketsFromClients	Integer64	The total number of separate reply packets received from the client node
requestBytesToClients	Integer64	The total number of bytes sent for requests to the client node
replyBytesFromClients	Integer64	The total number of bytes received for replies from the client node
notificationPacketsToClients	Integer64	The total number of separate packets sent to the client node for notification delivery
notificationBytesToClients	Integer64	The total number of bytes sent to the client node for notification delivery

If the value of the **detailed** parameter is **true**, the additional properties listed in the following table are returned in the dynamic object.

Property	Type	Description
— Number of requests from the client node —		
reqsFromClnt_initialize	Integer64	Node initialize
reqsFromClnt_finalize	Integer64	Node finalize
reqsFromClnt_signOn	Integer64	Sign on process
reqsFromClnt_signOff	Integer64	Sign off process
reqsFromClnt_chgAccessMode	Integer64	Change access mode
reqsFromClnt_setSchema	Integer64	Set schema
reqsFromClnt_setExecInsts	Integer64	Set exec instances
reqsFromClnt_openFile	Integer64	Open a file
reqsFromClnt_closeFile	Integer64	Close a file
reqsFromClnt_beginTrans	Integer64	beginTransaction
reqsFromClnt_endTrans	Integer64	commitTransaction

Property	Type	Description
reqsFromClnt_abortTrans	Integer64	abortTransaction
reqsFromClnt_getOid	Integer64	Get an oid
reqsFromClnt_getEdition	Integer64	Get an edition
reqsFromClnt_getObject	Integer64	Get an object
reqsFromClnt_getObjGroup	Integer64	Get a group of objects
reqsFromClnt_putObject	Integer64	Send object to database
reqsFromClnt_lockObject	Integer64	Lock an object
reqsFromClnt_lockObjGroup	Integer64	Lock a group of objects
reqsFromClnt_unlockObject	Integer64	Unlock an object
reqsFromClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsFromClnt_removeLocks	Integer64	Remove all locks for a process
reqsFromClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsFromClnt_sendDbMsg	Integer64	Special database request
reqsFromClnt_beginNote	Integer64	beginNotification
reqsFromClnt_returnNotes	Integer64	Send pending notifications
reqsFromClnt_queryNote	Integer64	Query notifications
reqsFromClnt_endNote	Integer64	endNotification
reqsFromClnt_causeEvent	Integer64	causeEvent
reqsFromClnt_acquireSchema	Integer64	Acquire a schema
reqsFromClnt_releaseSchema	Integer64	Release a schema
reqsFromClnt_authRequest	Integer64	Authorization request
reqsFromClnt_authChallenge	Integer64	Authorization challenge
reqsFromClnt_authResponse	Integer64	Authorization response
reqsFromClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsFromClnt_acquireClass	Integer64	Acquire a class
reqsFromClnt_releaseClass	Integer64	Release a class
reqsFromClnt_canUpdateClass	Integer64	Query if class can be updated
reqsFromClnt_beginLoad	Integer64	Begin load state
reqsFromClnt_endLoad	Integer64	End load state
reqsFromClnt_beginLock	Integer64	Begin lock state
reqsFromClnt_endLock	Integer64	End lock state
reqsFromClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsFromClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsFromClnt_getObjBranches	Integer64	Versioning – get object branches
reqsFromClnt_getBranchObjs	Integer64	Versioning – get branch objects

Property	Type	Description
reqsFromClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsFromClnt_getLastBranch	Integer64	Versioning – get last branch
reqsFromClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsFromClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsFromClnt_mgmtRequest	Integer64	Special administrative request
reqsFromClnt_unknown	Integer64	Unknown request type
— Number of replies to the client node —		
repliesToClnt_initialize	Integer64	Node initialize
repliesToClnt_finalize	Integer64	Node finalize
repliesToClnt_signOn	Integer64	Sign on process
repliesToClnt_signOff	Integer64	Sign off process
repliesToClnt_chgAccessMode	Integer64	Change access mode
repliesToClnt_setSchema	Integer64	Set schema
repliesToClnt_setExecInsts	Integer64	Set exec instances
repliesToClnt_openFile	Integer64	Open a file
repliesToClnt_closeFile	Integer64	Close a file
repliesToClnt_beginTrans	Integer64	beginTransaction
repliesToClnt_endTrans	Integer64	commitTransaction
repliesToClnt_abortTrans	Integer64	abortTransaction
repliesToClnt_getOid	Integer64	Get an oid
repliesToClnt_getEdition	Integer64	Get an edition
repliesToClnt_getObject	Integer64	Get an object
repliesToClnt_getObjGroup	Integer64	Get a group of objects
repliesToClnt_putObject	Integer64	Send object to database
repliesToClnt_lockObject	Integer64	Lock an object
repliesToClnt_lockObjGroup	Integer64	Lock a group of objects
repliesToClnt_unlockObject	Integer64	Unlock an object
repliesToClnt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesToClnt_removeLocks	Integer64	Remove all locks for a process
repliesToClnt_sendMsg	Integer64	Send a message, that is execute a method
repliesToClnt_sendDbMsg	Integer64	Special database request
repliesToClnt_beginNote	Integer64	beginNotification
repliesToClnt_returnNotes	Integer64	Send pending notifications
repliesToClnt_queryNote	Integer64	Query notifications
repliesToClnt_endNote	Integer64	endNotification

Property	Type	Description
repliesToClnt_causeEvent	Integer64	causeEvent
repliesToClnt_acquireSchema	Integer64	Acquire a schema
repliesToClnt_releaseSchema	Integer64	Release a schema
repliesToClnt_authRequest	Integer64	Authorization request
repliesToClnt_authChallenge	Integer64	Authorization challenge
repliesToClnt_authResponse	Integer64	Authorization response
repliesToClnt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesToClnt_acquireClass	Integer64	Acquire a class
repliesToClnt_releaseClass	Integer64	Release a class
repliesToClnt_canUpdateClass	Integer64	Query if class can be updated
repliesToClnt_beginLoad	Integer64	Begin load state
repliesToClnt_endLoad	Integer64	End load state
repliesToClnt_beginLock	Integer64	Begin lock state
repliesToClnt_endLock	Integer64	End lock state
repliesToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesToClnt_commitTrans	Integer64	Prepare to commit a transaction
repliesToClnt_getObjBranches	Integer64	Versioning – get object branches
repliesToClnt_getBranchObjs	Integer64	Versioning – get branch objects
repliesToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesToClnt_getLastBranch	Integer64	Versioning – get last branch
repliesToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
repliesToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesToClnt_mgmtRequest	Integer64	Special administrative request
repliesToClnt_unknown	Integer64	Unknown request type
— Total elapsed time for processing requests from the client node (in microseconds) —		
reqsElapsedTime _initialize	Integer64	Node initialize
reqsElapsedTime _finalize	Integer64	Node finalize
reqsElapsedTime _signOn	Integer64	Sign on process
reqsElapsedTime _signOff	Integer64	Sign off process
reqsElapsedTime _chgAccessMode	Integer64	Change access mode
reqsElapsedTime _setSchema	Integer64	Set schema
reqsElapsedTime _setExecInsts	Integer64	Set exec instances
reqsElapsedTime _openFile	Integer64	Open a file
reqsElapsedTime _closeFile	Integer64	Close a file
reqsElapsedTime _beginTrans	Integer64	beginTransaction

Property	Type	Description
reqsElapsedTime _endTrans	Integer64	commitTransaction
reqsElapsedTime _abortTrans	Integer64	abortTransaction
reqsElapsedTime _getOid	Integer64	Get an oid
reqsElapsedTime _getEdition	Integer64	Get an edition
reqsElapsedTime _getObject	Integer64	Get an object
reqsElapsedTime _getObjGroup	Integer64	Get a group of objects
reqsElapsedTime _putObject	Integer64	Send object to database
reqsElapsedTime _lockObject	Integer64	Lock an object
reqsElapsedTime _lockObjGroup	Integer64	Lock a group of objects
reqsElapsedTime _unlockObject	Integer64	Unlock an object
reqsElapsedTime _UnlockObjGroup	Integer64	Unlock a group of objects
reqsElapsedTime _removeLocks	Integer64	Remove all locks for a process
reqsElapsedTime _sendMsg	Integer64	Send a message, that is execute a method
reqsElapsedTime _sendDbMsg	Integer64	Special database request
reqsElapsedTime _beginNote	Integer64	beginNotification
reqsElapsedTime _returnNotes	Integer64	Send pending notifications
reqsElapsedTime _queryNote	Integer64	Query notifications
reqsElapsedTime _endNote	Integer64	endNotification
reqsElapsedTime _causeEvent	Integer64	causeEvent
reqsElapsedTime _acquireSchema	Integer64	Acquire a schema
reqsElapsedTime _releaseSchema	Integer64	Release a schema
reqsElapsedTime _authRequest	Integer64	Authorization request
reqsElapsedTime _authChallenge	Integer64	Authorization challenge
reqsElapsedTime _authResponse	Integer64	Authorization response
reqsElapsedTime _getReplyCont	Integer64	Retrieve next portion of large reply
reqsElapsedTime _acquireClass	Integer64	Acquire a class
reqsElapsedTime _releaseClass	Integer64	Release a class
reqsElapsedTime _canUpdateClass	Integer64	Query if class can be updated
reqsElapsedTime _beginLoad	Integer64	Begin load state
reqsElapsedTime _endLoad	Integer64	End load state
reqsElapsedTime _beginLock	Integer64	Begin lock state
reqsElapsedTime _endLock	Integer64	End lock state
reqsElapsedTime _flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsElapsedTime _commitTrans	Integer64	Prepare to commit a transaction
reqsElapsedTime _getObjBranches	Integer64	Versioning – get object branches

Property	Type	Description
reqsElapsedTime _getBranchObjs	Integer64	Versioning – get branch objects
reqsElapsedTime _getBranchObjOid	Integer64	Versioning – get branch object oid
reqsElapsedTime _getLastBranch	Integer64	Versioning – get last branch
reqsElapsedTime _getLatestVnOid	Integer64	Versioning – get latest version oid
reqsElapsedTime _getLatestEdtn	Integer64	Versioning – get latest edition
reqsElapsedTime _mgmtRequest	Integer64	Special administrative request
reqsElapsedTime _unknown	Integer64	Unknown request type
— Number of requests sent to the client node —		
reqsToClnt_initialize	Integer64	Node initialize
reqsToClnt_finalize	Integer64	Node finalize
reqsToClnt_signOn	Integer64	Sign on process
reqsToClnt_signOff	Integer64	Sign off process
reqsToClnt_chgAccessMode	Integer64	Change access mode
reqsToClnt_setSchema	Integer64	Set schema
reqsToClnt_setExecInsts	Integer64	Set exec instances
reqsToClnt_openFile	Integer64	Open a file
reqsToClnt_closeFile	Integer64	Close a file
reqsToClnt_beginTrans	Integer64	beginTransaction
reqsToClnt_endTrans	Integer64	commitTransaction
reqsToClnt_abortTrans	Integer64	abortTransaction
reqsToClnt_getOid	Integer64	Get an oid
reqsToClnt_getEdition	Integer64	Get an edition
reqsToClnt_getObject	Integer64	Get an object
reqsToClnt_getObjGroup	Integer64	Get a group of objects
reqsToClnt_putObject	Integer64	Send object to database
reqsToClnt_lockObject	Integer64	Lock an object
reqsToClnt_lockObjGroup	Integer64	Lock a group of objects
reqsToClnt_unlockObject	Integer64	Unlock an object
reqsToClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsToClnt_removeLocks	Integer64	Remove all locks for a process
reqsToClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsToClnt_sendDbMsg	Integer64	Special database request
reqsToClnt_beginNote	Integer64	beginNotification
reqsToClnt_returnNotes	Integer64	Send pending notifications
reqsToClnt_queryNote	Integer64	Query notifications

Property	Type	Description
reqsToClnt_endNote	Integer64	endNotification
reqsToClnt_causeEvent	Integer64	causeEvent
reqsToClnt_acquireSchema	Integer64	Acquire a schema
reqsToClnt_releaseSchema	Integer64	Release a schema
reqsToClnt_authRequest	Integer64	Authorization request
reqsToClnt_authChallenge	Integer64	Authorization challenge
reqsToClnt_authResponse	Integer64	Authorization response
reqsToClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsToClnt_acquireClass	Integer64	Acquire a class
reqsToClnt_releaseClass	Integer64	Release a class
reqsToClnt_canUpdateClass	Integer64	Query if class can be updated
reqsToClnt_beginLoad	Integer64	Begin load state
reqsToClnt_endLoad	Integer64	End load state
reqsToClnt_beginLock	Integer64	Begin lock state
reqsToClnt_endLock	Integer64	End lock state
reqsToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsToClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsToClnt_getObjBranches	Integer64	Versioning – get object branches
reqsToClnt_getBranchObjs	Integer64	Versioning – get branch objects
reqsToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsToClnt_getLastBranch	Integer64	Versioning – get last branch
reqsToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsToClnt_mgmtRequest	Integer64	Special administrative request
reqsToClnt_unknown	Integer64	Unknown request type
— Total number of replies received from the client node —		
repliesFromClnt_initialize	Integer64	Node initialize
repliesFromClnt_finalize	Integer64	Node finalize
repliesFromClnt_signOn	Integer64	Sign on process
repliesFromClnt_signOff	Integer64	Sign off process
repliesFromClnt_chgAccessMode	Integer64	Change access mode
repliesFromClnt_setSchema	Integer64	Set schema
repliesFromClnt_setExecInsts	Integer64	Set exec instances
repliesFromClnt_openFile	Integer64	Open a file
repliesFromClnt_closeFile	Integer64	Close a file

Property	Type	Description
repliesFromCInt_beginTrans	Integer64	beginTransaction
repliesFromCInt_endTrans	Integer64	commitTransaction
repliesFromCInt_abortTrans	Integer64	abortTransaction
repliesFromCInt_getOid	Integer64	Get an oid
repliesFromCInt_getEdition	Integer64	Get an edition
repliesFromCInt_getObject	Integer64	Get an object
repliesFromCInt_getObjGroup	Integer64	Get a group of objects
repliesFromCInt_putObject	Integer64	Send object to database
repliesFromCInt_lockObject	Integer64	Lock an object
repliesFromCInt_lockObjGroup	Integer64	Lock a group of objects
repliesFromCInt_unlockObject	Integer64	Unlock an object
repliesFromCInt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesFromCInt_removeLocks	Integer64	Remove all locks for a process
repliesFromCInt_sendMsg	Integer64	Send a message, that is execute a method
repliesFromCInt_sendDbMsg	Integer64	Special database request
repliesFromCInt_beginNote	Integer64	beginNotification
repliesFromCInt_returnNotes	Integer64	Send pending notifications
repliesFromCInt_queryNote	Integer64	Query notifications
repliesFromCInt_endNote	Integer64	endNotification
repliesFromCInt_causeEvent	Integer64	causeEvent
repliesFromCInt_acquireSchema	Integer64	Acquire a schema
repliesFromCInt_releaseSchema	Integer64	Release a schema
repliesFromCInt_authRequest	Integer64	Authorization request
repliesFromCInt_authChallenge	Integer64	Authorization challenge
repliesFromCInt_authResponse	Integer64	Authorization response
repliesFromCInt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesFromCInt_acquireClass	Integer64	Acquire a class
repliesFromCInt_releaseClass	Integer64	Release a class
repliesFromCInt_canUpdateClass	Integer64	Query if class can be updated
repliesFromCInt_beginLoad	Integer64	Begin load state
repliesFromCInt_endLoad	Integer64	End load state
repliesFromCInt_beginLock	Integer64	Begin lock state
repliesFromCInt_endLock	Integer64	End lock state
repliesFromCInt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesFromCInt_commitTrans	Integer64	Prepare to commit a transaction

Property	Type	Description
repliesFromCInt_getObjBranches	Integer64	Versioning – get object branches
repliesFromCInt_getBranchObjs	Integer64	Versioning – get branch objects
repliesFromCInt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesFromCInt_getLastBranch	Integer64	Versioning – get last branch
repliesFromCInt_getLatestVnOid	Integer64	Versioning – get latest version oid
repliesFromCInt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesFromCInt_mgmtRequest	Integer64	Special administrative request
repliesFromCInt_unknown	Integer64	Unknown request type

Properties are added to the dynamic object when the method is first called. The object can then be used in subsequent calls.

If the dynamic object passed to the method already contains properties that do not match the properties to be returned, the existing dynamic object properties are removed and replaced with the appropriate properties.

The method is most efficient when the properties match those to be returned.

For an example of the use of the **getRpcServerStatistics** method, see the **Node** class **getRpcServerStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Process::*getRpcServerStatistics* Method

Signature `getRpcServerStatistics(jdo: JadeDynamicObject input; detailed: Boolean);`

The **Process** class **getRpcServerStatistics** method retrieves RPC statistics relating to requests and replies sent to and from the database server node and the process specified by receiver object.

The returned values represent information relating to the connection between client node and database server node, and requests and replies between the nodes for the specified process. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The **detailed** parameter specifies whether the values include individual totals for each request type.

If the value of the **detailed** parameter is **false**, the properties listed in the following table are returned in the dynamic object.

Property	Type	Description
timeStarted	TimeStamp	The time at which the connection to the process was established
connectionType	Integer	Zero (**0**) for TCP/IP, **1** for Shared Memory
lastInboundRequest	TimeStamp	The time at which the most recent request from the process was received
requestsFromClients	Integer64	The total number of requests from the process
repliesToClients	Integer64	The total number of replies sent to the process
requestPacketsFromClients	Integer64	The total number of separate request packets received from the process

Property	Type	Description
replyPacketsToClients	Integer64	The total number of separate reply packets sent to the process
requestBytesFromClients	Integer64	The total number of bytes received for requests from the process
replyBytesToClients	Integer64	The total number of bytes sent for replies to the process
requestsToClients	Integer64	The total number of requests sent to the process
repliesFromClients	Integer64	The total number of replies received from the process
requestPacketsToClients	Integer64	The total number of separate request packets sent to the process
replyPacketsFromClients	Integer64	The total number of separate reply packets received from the process
requestBytesToClients	Integer64	The total number of bytes sent for requests to the process
replyBytesFromClients	Integer64	The total number of bytes received for replies from the process
notificationPacketsToClients	Integer64	The total number of separate packets sent to the process for notification delivery
notificationBytesToClients	Integer64	The total number of bytes sent to the process for notification delivery

If the value of the **detailed** parameter is **true**, the additional properties listed in the following table are returned in the dynamic object.

Property	Type	Description
— Number of requests from the client node —		
reqsFromClnt_initialize	Integer64	Node initialize
reqsFromClnt_finalize	Integer64	Node finalize
reqsFromClnt_signOn	Integer64	Sign on process
reqsFromClnt_signOff	Integer64	Sign off process
reqsFromClnt_chgAccessMode	Integer64	Change access mode
reqsFromClnt_setSchema	Integer64	Set schema
reqsFromClnt_setExecInsts	Integer64	Set exec instances
reqsFromClnt_openFile	Integer64	Open a file
reqsFromClnt_closeFile	Integer64	Close a file
reqsFromClnt_beginTrans	Integer64	beginTransaction
reqsFromClnt_endTrans	Integer64	commitTransaction
reqsFromClnt_abortTrans	Integer64	abortTransaction
reqsFromClnt_getOid	Integer64	Get an oid
reqsFromClnt_getEdition	Integer64	Get an edition
reqsFromClnt_getObject	Integer64	Get an object
reqsFromClnt_getObjGroup	Integer64	Get a group of objects

Property	Type	Description
reqsFromClnt_putObject	Integer64	Send object to database
reqsFromClnt_lockObject	Integer64	Lock an object
reqsFromClnt_lockObjGroup	Integer64	Lock a group of objects
reqsFromClnt_unlockObject	Integer64	Unlock an object
reqsFromClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsFromClnt_removeLocks	Integer64	Remove all locks for a process
reqsFromClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsFromClnt_sendDbMsg	Integer64	Special database request
reqsFromClnt_beginNote	Integer64	beginNotification
reqsFromClnt_returnNotes	Integer64	Send pending notifications
reqsFromClnt_queryNote	Integer64	Query notifications
reqsFromClnt_endNote	Integer64	endNotification
reqsFromClnt_causeEvent	Integer64	causeEvent
reqsFromClnt_acquireSchema	Integer64	Acquire a schema
reqsFromClnt_releaseSchema	Integer64	Release a schema
reqsFromClnt_authRequest	Integer64	Authorization request
reqsFromClnt_authChallenge	Integer64	Authorization challenge
reqsFromClnt_authResponse	Integer64	Authorization response
reqsFromClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsFromClnt_acquireClass	Integer64	Acquire a class
reqsFromClnt_releaseClass	Integer64	Release a class
reqsFromClnt_canUpdateClass	Integer64	Query if class can be updated
reqsFromClnt_beginLoad	Integer64	Begin load state
reqsFromClnt_endLoad	Integer64	End load state
reqsFromClnt_beginLock	Integer64	Begin lock state
reqsFromClnt_endLock	Integer64	End lock state
reqsFromClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsFromClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsFromClnt_getObjBranches	Integer64	Versioning – get object branches
reqsFromClnt_getBranchObjs	Integer64	Versioning – get branch objects
reqsFromClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsFromClnt_getLastBranch	Integer64	Versioning – get last branch
reqsFromClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsFromClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsFromClnt_mgmtRequest	Integer64	Special administrative request

Property	Type	Description
reqsFromClnt_unknown	Integer64	Unknown request type
— Number of replies to the process —		
repliesToClnt_initialize	Integer64	Node initialize
repliesToClnt_finalize	Integer64	Node finalize
repliesToClnt_signOn	Integer64	Sign on process
repliesToClnt_signOff	Integer64	Sign off process
repliesToClnt_chgAccessMode	Integer64	Change access mode
repliesToClnt_setSchema	Integer64	Set schema
repliesToClnt_setExecInsts	Integer64	Set exec instances
repliesToClnt_openFile	Integer64	Open a file
repliesToClnt_closeFile	Integer64	Close a file
repliesToClnt_beginTrans	Integer64	beginTransaction
repliesToClnt_endTrans	Integer64	commitTransaction
repliesToClnt_abortTrans	Integer64	abortTransaction
repliesToClnt_getOid	Integer64	Get an oid
repliesToClnt_getEdition	Integer64	Get an edition
repliesToClnt_getObject	Integer64	Get an object
repliesToClnt_getObjGroup	Integer64	Get a group of objects
repliesToClnt_putObject	Integer64	Send object to database
repliesToClnt_lockObject	Integer64	Lock an object
repliesToClnt_lockObjGroup	Integer64	Lock a group of objects
repliesToClnt_unlockObject	Integer64	Unlock an object
repliesToClnt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesToClnt_removeLocks	Integer64	Remove all locks for a process
repliesToClnt_sendMsg	Integer64	Send a message, that is execute a method
repliesToClnt_sendDbMsg	Integer64	Special database request
repliesToClnt_beginNote	Integer64	beginNotification
repliesToClnt_returnNotes	Integer64	Send pending notifications
repliesToClnt_queryNote	Integer64	Query notifications
repliesToClnt_endNote	Integer64	endNotification
repliesToClnt_causeEvent	Integer64	causeEvent
repliesToClnt_acquireSchema	Integer64	Acquire a schema
repliesToClnt_releaseSchema	Integer64	Release a schema
repliesToClnt_authRequest	Integer64	Authorization request
repliesToClnt_authChallenge	Integer64	Authorization challenge

Property	Type	Description
repliesToClnt_authResponse	Integer64	Authorization response
repliesToClnt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesToClnt_acquireClass	Integer64	Acquire a class
repliesToClnt_releaseClass	Integer64	Release a class
repliesToClnt_canUpdateClass	Integer64	Query if class can be updated
repliesToClnt_beginLoad	Integer64	Begin load state
repliesToClnt_endLoad	Integer64	End load state
repliesToClnt_beginLock	Integer64	Begin lock state
repliesToClnt_endLock	Integer64	End lock state
repliesToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesToClnt_commitTrans	Integer64	Prepare to commit a transaction
repliesToClnt_getObjBranches	Integer64	Versioning – get object branches
repliesToClnt_getBranchObjs	Integer64	Versioning – get branch objects
repliesToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesToClnt_getLastBranch	Integer64	Versioning – get last branch
repliesToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
repliesToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesToClnt_mgmtRequest	Integer64	Special administrative request
repliesToClnt_unknown	Integer64	Unknown request type
— Total elapsed time for processing requests from the process (in microseconds) —		
reqsElapsedTime _initialize	Integer64	Node initialize
reqsElapsedTime _finalize	Integer64	Node finalize
reqsElapsedTime _signOn	Integer64	Sign on process
reqsElapsedTime _signOff	Integer64	Sign off process
reqsElapsedTime _chgAccessMode	Integer64	Change access mode
reqsElapsedTime _setSchema	Integer64	Set schema
reqsElapsedTime _setExecInsts	Integer64	Set exec instances
reqsElapsedTime _openFile	Integer64	Open a file
reqsElapsedTime _closeFile	Integer64	Close a file
reqsElapsedTime _beginTrans	Integer64	beginTransaction
reqsElapsedTime _endTrans	Integer64	commitTransaction
reqsElapsedTime _abortTrans	Integer64	abortTransaction
reqsElapsedTime _getOid	Integer64	Get an oid
reqsElapsedTime _getEdition	Integer64	Get an edition
reqsElapsedTime _getObject	Integer64	Get an object

Property	Type	Description
reqsElapsedTime _getObjGroup	Integer64	Get a group of objects
reqsElapsedTime _putObject	Integer64	Send object to database
reqsElapsedTime _lockObject	Integer64	Lock an object
reqsElapsedTime _lockObjGroup	Integer64	Lock a group of objects
reqsElapsedTime _unlockObject	Integer64	Unlock an object
reqsElapsedTime _UnlockObjGroup	Integer64	Unlock a group of objects
reqsElapsedTime _removeLocks	Integer64	Remove all locks for a process
reqsElapsedTime _sendMsg	Integer64	Send a message, that is execute a method
reqsElapsedTime _sendDbMsg	Integer64	Special database request
reqsElapsedTime _beginNote	Integer64	beginNotification
reqsElapsedTime _returnNotes	Integer64	Send pending notifications
reqsElapsedTime _queryNote	Integer64	Query notifications
reqsElapsedTime _endNote	Integer64	endNotification
reqsElapsedTime _causeEvent	Integer64	causeEvent
reqsElapsedTime _acquireSchema	Integer64	Acquire a schema
reqsElapsedTime _releaseSchema	Integer64	Release a schema
reqsElapsedTime _authRequest	Integer64	Authorization request
reqsElapsedTime _authChallenge	Integer64	Authorization challenge
reqsElapsedTime _authResponse	Integer64	Authorization response
reqsElapsedTime _getReplyCont	Integer64	Retrieve next portion of large reply
reqsElapsedTime _acquireClass	Integer64	Acquire a class
reqsElapsedTime _releaseClass	Integer64	Release a class
reqsElapsedTime _canUpdateClass	Integer64	Query if class can be updated
reqsElapsedTime _beginLoad	Integer64	Begin load state
reqsElapsedTime _endLoad	Integer64	End load state
reqsElapsedTime _beginLock	Integer64	Begin lock state
reqsElapsedTime _endLock	Integer64	End lock state
reqsElapsedTime _flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsElapsedTime _commitTrans	Integer64	Prepare to commit a transaction
reqsElapsedTime _getObjBranches	Integer64	Versioning – get object branches
reqsElapsedTime _getBranchObjs	Integer64	Versioning – get branch objects
reqsElapsedTime _getBranchObjOid	Integer64	Versioning – get branch object oid
reqsElapsedTime _getLastBranch	Integer64	Versioning – get last branch
reqsElapsedTime _getLatestVnOid	Integer64	Versioning – get latest version oid
reqsElapsedTime _getLatestEdtn	Integer64	Versioning – get latest edition

Property	Type	Description
reqsElapsedTime _mgmtRequest	Integer64	Special administrative request
reqsElapsedTime _unknown	Integer64	Unknown request type
— Number of requests sent to the process —		
reqsToClnt_initialize	Integer64	Node initialize
reqsToClnt_finalize	Integer64	Node finalize
reqsToClnt_signOn	Integer64	Sign on process
reqsToClnt_signOff	Integer64	Sign off process
reqsToClnt_chgAccessMode	Integer64	Change access mode
reqsToClnt_setSchema	Integer64	Set schema
reqsToClnt_setExecInsts	Integer64	Set exec instances
reqsToClnt_openFile	Integer64	Open a file
reqsToClnt_closeFile	Integer64	Close a file
reqsToClnt_beginTrans	Integer64	beginTransaction
reqsToClnt_endTrans	Integer64	commitTransaction
reqsToClnt_abortTrans	Integer64	abortTransaction
reqsToClnt_getOid	Integer64	Get an oid
reqsToClnt_getEdition	Integer64	Get an edition
reqsToClnt_getObject	Integer64	Get an object
reqsToClnt_getObjGroup	Integer64	Get a group of objects
reqsToClnt_putObject	Integer64	Send object to database
reqsToClnt_lockObject	Integer64	Lock an object
reqsToClnt_lockObjGroup	Integer64	Lock a group of objects
reqsToClnt_unlockObject	Integer64	Unlock an object
reqsToClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsToClnt_removeLocks	Integer64	Remove all locks for a process
reqsToClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsToClnt_sendDbMsg	Integer64	Special database request
reqsToClnt_beginNote	Integer64	beginNotification
reqsToClnt_returnNotes	Integer64	Send pending notifications
reqsToClnt_queryNote	Integer64	Query notifications
reqsToClnt_endNote	Integer64	endNotification
reqsToClnt_causeEvent	Integer64	causeEvent
reqsToClnt_acquireSchema	Integer64	Acquire a schema
reqsToClnt_releaseSchema	Integer64	Release a schema
reqsToClnt_authRequest	Integer64	Authorization request

Property	Type	Description
reqsToClnt_authChallenge	Integer64	Authorization challenge
reqsToClnt_authResponse	Integer64	Authorization response
reqsToClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsToClnt_acquireClass	Integer64	Acquire a class
reqsToClnt_releaseClass	Integer64	Release a class
reqsToClnt_canUpdateClass	Integer64	Query if class can be updated
reqsToClnt_beginLoad	Integer64	Begin load state
reqsToClnt_endLoad	Integer64	End load state
reqsToClnt_beginLock	Integer64	Begin lock state
reqsToClnt_endLock	Integer64	End lock state
reqsToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsToClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsToClnt_getObjBranches	Integer64	Versioning – get object branches
reqsToClnt_getBranchObjs	Integer64	Versioning – get branch objects
reqsToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsToClnt_getLastBranch	Integer64	Versioning – get last branch
reqsToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsToClnt_mgmtRequest	Integer64	Special administrative request
reqsToClnt_unknown	Integer64	Unknown request type
— Total number of replies received from the process —		
repliesFromClnt_initialize	Integer64	Node initialize
repliesFromClnt_finalize	Integer64	Node finalize
repliesFromClnt_signOn	Integer64	Sign on process
repliesFromClnt_signOff	Integer64	Sign off process
repliesFromClnt_chgAccessMode	Integer64	Change access mode
repliesFromClnt_setSchema	Integer64	Set schema
repliesFromClnt_setExecInsts	Integer64	Set exec instances
repliesFromClnt_openFile	Integer64	Open a file
repliesFromClnt_closeFile	Integer64	Close a file
repliesFromClnt_beginTrans	Integer64	beginTransaction
repliesFromClnt_endTrans	Integer64	commitTransaction
repliesFromClnt_abortTrans	Integer64	abortTransaction
repliesFromClnt_getOid	Integer64	Get an oid
repliesFromClnt_getEdition	Integer64	Get an edition

Property	Type	Description
repliesFromCInt_getObject	Integer64	Get an object
repliesFromCInt_getObjGroup	Integer64	Get a group of objects
repliesFromCInt_putObject	Integer64	Send object to database
repliesFromCInt_lockObject	Integer64	Lock an object
repliesFromCInt_lockObjGroup	Integer64	Lock a group of objects
repliesFromCInt_unlockObject	Integer64	Unlock an object
repliesFromCInt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesFromCInt_removeLocks	Integer64	Remove all locks for a process
repliesFromCInt_sendMsg	Integer64	Send a message, that is execute a method
repliesFromCInt_sendDbMsg	Integer64	Special database request
repliesFromCInt_beginNote	Integer64	beginNotification
repliesFromCInt_returnNotes	Integer64	Send pending notifications
repliesFromCInt_queryNote	Integer64	Query notifications
repliesFromCInt_endNote	Integer64	endNotification
repliesFromCInt_causeEvent	Integer64	causeEvent
repliesFromCInt_acquireSchema	Integer64	Acquire a schema
repliesFromCInt_releaseSchema	Integer64	Release a schema
repliesFromCInt_authRequest	Integer64	Authorization request
repliesFromCInt_authChallenge	Integer64	Authorization challenge
repliesFromCInt_authResponse	Integer64	Authorization response
repliesFromCInt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesFromCInt_acquireClass	Integer64	Acquire a class
repliesFromCInt_releaseClass	Integer64	Release a class
repliesFromCInt_canUpdateClass	Integer64	Query if class can be updated
repliesFromCInt_beginLoad	Integer64	Begin load state
repliesFromCInt_endLoad	Integer64	End load state
repliesFromCInt_beginLock	Integer64	Begin lock state
repliesFromCInt_endLock	Integer64	End lock state
repliesFromCInt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesFromCInt_commitTrans	Integer64	Prepare to commit a transaction
repliesFromCInt_getObjBranches	Integer64	Versioning – get object branches
repliesFromCInt_getBranchObjs	Integer64	Versioning – get branch objects
repliesFromCInt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesFromCInt_getLastBranch	Integer64	Versioning – get last branch
repliesFromCInt_getLatestVnOid	Integer64	Versioning – get latest version oid

Property	Type	Description
repliesFromCInt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesFromCInt_mgmtRequest	Integer64	Special administrative request
repliesFromCInt_unknown	Integer64	Unknown request type

Properties are added to the dynamic 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.

For an example of the use of the **getRpcServerStatistics** method, see the **Process** class **getRpcServerStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

System::*getRpcServerStatistics* Method

Signature `getRpcServerStatistics(jdo: JadeDynamicObject input
detailed: Boolean);`

The **System** class **getRpcServerStatistics** method retrieves RPC statistics relating to activity between the database server node and all client nodes.

The values returned by this method represent information about the connections between client nodes and the database server, and totals for requests received and replies sent. The values are returned as **Integer64** properties in the dynamic object specified by the **jdo** parameter.

The **detailed** parameter specifies whether the values returned should be combined for all requests, or individual totals for each request type.

If the value of the **detailed** parameter is **false**, the properties listed in the following table are returned in the dynamic object.

Property	Type	Description
timeStarted	TimeStamp	The time at which the database server node was started
connectionType	Integer	Zero (**0**) for TCP/IP, **1** for Shared Memory
lastInboundRequest	TimeStamp	The time at which the most recent request from a client node was received
requestsFromClients	Integer64	The total number of requests from all client nodes
repliesToClients	Integer64	The total number of replies sent to client nodes
requestPacketsFromClients	Integer64	The total number of separate request packets received from all client nodes
replyPacketsToClients	Integer64	The total number of separate reply packets sent to all client nodes
requestBytesFromClients	Integer64	The total number of bytes received for requests from all client nodes
replyBytesToClients	Integer64	The total number of bytes sent for replies to all client nodes
requestsToClients	Integer64	The total number of requests sent to all client nodes

Property	Type	Description
repliesFromClients	Integer64	The total number of replies received from all client nodes
requestPacketsToClients	Integer64	The total number of separate request packets sent to all client nodes
replyPacketsFromClients	Integer64	The total number of separate reply packets received from all client nodes
requestBytesToClients	Integer64	The total number of bytes sent for requests to all client nodes
replyBytesFromClients	Integer64	The total number of bytes received for replies from all client nodes
notificationPacketsToClients	Integer64	The total number of separate packets sent to all client nodes for notification delivery
notificationBytesToClients	Integer64	The total number of bytes sent to all client nodes for notification delivery

If the value of the **detailed** parameter is **true**, the additional properties listed in the following table are returned in the dynamic object.

Property	Type	Description
— Number of requests from client nodes —		
reqsFromClnt_initialize	Integer64	Node initialize
reqsFromClnt_finalize	Integer64	Node finalize
reqsFromClnt_signOn	Integer64	Sign on process
reqsFromClnt_signOff	Integer64	Sign off process
reqsFromClnt_chgAccessMode	Integer64	Change access mode
reqsFromClnt_setSchema	Integer64	Set schema
reqsFromClnt_setExecInsts	Integer64	Set exec instances
reqsFromClnt_openFile	Integer64	Open a file
reqsFromClnt_closeFile	Integer64	Close a file
reqsFromClnt_beginTrans	Integer64	beginTransaction
reqsFromClnt_endTrans	Integer64	commitTransaction
reqsFromClnt_abortTrans	Integer64	abortTransaction
reqsFromClnt_getOid	Integer64	Get an oid
reqsFromClnt_getEdition	Integer64	Get an edition
reqsFromClnt_getObject	Integer64	Get an object
reqsFromClnt_getObjGroup	Integer64	Get a group of objects
reqsFromClnt_putObject	Integer64	Send object to database
reqsFromClnt_lockObject	Integer64	Lock an object
reqsFromClnt_lockObjGroup	Integer64	Lock a group of objects
reqsFromClnt_unlockObject	Integer64	Unlock an object

Property	Type	Description
reqsFromClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsFromClnt_removeLocks	Integer64	remove all locks for a process
reqsFromClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsFromClnt_sendDbMsg	Integer64	Special database request
reqsFromClnt_beginNote	Integer64	beginNotification
reqsFromClnt_returnNotes	Integer64	Send pending notifications
reqsFromClnt_queryNote	Integer64	Query notifications
reqsFromClnt_endNote	Integer64	endNotification
reqsFromClnt_causeEvent	Integer64	causeEvent
reqsFromClnt_acquireSchema	Integer64	Acquire a schema
reqsFromClnt_releaseSchema	Integer64	Release a schema
reqsFromClnt_authRequest	Integer64	Authorization request
reqsFromClnt_authChallenge	Integer64	Authorization challenge
reqsFromClnt_authResponse	Integer64	Authorization response
reqsFromClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsFromClnt_acquireClass	Integer64	Acquire a class
reqsFromClnt_releaseClass	Integer64	Release a class
reqsFromClnt_canUpdateClass	Integer64	Query if class can be updated
reqsFromClnt_beginLoad	Integer64	Begin load state
reqsFromClnt_endLoad	Integer64	End load state
reqsFromClnt_beginLock	Integer64	Begin lock state
reqsFromClnt_endLock	Integer64	End lock state
reqsFromClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsFromClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsFromClnt_getObjBranches	Integer64	Versioning – get object branches
reqsFromClnt_getBranchObjs	Integer64	Versioning – get branch objects
reqsFromClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsFromClnt_getLastBranch	Integer64	Versioning – get last branch
reqsFromClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsFromClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsFromClnt_mgmtRequest	Integer64	Special administrative request
reqsFromClnt_unknown	Integer64	Unknown request type
— Number of replies to client nodes —		
repliesToClnt_initialize	Integer64	Node initialize
repliesToClnt_finalize	Integer64	Node finalize

Property	Type	Description
repliesToCInt_signOn	Integer64	Sign on process
repliesToCInt_signOff	Integer64	Sign off process
repliesToCInt_chgAccessMode	Integer64	Change access mode
repliesToCInt_setSchema	Integer64	Set schema
repliesToCInt_setExecInsts	Integer64	Set exec instances
repliesToCInt_openFile	Integer64	Open a file
repliesToCInt_closeFile	Integer64	Close a file
repliesToCInt_beginTrans	Integer64	beginTransaction
repliesToCInt_endTrans	Integer64	commitTransaction
repliesToCInt_abortTrans	Integer64	abortTransaction
repliesToCInt_getOid	Integer64	Get an oid
repliesToCInt_getEdition	Integer64	Get an edition
repliesToCInt_getObject	Integer64	Get an object
repliesToCInt_getObjGroup	Integer64	Get a group of objects
repliesToCInt_putObject	Integer64	Send object to database
repliesToCInt_lockObject	Integer64	Lock an object
repliesToCInt_lockObjGroup	Integer64	Lock a group of objects
repliesToCInt_unlockObject	Integer64	Unlock an object
repliesToCInt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesToCInt_removeLocks	Integer64	Remove all locks for a process
repliesToCInt_sendMsg	Integer64	Send a message, that is execute a method
repliesToCInt_sendDbMsg	Integer64	Special database request
repliesToCInt_beginNote	Integer64	beginNotification
repliesToCInt_returnNotes	Integer64	Send pending notifications
repliesToCInt_queryNote	Integer64	Query notifications
repliesToCInt_endNote	Integer64	endNotification
repliesToCInt_causeEvent	Integer64	causeEvent
repliesToCInt_acquireSchema	Integer64	Acquire a schema
repliesToCInt_releaseSchema	Integer64	Release a schema
repliesToCInt_authRequest	Integer64	Authorization request
repliesToCInt_authChallenge	Integer64	Authorization challenge
repliesToCInt_authResponse	Integer64	Authorization response
repliesToCInt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesToCInt_acquireClass	Integer64	Acquire a class
repliesToCInt_releaseClass	Integer64	Release a class

Property	Type	Description
repliesToClnt_canUpdateClass	Integer64	Query if class can be updated
repliesToClnt_beginLoad	Integer64	Begin load state
repliesToClnt_endLoad	Integer64	End load state
repliesToClnt_beginLock	Integer64	Begin lock state
repliesToClnt_endLock	Integer64	End lock state
repliesToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesToClnt_commitTrans	Integer64	Prepare to commit a transaction
repliesToClnt_getObjBranches	Integer64	Versioning – get object branches
repliesToClnt_getBranchObjs	Integer64	Versioning – get branch objects
repliesToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesToClnt_getLastBranch	Integer64	Versioning – get last branch
repliesToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
repliesToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesToClnt_mgmtRequest	Integer64	Special administrative request
repliesToClnt_unknown	Integer64	Unknown request type
— Total elapsed time for processing requests from client nodes (microseconds) —		
reqsElapsedTime _initialize	Integer64	Node initialize
reqsElapsedTime _finalize	Integer64	Node finalize
reqsElapsedTime _signOn	Integer64	Sign on process
reqsElapsedTime _signOff	Integer64	Sign off process
reqsElapsedTime _chgAccessMode	Integer64	Change access mode
reqsElapsedTime _setSchema	Integer64	Set schema
reqsElapsedTime _setExecInsts	Integer64	Set exec instances
reqsElapsedTime _openFile	Integer64	Open a file
reqsElapsedTime _closeFile	Integer64	Close a file
reqsElapsedTime _beginTrans	Integer64	beginTransaction
reqsElapsedTime _endTrans	Integer64	commitTransaction
reqsElapsedTime _abortTrans	Integer64	abortTransaction
reqsElapsedTime _getOid	Integer64	Get an oid
reqsElapsedTime _getEdition	Integer64	Get an edition
reqsElapsedTime _getObject	Integer64	Get an object
reqsElapsedTime _getObjGroup	Integer64	Get a group of objects
reqsElapsedTime _putObject	Integer64	Send object to database
reqsElapsedTime _lockObject	Integer64	Lock an object
reqsElapsedTime _lockObjGroup	Integer64	Lock a group of objects

Property	Type	Description
reqsElapsedTime _unlockObject	Integer64	Unlock an object
reqsElapsedTime _UnlockObjGroup	Integer64	Unlock a group of objects
reqsElapsedTime _removeLocks	Integer64	Remove all locks for a process
reqsElapsedTime _sendMsg	Integer64	Send a message, that is execute a method
reqsElapsedTime _sendDbMsg	Integer64	Special database request
reqsElapsedTime _beginNote	Integer64	beginNotification
reqsElapsedTime _returnNotes	Integer64	Send pending notifications
reqsElapsedTime _queryNote	Integer64	Query notifications
reqsElapsedTime _endNote	Integer64	endNotification
reqsElapsedTime _causeEvent	Integer64	causeEvent
reqsElapsedTime _acquireSchema	Integer64	Acquire a schema
reqsElapsedTime _releaseSchema	Integer64	Release a schema
reqsElapsedTime _authRequest	Integer64	Authorization request
reqsElapsedTime _authChallenge	Integer64	Authorization challenge
reqsElapsedTime _authResponse	Integer64	Authorization response
reqsElapsedTime _getReplyCont	Integer64	Retrieve next portion of large reply
reqsElapsedTime _acquireClass	Integer64	Acquire a class
reqsElapsedTime _releaseClass	Integer64	Release a class
reqsElapsedTime _canUpdateClass	Integer64	Query if class can be updated
reqsElapsedTime _beginLoad	Integer64	Begin load state
reqsElapsedTime _endLoad	Integer64	End load state
reqsElapsedTime _beginLock	Integer64	Begin lock state
reqsElapsedTime _endLock	Integer64	End lock state
reqsElapsedTime _flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsElapsedTime _commitTrans	Integer64	Prepare to commit a transaction
reqsElapsedTime _getObjBranches	Integer64	Versioning – get object branches
reqsElapsedTime _getBranchObjs	Integer64	Versioning – get branch objects
reqsElapsedTime _getBranchObjOid	Integer64	Versioning – get branch object oid
reqsElapsedTime _getLastBranch	Integer64	Versioning – get last branch
reqsElapsedTime _getLatestVnOid	Integer64	Versioning – get latest version oid
reqsElapsedTime _getLatestEdtn	Integer64	Versioning – get latest edition
reqsElapsedTime _mgmtRequest	Integer64	Special administrative request
reqsElapsedTime _unknown	Integer64	Unknown request type
— Number of requests to client nodes —		
reqsToClnt_initialize	Integer64	Node initialize

Property	Type	Description
reqsToClnt_finalize	Integer64	Node finalize
reqsToClnt_signOn	Integer64	Sign on process
reqsToClnt_signOff	Integer64	Sign off process
reqsToClnt_chgAccessMode	Integer64	Change access mode
reqsToClnt_setSchema	Integer64	Set schema
reqsToClnt_setExecInsts	Integer64	Set exec instances
reqsToClnt_openFile	Integer64	Open a file
reqsToClnt_closeFile	Integer64	Close a file
reqsToClnt_beginTrans	Integer64	beginTransaction
reqsToClnt_endTrans	Integer64	commitTransaction
reqsToClnt_abortTrans	Integer64	abortTransaction
reqsToClnt_getOid	Integer64	Get an oid
reqsToClnt_getEdition	Integer64	Get an edition
reqsToClnt_getObject	Integer64	Get an object
reqsToClnt_getObjGroup	Integer64	Get a group of objects
reqsToClnt_putObject	Integer64	Send object to database
reqsToClnt_lockObject	Integer64	Lock an object
reqsToClnt_lockObjGroup	Integer64	Lock a group of objects
reqsToClnt_unlockObject	Integer64	Unlock an object
reqsToClnt_UnlockObjGroup	Integer64	Unlock a group of objects
reqsToClnt_removeLocks	Integer64	Remove all locks for a process
reqsToClnt_sendMsg	Integer64	Send a message, that is execute a method
reqsToClnt_sendDbMsg	Integer64	Special database request
reqsToClnt_beginNote	Integer64	beginNotification
reqsToClnt_returnNotes	Integer64	Send pending notifications
reqsToClnt_queryNote	Integer64	Query notifications
reqsToClnt_endNote	Integer64	endNotification
reqsToClnt_causeEvent	Integer64	causeEvent
reqsToClnt_acquireSchema	Integer64	Acquire a schema
reqsToClnt_releaseSchema	Integer64	Release a schema
reqsToClnt_authRequest	Integer64	Authorization request
reqsToClnt_authChallenge	Integer64	Authorization challenge
reqsToClnt_authResponse	Integer64	Authorization response
reqsToClnt_getReplyCont	Integer64	Retrieve next portion of large reply
reqsToClnt_acquireClass	Integer64	Acquire a class

Property	Type	Description
reqsToClnt_releaseClass	Integer64	Release a class
reqsToClnt_canUpdateClass	Integer64	Query if class can be updated
reqsToClnt_beginLoad	Integer64	Begin load state
reqsToClnt_endLoad	Integer64	End load state
reqsToClnt_beginLock	Integer64	Begin lock state
reqsToClnt_endLock	Integer64	End lock state
reqsToClnt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
reqsToClnt_commitTrans	Integer64	Prepare to commit a transaction
reqsToClnt_getObjBranches	Integer64	Versioning – get object branches
reqsToClnt_getBranchObjs	Integer64	Versioning – get branch objects
reqsToClnt_getBranchObjOid	Integer64	Versioning – get branch object oid
reqsToClnt_getLastBranch	Integer64	Versioning – get last branch
reqsToClnt_getLatestVnOid	Integer64	Versioning – get latest version oid
reqsToClnt_getLatestEdtn	Integer64	Versioning – get latest edition
reqsToClnt_mgmtRequest	Integer64	Special administrative request
reqsToClnt_unknown	Integer64	Unknown request type
— Total number of replies from client nodes —		
repliesFromClnt_initialize	Integer64	Node initialize
repliesFromClnt_finalize	Integer64	Node finalize
repliesFromClnt_signOn	Integer64	Sign on process
repliesFromClnt_signOff	Integer64	Sign off process
repliesFromClnt_chgAccessMode	Integer64	Change access mode
repliesFromClnt_setSchema	Integer64	Set schema
repliesFromClnt_setExecInsts	Integer64	Set exec instances
repliesFromClnt_openFile	Integer64	Open a file
repliesFromClnt_closeFile	Integer64	Close a file
repliesFromClnt_beginTrans	Integer64	beginTransaction
repliesFromClnt_endTrans	Integer64	commitTransaction
repliesFromClnt_abortTrans	Integer64	abortTransaction
repliesFromClnt_getOid	Integer64	Get an oid
repliesFromClnt_getEdition	Integer64	Get an edition
repliesFromClnt_getObject	Integer64	Get an object
repliesFromClnt_getObjGroup	Integer64	Get a group of objects
repliesFromClnt_putObject	Integer64	Send object to database
repliesFromClnt_lockObject	Integer64	Lock an object

Property	Type	Description
repliesFromCInt_lockObjGroup	Integer64	Lock a group of objects
repliesFromCInt_unlockObject	Integer64	Unlock an object
repliesFromCInt_UnlockObjGroup	Integer64	Unlock a group of objects
repliesFromCInt_removeLocks	Integer64	Remove all locks for a process
repliesFromCInt_sendMsg	Integer64	Send a message, that is execute a method
repliesFromCInt_sendDbMsg	Integer64	Special database request
repliesFromCInt_beginNote	Integer64	beginNotification
repliesFromCInt_returnNotes	Integer64	Send pending notifications
repliesFromCInt_queryNote	Integer64	Query notifications
repliesFromCInt_endNote	Integer64	endNotification
repliesFromCInt_causeEvent	Integer64	causeEvent
repliesFromCInt_acquireSchema	Integer64	Acquire a schema
repliesFromCInt_releaseSchema	Integer64	Release a schema
repliesFromCInt_authRequest	Integer64	Authorization request
repliesFromCInt_authChallenge	Integer64	Authorization challenge
repliesFromCInt_authResponse	Integer64	Authorization response
repliesFromCInt_getReplyCont	Integer64	Retrieve next portion of large reply
repliesFromCInt_acquireClass	Integer64	Acquire a class
repliesFromCInt_releaseClass	Integer64	Release a class
repliesFromCInt_canUpdateClass	Integer64	Query if class can be updated
repliesFromCInt_beginLoad	Integer64	Begin load state
repliesFromCInt_endLoad	Integer64	End load state
repliesFromCInt_beginLock	Integer64	Begin lock state
repliesFromCInt_endLock	Integer64	End lock state
repliesFromCInt_flushBuffObjs	Integer64	Force buffered objects to be sent to the database
repliesFromCInt_commitTrans	Integer64	Prepare to commit a transaction
repliesFromCInt_getObjBranches	Integer64	Versioning – get object branches
repliesFromCInt_getBranchObjs	Integer64	Versioning – get branch objects
repliesFromCInt_getBranchObjOid	Integer64	Versioning – get branch object oid
repliesFromCInt_getLastBranch	Integer64	Versioning – get last branch
repliesFromCInt_getLatestVnOid	Integer64	Versioning – get latest version oid
repliesFromCInt_getLatestEdtn	Integer64	Versioning – get latest edition
repliesFromCInt_mgmtRequest	Integer64	Special administrative request
repliesFromCInt_unknown	Integer64	Unknown request type

Properties are added to the dynamic 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.

For an example of the use of the **getRpcServerStatistics** method, see the **System** class **getRpcServerStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Getting Web Statistics

You can asynchronously obtain World Wide Web (or Internet activity) statistics for a specific process, by using the **Process** class **sendWebStatistics** method in conjunction with the **extractWebStatistics** method.

To obtain Web statistics, a process calls the:

1. **Object** class **beginNotification** method, to register receive the notification containing the statistics
2. **Process** class **sendWebStatistics** method, to request the target process to send a notification containing statistics
3. **Process** class **extractWebStatistics** method in the **Object** class **userNotification** method, to retrieve the statistics

For details, see the following subsections.

Process::*sendWebStatistics* Method

Signature `sendWebStatistics();`

The **Process** class **sendWebStatistics** method requests a process to send a notification containing World Wide Web performance statistics. The process is indicated by the **Process** instance used as the method receiver.

Any current process can be specified, including the requesting process itself and processes executing on other nodes. However, only processes that are using Web services send meaningful data. Other processes send the notification, but it does not contain any values.

The target object for the notification is the **Process** instance of the process making the request.

The target process is temporarily activated or interrupted in order to send the notification. After sending the notification, it resumes whatever it was previously doing.

Note This method is asynchronous; that is, the **sendWebStatistics** method does not wait until the statistics have been received. The statistics are received as a notification some time after the **sendWebStatistics** method has been called.

The information in the notification relating to the Web performance statistics is shown in the following table.

Parameter	Contains …
eventType	**Process_Web_Stats_Event** in the **JadeProcessEvents** category
target	Process instance of the process that made the request
userInfo	Statistical values encoded within a Binary value

The process making the request should register to receive type **Process_Web_Stats_Event** notifications on its **Process** instance using the **beginNotification** method defined in the **Object** class before executing the **sendWebStatistics** method, as shown in the following code fragment.

```
self.beginNotification(process, Process_Web_Stats_Event,
                       Response_Continuous, 0);
```

To test whether a notification contains Web statistics information, the **userNotification** method of its **Process** instance should test whether the value of the **eventType** parameter is **Process_Web_Stats_Event**, which indicates a Web statistics notification.

The **userInfo** parameter of the notification should be passed as a parameter to the **extractWebStatistics** method, to extract the Web statistics as properties in a **JadeDynamicObject** instance.

If the target process (the method receiver) is not a valid current process, exception 1128 (*The target process is not valid*) is raised.

Process::*extractWebStatistics* Method

Signature

```
extractWebStatistics(proc: Process output;
                     jdo: JadeDynamicObject input;
                     any: Any);
```

The **Process** class **extractWebStatistics** method extracts Web performance statistics from the **userInfo** part of notifications sent in response to **sendWebStatistics** method requests, defined in the **Process** class.

The extracted statistics are inserted as properties in a **JadeDynamicObject** instance.

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

Parameter	Description
proc	An output parameter that receives a reference to the **Process** instance to which the statistics relate
jdo	A **JadeDynamicObject** instance into which the statistics values are placed as properties
any	The value of the **userInfo** parameter of the notification that was received

The calling process is responsible for creating the **JadeDynamicObject** instance that is used as the **jdo** parameter. Any existing properties that the instance has are cleared every time the method is called. If the process that sent the notification is not using Web services, no properties are added to the **JadeDynamicObject** instance.

If the process is using Web services, the properties listed in the following table are added.

Property	Type
maximumResponseTime	Integer64 (milliseconds)
minimumResponseTime	Integer64 (milliseconds)
totalRequests	Integer64
totalResponseTime	Integer64 (milliseconds)
rejectedRequests	Integer64

If the **any** parameter is not recognized as containing encoded Web statistics values, exception 1000 (*Invalid parameter type*) is raised, exception 1002 (*Invalid parameter value*) is raised, or exception 1137 (*An internal data packet inconsistency was detected*) is raised. This could happen if the **any** parameter is not the **userInfo** part of a notification received in response to a **sendWebStatistics** request.

For examples of using methods for a form to obtain and display information about Web statistics, see the **Process** class **extractWebStatistics** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Node::*wbemListClasses* Method

Signature `wbemListClasses(hsa: HugeStringArray input);`

The **Node** class **wbemListClasses** method retrieves a list of the Web-Based Enterprise Management (WBEM) classes that can be queried for the host machine on which the node specified by the receiver object is running. This is a subset of the full WBEM classes available, as JADE allows only a subset of classes to be queried.

The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information.

The method inserts strings containing the allowed class names into the huge string array specified by the **hsa** parameter. This method always empties the array before inserting the class names. The caller is responsible for creating and deleting this array.

The strings that are inserted into the huge string array parameter are fully qualified WBEM class names that can be used directly as class names for the other WBEM methods provided by the **Node** class.

For an example of the use of and output from this method, see the **Node** class **wbemListClasses** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Node::*wbemListInstanceNames* Method

Signature
```
wbemListInstanceNames(className:     String;
                      instNameArray: HugeStringArray input);
```

The **Node** class **wbemListInstanceNames** method retrieves the names of all instances of a Web-Based Enterprise Management (WBEM) class specified by the **className** parameter for the host machine for the node specified by the receiver object is running.

The class name must be a fully qualified WBEM class name.

The instance names are inserted as strings into the huge string array specified by the **instNameArray** parameter. This method always empties the array before inserting the instance names. The caller is responsible for creating and deleting this array.

JADE allows only a subset of the available WBEM classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information.

The **wbemListClasses** method of the **Node** class can be used to retrieve the fully qualified WBEM class names that can be used. For an example of the use of and output from this method, see the **Node** class **wbemListInstanceNames** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Node::*wbemQueryQualifiers* Method

Signature
```
wbemQueryQualifiers(className:      String;
                    attributeNames: StringArray input;
                    counterTypes:   IntegerArray input;
                    scaleFactors:   IntegerArray input);
```

The **Node** class **wbemQueryQualifiers** method retrieves the names, type, and scale factor for each property of the Web-Based Enterprise Management (WBEM) class specified by the **className** parameter. This allows property values returned by the **wbemRetrieveData** method defined in the **Node** class to be correctly interpreted.

The qualifier information is placed into three matched arrays. Information for the first property is placed into the first member of each of the three arrays, information for the second property is placed into the second member, and so on.

The caller is responsible for creating and deleting the three arrays. This method always empties these arrays before inserting the qualifier information.

The class name must be a fully qualified WBEM class name. JADE allows only a subset of the available WBEM classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information. The **wbemListClasses** method of the **Node** class can be used to retrieve the fully qualified WBEM class names that can be used.

If a name that is not allowed or recognized is used, an 1136 exception (*Attempt to use a prohibited WBEM interface class name*) or an 1133 exception (*WBEM interface error when querying data*) is raised.

The string array specified by the **attributeNames** parameter contains the name of each property for the specified class. These match the names of the properties that the **wbemRetrieveData** method creates in the **JadeDynamicObject** it uses.

The integer array specified by the **counterTypes** parameter contains performance counter type values for the properties. The values are those defined by Microsoft and documented in the MSDN (Microsoft Developer Network) literature.

The integer array specified by the **scaleFactors** parameter contains the default scale factor to be applied to the property values. This is a power of 10 that can be used to estimate the likely range of the value.

The meaning of each counter type value and the correct way to extract meaningful information from the property values is described in the MSDN literature. Searching using "WMI Performance Counter Types" should locate the relevant information.

For an example of the use of and output from this method, see the **Node** class **wbemQueryQualifiers** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Node::*wbemRetrieveData* Method

Signature
```
wbemRetrieveData(className:     String;
                 instNameArray: HugeStringArray;
                 jdoArray:      JadeDynamicObjectArray input);
```

The **Node** class **wbemRetrieveData** method retrieves Web-Based Enterprise Management (WBEM) instances and property values for a specified WBEM class. The values are retrieved from the machine on which the node specified by the receiver object is running.

JADE allows only a subset of classes to be used. The allowed classes are those relating to cache, memory, system, processor, server, disk, and network interface information.

A **JadeDynamicObject** instance is created for each WBEM instance that is retrieved and added to the instance of the **JadeDynamicObjectArray** specified by the **jdoArray** input parameter. If the array is transient, the **JadeDynamicObject** instances are transient (not shared). If the array is persistent, the instances are persistent.

The caller is responsible for creating and deleting the **JadeDynamicObjectArray** instance, and for deleting any **JadeDynamicObject** instances that are added to it.

The **instNameArray** parameter is used to select the set of WBEM instances that are retrieved. If used, the array should contain a set of strings representing the names of the WBEM instances to be retrieved. Only WBEM instances that have names that match entries in the array are returned. If the value of the **instNameArray** parameter is null, all instances for the class specified by the **className** parameter are returned.

This method does not clear or purge the **JadeDynamicObjectArray** before inserting the **JadeDynamicObject** instances. Therefore, if called multiple times without first calling the **purge** or **clear** methods, previously added entries will remain in the array.

Each **JadeDynamicObject** that is created contains properties representing each property of the corresponding WBEM instance. The name of each property matches the WBEM class property name.

The property value is one of the following types, depending on the corresponding WBEM property type:

- **Integer64**
- **String**
- **Real**
- **Decimal**
- **Boolean**

JADE converts WBEM properties that are arrays into individual properties with the array index inserted at the end of each property name.

For an example of the use of and output from this method, see the **Node** class **wbemRetrieveData** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Some properties are not returned by JADE. These are mainly properties that pertain to WBEM class and superclass names.

There is a limit of approximately 48K bytes to the size of the WBEM data that can be retrieved from remote computers. If this limit is exceeded, an 1141 exception (*WBEM data exceeded maximum allowable size*) is raised. If this exception occurs, the **instNameArray** parameter should be used to restrict the number of WBEM class instances that are retrieved.

If a name that is not allowed or recognized is used, an 1136 exception (*Attempt to use a prohibited WBEM interface class name*) or an 1133 exception (*WBEM interface error when querying data*) is raised.

Mutex Contention Count

The **Node** class provides a method that enables you to determine the number of contentions on mutexes for the receiver node. A *mutex* is a locking mechanism used internally to ensure thread safety when executing critical sections of code. (This information is for use mainly by the Jade Software Corporation.)

Node::*getMutexCounts* Method

Signature `getMutexCounts(jdo: JadeDynamicObject input;`
`includeZeroContentions: Boolean);`

The **Node** class **getMutexCounts** method retrieves the number of contentions on mutexes used internally by JADE for the node identified as the method receiver.

The contention counts are returned as **Integer64** properties of the **JadeDynamicObject** instance specified by the **jdo** parameter. The name of each property represents the internal mutex name, and the value represents the number of times that mutex has been contended (that is, the number of times execution of a thread has been temporarily suspended because another thread was executing in a section of code protected by the mutex).

The contention counts are cumulative from the time the specified node is initiated.

The **includeZeroContentions** parameter indicates whether mutexes that have not yet encountered any contentions should be included in the returned information. If you set this parameter to **false**, only information for those mutexes that have had at least one contention are added to the dynamic object. If you set it to **true**, information about all current mutexes is added.

The calling process is responsible for creating and deleting the **JadeDynamicObject** instance. The **getMutexCounts** method clears any existing properties from the **JadeDynamicObject** instance each time it is called.

Because mutexes can be dynamically created and deleted, the number of mutexes reported and the order in which the properties are added to the dynamic object may vary from call to call.

Note If a mutex is deleted then recreated between calls to the **getMutexCounts** method, the contention count can appear to reduce in value. Any application attempting to calculate contention count differences should therefore take this into account.

The mutex contention information is primarily for internal use by the Jade Software Corporation. High mutex contention counts may indicate bottlenecks that are impacting overall system performance.

For an example of the use of and output from this method, see the **getMutexCounts** method in Volume 2 of the *JADE Encyclopaedia of Classes*.

Invoking an Operating System Process Dump

You can obtain a process dump of a specific node or all nodes attached to a database server and the server node itself.

Node::*processDump* Method

Signature `processDump;`

The **Node** class **processDump** method invokes a non-fatal operating system process dump of the node specified by the receiver.

System::*processDumpAllNodes* Method

Signature `processDumpAllNodes;`

The **System** class **processDumpAllNodes** method invokes a near-simultaneous operating system process dump of all nodes attached to a database server and the server node itself.

Recording Lock Contention Information

You can record information about lock contentions; that is, the number of times lock requests on individual persistent objects have had to be rejected or queued because the object was already locked. Lock contention information is recorded:

- On the database server node
- Only for persistent objects

The information recorded includes the number of contentions, the total time processes have spent waiting for a lock on each object, and the maximum time spent waiting for each lock.

By default, lock contentions are not recorded.

Lock contention information is returned in instances of the **LockContentionInfo** class.

Only one process at a time can be in control of lock contention recording; that is, if a process has called the **beginLockContentionStats** method to start lock contention recording, no other process can begin, end, or clear contention recording. However, other processes can retrieve the lock contention information.

The method in the following example shows the use of the **System** class lock contentions methods.

```
showLockContentions();
vars
    oa : ObjectArray;
    o : Object;
    lci : LockContentionInfo;
    ts : TimeStamp;
    avgWaitTime : Decimal[10,1];
begin
    create oa transient;
    system.beginLockContentionStats(10000);
    process.sleep(60000); //record 1 minute of lock activity
    system.getLockContentionStats(oa, 10000, 5, ts);
    write "FIRST MINUTE";
    foreach o in oa do
        lci := o.LockContentionInfo;
        write CrLf & "Object= " & lci.target.String;
        write "Contentions= " & lci.totalContentions.String;
        avgWaitTime := (lci.totalWaitTime / lci.totalContentions);
        write "Average wait time= " & avgWaitTime.String;
    endforeach;
    oa.purge;
    //Repeat
    system.clearLockContentionStats();
    process.sleep(60000); //record 1 minute of lock activity
    system.getLockContentionStats(oa, 10000, 5, ts);
    write CrLf & "SECOND MINUTE";
    foreach o in oa do
        lci := o.LockContentionInfo;
        write CrLf & "Object= " & lci.target.String;
        write "Contentions= " & lci.totalContentions.String;
        avgWaitTime := (lci.totalWaitTime / lci.totalContentions);
        write "Average wait time= " & avgWaitTime.String;
    endforeach;
    oa.purge;
    system.endLockContentionStats();
```

```
epilog
    delete oa;
end;
```

For details, see the following subsections.

LockContentionInfo Class

The transient **LockContentionInfo** class is used to return information about lock contentions (that is, when an attempt to lock a persistent object is queued or rejected because the object is already locked). When lock contention recording is started, lock contention information is recorded on the database server node.

Use the **System** class **beginLockContentionStats**, **clearLockContentionStats**, and **endLockContentionStats** methods to start and stop recording lock contentions and the **System** class **getLockContentionStats** and **getLockContentionInfo** methods to retrieve recorded lock contention information.

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

Property	Description
maxWaitTime	The longest time in milliseconds that any process spent queued waiting to obtain a lock on the object
totalContentions	The number of lock contentions recorded for the object
totalWaitTime	The total time in milliseconds that all processes spent queued waiting to obtain locks on the object

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

Method	Description
target	Returns a reference to the object to which the lock contention information relates

For details about:

- The **LockContentionInfo** class, see Volume 2 of the *JADE Encyclopaedia of Classes*
- Locking, see Chapter 6 of the *JADE Developer's Reference*

System::*beginLockContentionStats* Method

Signature `beginLockContentionStats(tableSize: Integer);`

The **System** class **beginLockContentionStats** method starts recording lock contentions for persistent objects. A lock contention occurs when an attempt to lock a persistent object is queued because the object is already locked.

After recording has been initiated, you can call the **getLockContentionStats** method to retrieve lock contention information. Call the **endLockContentionStats** method to stop the recording of lock contentions.

The **tableSize** parameter enables you to specify the maximum number of individual contended objects that can be recorded. When the first contention for an object is noted, the object is added to the table of contentions, provided that the maximum table size has not been reached. If the table has reached the maximum size, contentions for objects not found in the table are grouped together in a single entry identified by a null object identifier; that is, the class number and instance number are both set to zero (**0**).

As a guideline, the size of each entry is approximately 40 bytes, so 25,000 entries would consume approximately 1M byte of memory.

Only one process at a time can control the recording of lock contentions. If a process executes the **beginLockContentionStats** method when recording of lock contentions has already been initiated by another process, exception 1131 (*Another process is currently in control of lock contention statistics*) is raised. However, processes other than the one that initiated lock contention recording can retrieve lock contention information, although the information may be cleared or become unavailable at any time.

If the process that started to record lock contentions terminates without having called the **endLockContentionStats** method to stop lock contention recording, the recording of lock contentions is automatically ended.

System::*endLockContentionStats* Method

Signature `endLockContentionStats();`

The **System** class **endLockContentionStats** method stops recording lock contentions for persistent objects and removes all lock contention data.

Only the process that started lock contention recording can use this method. If any other process attempts to use this method, exception 1131 (*Another process is currently in control of lock contention statistics*) is raised.

If the process that started the recording of lock contentions terminates without having called **endLockContentionStats** to stop lock contention recording, the lock contention recording is automatically ended.

System::*getLockContentionStats* Method

Signature
```
getLockContentionStats(oa:             ObjectArray input;
                       maxEntries:     Integer;
                       minContentions: Integer;
                       startTime:      TimeStamp output);
```

The **System** class **getLockContentionStats** method retrieves lock contention information. A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked. The information includes the number of lock contentions for individual objects and the average and maximum times spent waiting to acquire a lock on each individual object. Information is returned in transient instances of the **LockContentionInfo** class, added to the transient **ObjectArray** instance specified by the **oa** parameter.

The value of the **maxEntries** parameter specifies the maximum number of entries to be returned. Returned entries are added to the array in no particular order. When the **maxEntries** limit is reached, no more entries are added.

The value of the **minContentions** parameter specifies the minimum number of contentions for entries to be returned. Only entries with contention counts greater than or equal to the specified minimum are returned.

The **startTime** parameter is an output parameter that is set to the time at which the lock contention recording was started or restarted. This value enables you to calculate the number of contentions per second.

The calling method is responsible for creating and deleting the transient **ObjectArray** instance and for deleting **LockContentionInfo** instances in the array (for example, by using the **purge** method on the **ObjectArray** instance before deleting it). When the method is called, any existing **LockContentionInfo** instances in the array are not removed. New **LockContentionInfo** instances are added to the end of the array.

If lock contentions are not being recorded when the **getLockContentionStats** method is called, no entries are added to the **ObjectArray** instance. You can use the **queryLockContentionStats** method to determine if lock contentions are currently being recorded.

The information available in **LockContentionInfo** instances is listed in the following table.

Feature	Description
target	Returns a reference to the object being locked
totalContentions	Contains the number of lock contentions recorded for the object
maxWaitTime	Contains the longest time in milliseconds that any process spent queued waiting to obtain a lock on the object
totalWaitTime	Contains the total time in milliseconds that all processes spent queued waiting to obtain locks on the object

Note A **LockContentionInfo** instance with a null object reference for the **target** value indicates that it holds combined information for all contentions that occurred on objects that could not be included in the table because the maximum table size had been reached.

System::*clearLockContentionStats* Method

Signature `clearLockContentionStats();`

The **System** class **clearLockContentionStats** method removes all existing lock contention data and restarts recording of lock contentions. A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked.

The lock contention table is cleared, but retains the same maximum size. Use this method for recording lock contention activity over set periods without having to end and begin lock contention recording multiple times.

Only the process that started lock contention recording can call this method. If any other process attempts to use this method, exception 1131 (*Another process is currently in control of lock contention statistics*) is raised.

If lock contention recording is not active when this method is called, it has no effect.

System::*queryLockContentionStats* Method

Signature
```
queryLockContentionStats(active:       Boolean output;
                         startingProc: Process output;
                         maxEntries:   Integer output;
                         startTime:    TimeStamp output);
```

The **System** class **queryLockContentionStats** method retrieves information about the current recording of lock contentions. A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked.

The information returned in the output parameters is listed in the following table.

Parameter	Description
active	Indicates if lock contention recording is currently active
startingProc	A reference to the Process that initiated lock contention recording
maxEntries	The maximum table size specified when lock contention recording was started, which determines the maximum number of individual objects that can have contentions recorded
startTime	The time at which lock contention recording was started or restarted

Note The recording of lock contentions is under the control of the process that initiated the recording; that is, only that process can stop or restart recording of lock contentions. Although any process can use the **queryLockContentionStats** method, it should be assume that recording can stop or restart at any time, if it is not the process that started the recording.

System::*getLockContentionInfo* Method

Signature

```
getLockContentionInfo(obj:       Object;
                      lci:       LockContentionInfo input;
                      startTime: TimeStamp output);
```

The **System** class **getLockContentionInfo** method returns lock contention information for a single object specified by the **obj** parameter. A lock contention occurs when an attempt to lock a persistent object is queued or rejected because the object is already locked.

The information is copied into properties of the **LockContentionInfo** instance specified by the **lci** parameter. The calling process is responsible for creating and deleting the **LockContentionInfo** instance.

The **startTime** parameter is an output parameter that receives the date and time at which lock contention recording was started or restarted.

The information available in **LockContentionInfo** instances is listed in the following table.

Feature	Description
target	Returns a reference to the object being locked
totalContentions	Contains the number of lock contentions recorded for the object
maxWaitTime	Contains the longest time in milliseconds that any process spent queued waiting to obtain a lock on the object
totalWaitTime	Contains the total time in milliseconds that all processes spent queued waiting to obtain locks on the object

If there have been no lock contentions for the specified object, the values of the totalContentions, maxWaitTime, and totalWaitTime property values are set to zero (**0**).

If this method is called when lock contentions are not being recorded, the **startTime** parameter and information in the **LockContentionInfo** instance are set to zero (**0**) values.

Use the **beginLockContentionStats** and **endLockContentionStats** methods to control the recording of lock contentions.

Instrumentation and Diagnosis Settings

The JADE initialization file contains the:

- [JadeMonitor] section, which stores JADE Monitor directives and preferences so that they can be re-used in your next work session.
- [JadeMonitorBackground] section, which contains parameters that specify the directives required to run the non-GUI JADE Monitor as a background process to log selected statistics into a flat file for subsequent analysis.

For details about these sections and the parameters that they can contain, see the *JADE Initialization File Reference*.

The parameters in the [JadeMonitor] section for the JADE Monitor application are listed in the following table.

Parameter	Example	Description
BackgroundWindow	0, 0, 450, 300, 0	Position and size of the client MDI window
DefaultLogDirectory	c:\jade\monitor\logs	Absolute path of directory to which samples are logged
DirectivesFile	c:\jade\bin\directives.txt	Text (.**txt**) file that contains sets of statistics displayed, recorded, and so on, when the JADE Monitor is invoked
NavigatorClickCausesRefresh	false	Specifies whether clicking in the Navigator pane causes a refresh action
ShowOverview	true	Specifies whether the overview is displayed below a JADE Monitor sheet
UIFontSize	10	Point size of the font displayed when the JADE Monitor is invoked

The parameters in the [JadeMonitorBackground] section for the non-GUI background JADE Monitor application are listed in the following table.

Parameter	Example	Description
DefaultLogDirectory	c:\jade\logs	Absolute path of directory to which samples are logged
DirectivesFile	c:\jade\bin\directives.txt	Text (.**txt**) file that contains sets of statistics displayed, recorded, and so on, when the background JADE Monitor is invoked

You can start the JADE Monitor background application by:

- Using the predefined application (that is, ***ServerApplication<application-number>*** **= JadeMonitorSchema,JadeMonitorBackground**) in the [JadeServer] section of the JADE initialization file on the server node to start the background application when the server node starts up
- Selecting the **Start Monitor Background Application** command from the File menu in the GUI JADE Monitor

Note The JADE Monitor background application does nothing if a directives file containing the statistics to be recorded has not yet been saved to the path and file specified in the **DirectivesFile** parameter in the [JadeMonitorBackground] section of the JADE initialization file (that is, if the directives file is empty, it does not exist, or it is not specified, the monitor application is launched and then waits for user input).

Node Sampling

The sampling of activities in JADE applications is produced at the node level. The JADE Object Manager invokes specified entry points in a user-provided dynamic library during its normal execution.

The user dynamic library can then process the information provided by the JADE Object Manager. For example, you could capture data to flat disk files for subsequent processing using a tool of your choice to obtain the operational profiles that you require.

The activities that you can sample enable you to understand the execution characteristics of runtime applications so that you can enhance application performance according to different hardware resources.

A JADE node can initiate sampling on any other node in the system. For example, a client node can sample statistics on another client node and the server node, or a server node can gather statistics on every client node that is attached to it.

When the JADE-supplied library is used, samples are output to your specified flat file, which must be visible to the node being sampled. These samples can then be gathered for subsequent analysis to meet your requirements.

Note You can use the JADE Monitor to analyze the contents of your node sampling files. For details, see the *JADE Monitor User's Guide*.

For details, see the following subsections.

Capturing Sampling Data

Directives to the JADE Object Manager for the invocation characteristics of the user library are provided during normal JADE execution. The available types of sampling, described in the following sections, are:

- Automatic, initiated by setting the **SamplingNode** parameter in the [JadeClient] or [JadeServer] section of your JADE initialization file to **true**.
- Manual local, initiated by using a set of methods in the **Node** class. This type of sampling assumes that if you have knowledge of what the application is doing internally, you can modify the application itself to direct the sampling of activities. For details, see "Node Class External Methods" or "Using the JADE Sampling Application", respectively, later in this chapter.
- Remote, initiated by using:
  - A set of methods in the **System** class.
  - Extended facilities in the JADE Sampling application.

  For details, see "System Class External Methods" or "Using the JADE Sampling Application", respectively, later in this chapter.

You can initiate more than one sampling context simultaneously in any node. Each set of directives relates to a sampling context. You can specify a different user library for each sampling context. For details, see "Sampling Library Interface", later in this chapter.

JADE provides the **filesmpl** and **tcpsmpl** standard libraries, which use the same interface that is available to you. The **filesmpl** library produces a flat file for each sampling context. The file contains a record for every invocation made to it by the JADE Object Manager. The **tcpsmpl** library outputs sample statistics to a TCP/IP connection and can read filter commands from a connection. Both libraries can read filter commands from a connection.

In addition, JADE provides a full project with the source code for a comprehensive sampling library. This chapter describes the format for those records and refers to them when illustrating the kind of information that is available to you.

JADE Initialization File

The [JadeClient] or [JadeServer] sections of the JADE initialization file contain parameters that enable you to automate the sampling of statistics on the server node or client nodes the next time your JADE system is initiated.

Note Unless you set the **SamplingNode** parameter to **true**, none of the other parameters described in the following subsections have any meaning.

These parameters are described in the following subsections.

IndividualLocalRequests

Value Type Boolean

Default False

The **IndividualLocalRequests** parameter, when set to **true**, causes all requests from the node to its local database to invoke the corresponding entry point in the user library.

For details about the individual requests that are produced by the JADE sampling libraries in record type **14**, see "Individual Local Requests" under "Statistics File Format", later in this chapter.

IndividualPersistentCacheActivities

Value Type Boolean

Default False

The **IndividualPersistentCacheActivities** parameter, when set to **true**, causes all activities in the persistent object cache of the node to invoke the corresponding entry point in the user library.

For details about the individual activities that are produced by the JADE Sampling libraries in record type **2**, see "Cache Buffer Activity" under "Statistics File Format", later in this chapter.

IndividualRemoteRequests

Value Type Boolean

Default False

The **IndividualRemoteRequests** parameter, when set to **true**, causes all remote requests from the client node to the server node to invoke the corresponding entry point in the user library.

For details about the individual requests that are produced by the JADE sampling libraries in record type **10**, see "Individual Remote Requests" under "Statistics File Format", later in this chapter.

IndividualRemoteTransientCacheActivities

Value Type Boolean

Default False

The **IndividualRemoteTransientCacheActivities** parameter in the [JadeServer] section, when set to **true**, causes all activities in the remote transient object cache of the node to invoke the corresponding entry point in the user library.

Note This parameter applies only to server nodes; that is, it is not valid in the [JadeClient] section.

For details about the individual activities that are produced by the JADE Sampling libraries in record type **2**, see "Cache Buffer Activity" under "Statistics File Format", later in this chapter.

IndividualTransientCacheActivities

Value Type Boolean

Default False

The **IndividualTransientCacheActivities** parameter, when set to **true**, causes all activities in the transient object cache of the node to invoke the corresponding entry point in the user library. For details about the individual activities that are produced by the JADE sampling libraries in record type **2**, see "Cache Buffer Activity" under "Statistics File Format", later in this chapter.

SamplingFilterFile

Value Type String

Default Not specified

The **SamplingFilterFile** parameter specifies the name of the optional object filter command file in a user-supplied library, which is read each time the **beginSample** method is invoked. For details, see "Sampling Filtering" and "Filter Commands" under "JADE Sampling Libraries", later in this chapter.

SamplingLibraryInitialization

Value Type String

Default sampling.smp

The **SamplingLibraryInitialization** parameter specifies the string that the JADE Object Manager passes to the user-supplied library when the JADE system initializes and automatic JADE node sampling is enabled (that is, when the **SamplingNode** parameter is set to **true**). You can use this parameter to specify any handshake or initialize information that your library requires.

If you are using the **filesmpl** sampling library provided by JADE, you can use this parameter to specify the name of the sampling file (and optionally the file path) to which sampling is output if you want output directed to a file with a name other than the default **sampling.smp** in a destination other than the JADE installation directory (that is, in the directory in which the **jade.exe** program is located). For details, see "JADE Sampling Libraries" under "Sampling Library Interface", later in this chapter.

If you are using the **filesmpl** or **tcpsmpl** JADE sampling library, you can set this parameter to **"<null>"** or **""** so that sample values will not be output. For **filesmpl**, the values will not be written to a file. For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted. For more details, see "Direct Retrieval of Node Sampling Statistics", later in this chapter.

SamplingLibraryName

Value Type String

Default filesmpl

The **SamplingLibraryName** parameter specifies the name of the user-supplied library that the JADE Object Manager calls according to the sampling directives. For details, see "JADE Sampling Libraries" under "Sampling Library Interface", later in this chapter.

SamplingNode

Value Type Boolean

Default False

The **SamplingNode** parameter, when set to **true**, automatically initiates the sampling of statistics on the current node when the JADE system is initiated.

For details about the request statistics that are produced by the JADE sampling libraries in record types **8** and **9**, see "Local Request Statistics Format" and "Remote Requests Statistics Format" under "Statistics File Format", later in this chapter.

Node Class External Methods

The methods provided by instances of the **Node** class that enable you to sample statistics are described (in the order in which they would be called) in the following subsections.

beginSample

Signature
```
beginSample(libraryName:              String;
            initializationParameter: String): Integer;
```

The **beginSample** method of the **Node** class opens a new sampling context for the node, begins the accumulation of sampling statistics for that node, and invokes the following entry points.

- **NodeSampleInfoCallBack**, passing it the **initializationParameter** string and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleProcessInfoCallBack**, invoked every time that a process begins and once for every existing process at the time sampling begins.

This method returns the sampling handle number used to identify the sampling context that is opened. All subsequent methods use this sampling context handle as the first parameter.

When the **beginSample** method is called in your application, request statistics are stored in transient memory for every process in the node until they are passed to a corresponding entry point in the user library specified in the **libraryName** parameter.

The JADE-supplied library writes a record type (**6**) to the statistics file. For details about the node sampling entry points and the contents of the statistics sample files, see "JADE Sampling Libraries" and "Statistics File Format", later in this chapter.

If you are using the **filesmpl** or **tcpsmpl** JADE sampling library, you can set the **initializationParameter** parameter to **"<null>"** or to **""** so that sample values will not be output. For **filesmpl**, the values will not be written to a file.

For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted. For more details, see "Direct Retrieval of Node Sampling Statistics", later in this chapter.

logObjectCaches

Signature

```
logObjectCaches(samplingHandle:              Integer;
                persistentCacheStats:        Boolean;
                persistentCacheBuffers:      Boolean;
                transientCacheStats:         Boolean;
                transientCacheBuffers:       Boolean;
                remoteTransientCacheStats:   Boolean;
                remoteTransientCacheBuffers: Boolean;
                userNumber:                  Integer;
                userText:                    String);
```

The **logObjectCaches** method of the **Node** class specifies the local node object cache statistics to be logged by invoking the **NodeSampleCacheInfoCallBack** or **NodeSampleObjectBuffer** entry point or both of these entry points in the user library.

Note This method does not reset cumulative values.

The JADE-supplied library logs the statistics to the file specified in the **initializationParameter** parameter of the **beginSample** method and writes the following statistics to your output file.

- Cache header record (type **1**) for cache statistics
- Cache buffer records (type **2**) for individual object buffers

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

Parameter	Description
samplingHandle	Identifies the sampling context
persistentCacheStats	Logs statistics of the persistent objects cache
persistentCacheBuffers	Logs statistics of the persistent object cache buffers
transientCacheStats	Logs statistics of the transient objects cache
transientCacheBuffers	Logs statistics of the transient object cache buffers
remoteTransientCacheStats	Logs statistics of the remote transient objects cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache buffers
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the cache statistics that you require, set the appropriate Boolean cache parameters to **true**.

The following code fragment shows an example of the **logObjectCaches** method and its parameters.

```
node.logObjectCaches(samplingHandle, true, true, false, false, false, false,
                     50, "After the load data operation");
```

For more details, see "Statistics File Format", later in this chapter.

logRequestStatistics

Signature
```
logRequestStatistics(samplingHandle: Integer;
                     local:          Boolean;
                     remote:         Boolean;
                     userNumber:     Integer;
                     userText:       String);
```

The **logRequestStatistics** method of the **Node** class specifies the request statistics that are to be logged for all processes in the node by invoking the **NodeSampleRequestStatisticsCallBack** entry point in the user library.

The JADE-supplied library automatically writes the following statistics.

- Local request statistics record (type **8**)
- Remote request statistics record (type **9**)

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

Parameter	Description
samplingHandle	Identifies the sampling context
local	Logs statistics of all requests invoked on the local node
remote	Logs statistics of all requests from the local node to remote nodes
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the request statistics that you require, set the appropriate Boolean cache parameters to **true**. The user number and text values specified in the **userNumber** and **userText** parameters are written in the corresponding records.

Note This method does not reset cumulative values.

The following code fragment shows an example of the **logRequestStatistics** method and its parameters.

```
node.logRequestStatistics(samplingHandle, true, true, 23, "Before method
                          m1");
```

For more details, see "Statistics File Format", later in this chapter.

logUserCommand

Signature
```
Signature logUserCommand(samplingHandle: Integer;
                         command:        String;
                         userNumber:     Integer;
                         userText:       String);
```

The **logUserCommand** method of the **Node** class invokes the **NodeSampleUserCommandCallBack** entry point in the user library, passing the **command** parameter to it.

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

Parameter	Description
samplingHandle	Identifies the sampling context
command	Action specific to your user library (for example, the JADE-supplied library uses this command for filtering and setting the **SamplingExceptionEvent**)
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The JADE-supplied library automatically writes the user command (type **13**).

For more details, see "JADE Sampling Libraries" and "Statistics File Format", later in this chapter. See also "Sampling Exception Handling", later in this chapter, for details about handling node sampling exceptions.

beginIndividualRequestsLogging

Signature

```
beginIndividualRequestsLogging(samplingHandle:              Integer;
                               localRequests:               Boolean;
                               remoteRequests:              Boolean;
                               persistentCacheBuffers:      Boolean;
                               transientCacheBuffers:       Boolean;
                               remoteTransientCacheBuffers: Boolean;
                               userNumber:                  Integer;
                               userText:                    String);
```

The **beginIndividualRequestsLogging** method of the **Node** class starts sampling the individual requests or cache activities, or both, of all processes in the local node and invokes the **NodeSampleIndividualRequestCallBack** or **NodeSampleObjectBufferCallBack** entry point, or both of these entry points, in the user library specified in the **libraryName** parameter of the **beginSample** method.

The **NodeSampleIntervalCallBack** entry point is invoked once only before these entry points, with the **eventType** parameter in the entry point set to **1**.

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

Parameter	Description
samplingHandle	Identifies the sampling context
localRequests	Logs individual requests to the database of the node
remoteRequests	Logs individual requests to remote nodes
persistentCacheBuffers	Logs activities in the persistent object cache
transientCacheBuffers	Logs activities in the transient object cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the sampling of the statistics that you require, set the appropriate Boolean parameters to **true**.

The following code fragment shows an example of the **beginIndividualRequestsLogging** method and its parameters.

```
node.beginIndividualRequestsLogging(samplingHandle, false, true, false,
                        false, false, 4, "Start logging of remote requests");
```

The JADE sampling libraries produce the following record types.

- Begin process record (type **6**) (optionally)
- **BeginInterval** record (type **11**), containing your specified user number and text to the output file immediately, followed by one **IndividualRequest** record for each of the subsequent individual requests or one cache buffer activity record for each of the subsequent buffer cache activities, or both
- Individual local request records (record type **14**)
- Individual remote request records (record type **10**)
- Cache buffer activity records (record type **2**)

For details about the individual remote requests that are sampled in record types **2**, **6**, **7**, **10**, **11**, and **14**, see "Statistics File Format", later in this chapter.

endIndividualRequestsLogging

Signature

```
endIndividualRequestsLogging(samplingHandle: Integer;
                             userNumber:     Integer;
                             userText:       String);
```

The **endIndividualRequestsLogging** method of the **Node** class terminates the sampling of individual requests or cache activities started by the **beginIndividualRequestsLogging** method of the **Node** class.

The **NodeSamplIntervalCallBack** entry point is invoked with the **eventType** parameter set to **2**.

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

Parameter	Description
samplingHandle	Identifies the sampling context
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The following code fragment shows an example of the **endIndividualRequestsLogging** method and its parameters.

```
node.endIndividualRequestsLogging(samplingHandle, 4, "End logging of remote
                                  requests");
```

The JADE-supplied library writes an **endInterval** record (type **12**), containing your specified user number and text, which is written to the output file specified in the **initializationParameter** parameter of the **beginSample** method.

For details about the end interval and end process records, see "End Interval" and "End Process" under "Statistics File Format", later in this chapter.

endSample

Signature `endSample(samplingHandle: Integer;);`

The **endSample** method of the **Node** class terminates the sampling of statistics on the local node for the context identified by the **samplingHandle** parameter and invokes the following entry points:

- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **2**.
- **NodeSampleInfoCallBack**, which your user library should consider the last call for the node sampling context.

The JADE-supplied library automatically writes the record trailer (type **20**) and then closes and releases the current sampling file, which you can then analyze.

You can produce multiple files during a node lifetime, by using the **Node** class **beginSample** and **endSample** methods.

System Class External Methods

Instances of the **System** class provide the following methods (described in the order in which they would be called) to sample statistics for all remote nodes in a JADE system that have been enabled by using the **enableRemoteSampling** method.

beginSampleGroupDefinition

Signature `beginSampleGroupDefinition(): Integer;`

The **beginSampleGroupDefinition** method of the **System** class opens a new remote sampling context for a group of nodes.

The nodes are included in the sampling context by using the **System** class **enableRemoteSampling** method.

The **beginSampleGroupDefinition** method returns the sampling handle number used to identify the sampling context that is opened. All subsequent methods use this sampling context handle as the first parameter and are initially executed by the server node and sent to each of the nodes in the definition group by internal notifications.

Any error condition at the individual node level is written to the JADE Object Manager message log file for that node.

enableRemoteSampling

Signature
```
enableRemoteSampling(samplingHandle: Integer;
                     n:              Node);
```

The **enableRemoteSampling** method of the **System** class enables the sampling of statistics on the node specified in the **n** parameter.

This method includes the node in the sampling group of the context identified in the **samplingHandle** parameter.

isRemoteSamplingEnabled

Signature
```
isRemoteSamplingEnabled(samplingHandle: Integer;
                        n:              Node): Boolean;
```

The **isRemoteSamplingEnabled** method of the **System** class returns **true** if the node specified in the **n** parameter is included in the sampling group identified in the **samplingHandle** parameter.

disableRemoteSampling

Signature
```
disableRemoteSampling(samplingHandle: Integer;
                      n:              Node);
```

The **disableRemoteSampling** method of the **System** class disables the sampling of statistics on the node specified in the **n** parameter. This method takes the specified node out of the sampling group of the context identified by the **samplingHandle** parameter.

beginSample

Signature
```
beginSample(samplingHandle:          Integer;
            libraryName:             String;
            initializationParameter: String);
```

The **beginSample** method of the **System** class opens a new sampling context for each of the nodes in the sample definition group, begins the accumulation of sampling statistics for those nodes, and invokes the following entry points:

- **NodeSampleInfoCallBack**, passing it the **initializationParameter** string and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleNodeInfoCallBack**, passing it information about the local node to the point and setting the **eventType** parameter in the user library entry point to **1**.
- **NodeSampleProcessInfoCallBack**, invoked every time a process begins and once for every existing process at the time sampling begins.

The **samplingHandle** parameter identifies the context that is sampled.

When this method is called in your application, request statistics are stored in transient memory for every process in the nodes of the sample definition group until they are passed to corresponding entry point in the user library specified in the **libraryName** parameter. The JADE-supplied library writes a record type (**6**) to the statistics file.

For details about the node sampling entry points and the contents of the statistics sample files, see "JADE Sampling Libraries" and "Statistics File Format", later in this chapter.

If you are using the **filesmpl** or **tcpsmpl** JADE sampling library, you can set the **initializationParameter** parameter to **"<null>"** or to **""** so that sample values will not be output. For **filesmpl**, the values will not be written to a file.

For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted. For more details, see "Direct Retrieval of Node Sampling Statistics", later in this chapter.

logObjectCaches

Signature

```
logObjectCaches(samplingHandle:              Integer;
                persistentCacheStats:        Boolean;
                persistentCacheBuffers:      Boolean;
                transientCacheStats:         Boolean;
                transientCacheBuffers:       Boolean;
                remoteTransientCacheStats:   Boolean;
                remoteTransientCacheBuffers: Boolean;
                userNumber:                  Integer;
                userText:                    String);
```

The **logObjectCaches** method of the **System** class specifies the node object cache statistics that are logged by invoking the **NodeSampleCacheInfoCallBack** or the **NodeSampleObjectBuffer** entry point, or both of these entry points, in the user library for each of the nodes in the sample definition group.

Note This method does not reset cumulative values.

The JADE-supplied library logs the statistics to the file specified in the **initializationParameter** parameter of the **beginSample** method and writes the following statistics to your output file on each node in the group.

- Cache header record (type **1**) for cache statistics
- Cache buffer records (type **2**) for individual object buffers

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

Parameter	Description
samplingHandle	Identifies the sampling context
persistentCacheStats	Logs statistics of the persistent objects cache
persistentCacheBuffers	Logs statistics of the persistent object cache buffers
transientCacheStats	Logs statistics of the transient objects cache
transientCacheBuffers	Logs statistics of the transient object cache buffers
remoteTransientCacheStats	Logs statistics of the remote transient objects cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache buffers
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the cache statistics that you require, set the appropriate Boolean cache parameters to **true**.

The following code fragment shows an example of the **logObjectCaches** method and its parameters.

```
system.logObjectCaches(samplingHandle, true, true, false, false, false,
                       false, 50, "After the load data operation");
```

For more details, see "Statistics File Format", later in this chapter.

logRequestStatistics

Signature
```
logRequestStatistics(samplingHandle: Integer;
                     local:          Boolean;
                     remote:         Boolean;
                     userNumber:     Integer;
                     userText:       String);
```

The **logRequestStatistics** method of the **System** class specifies the request statistics that are logged for all processes in each of the nodes in the sample definition group, by invoking the **NodeSampleRequestStatisticsCallBack** entry point in the user library.

Note This method does not reset cumulative values.

The JADE-supplied library automatically writes the following statistics.

- Local request statistics record (type **8**)
- Remote request statistics record (type **9**)

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

Parameter	Description
samplingHandle	Identifies the sampling context
local	Logs statistics of all requests invoked on the local node
remote	Logs statistics of all requests from the local node to remote nodes
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the logging of the request statistics that you require, set the appropriate Boolean cache parameters to **true**.

The user number and text values specified in the **userNumber** and **userText** parameters are written in the corresponding records.

The following code fragment shows an example of the **logRequestStatistics** method and its parameters.

```
system.logRequestStatistics(samplingHandle, true, true, 23, "Before method
                            m1");
```

For more details, see "Statistics File Format", later in this chapter.

logUserCommand

Signature
```
logUserCommand(samplingHandle: Integer;
               command:        String;
               userNumber:     Integer;
               userText:       String);
```

The **logUserCommand** method of the **System** class causes the invocation of the **NodeSampleUserCommandCallBack** entry point in the user library for each of the nodes in the sample definition group, passing the **command** parameter to it.

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

Parameter	Description
samplingHandle	Identifies the sampling context
command	Action specific to your user library (for example, the JADE-supplied library uses this command for filtering)
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The JADE-supplied library automatically writes the user command record (type **13**).

For more details, see "JADE Sampling Libraries" and "Statistics File Format", later in this chapter.

beginIndividualRequestsLogging

Signature

```
beginIndividualRequestsLogging(samplingHandle:              Integer;
                               localRequests:               Boolean;
                               remoteRequests:              Boolean;
                               persistentCacheBuffers:      Boolean;
                               transientCacheBuffers:       Boolean;
                               remoteTransientCacheBuffers: Boolean;
                               userNumber:                  Integer;
                               userText:                    String);
```

The **beginIndividualRequestsLogging** method of the **System** class starts sampling the individual remote requests or cache activities, or both, of all processes in each of the remote nodes in the sample definition group and invokes the **NodeSampleIndividualRequestCallBack** or the **NodeSampleObjectBufferCallBack** entry point, or both of these entry points, in the user library specified in the **libraryName** parameter of the **beginSample** method.

The **NodeSampleIntervalCallBack** entry point is invoked once only before these entry points, with the **eventType** parameter in the entry point set to **1**.

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

Parameter	Description
samplingHandle	Identifies the sampling context
localRequests	Logs individual requests to the database of the node
remoteRequests	Logs individual requests to remote nodes
persistentCacheBuffers	Logs activities in the persistent object cache
transientCacheBuffers	Logs activities in the transient object cache
remoteTransientCacheBuffers	Logs activities in the remote transient object cache
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

To enable the sampling of the statistics that you require, set the appropriate Boolean parameters to **true**.

The following code fragment shows an example of the **beginIndividualRequestsLogging** method and its parameters.

```
system.beginIndividualRequestsLogging(samplingHandle, false, true, false,
                      false, false, 4, "Start logging of remote requests");
```

The JADE sampling libraries produce the following record types.

- Begin process record (type **6**) (optionally)
- **BeginInterval** record (type **11**), containing your specified user number and text to the output file immediately, followed by one **IndividualRequest** record for each of the subsequent remote requests or one cache buffer activity record for each of the subsequent buffer cache activities, or both
- Individual local request records (record type **14**)
- Individual remote request records (record type **10**)
- Cache buffer activity records (record type **2**)

The JADE-supplied library writes a **beginInterval** record (type **11**) containing your specified user number and text to the output file immediately, followed by one **IndividualRequest** record for each of the subsequent remote requests.

For details about the individual remote requests that are sampled in record types **2**, **6**, **7**, **10**, **11**, and **14**, "Statistics File Format", later in this chapter.

endIndividualRequestsLogging

Signature

```
endIndividualRequestsLogging(samplingHandle: Integer;
                             userNumber:     Integer;
                             userText:       String);
```

The **endIndividualRequestsLogging** method of the **System** class terminates the sampling of individual requests or cache activities started by the **beginIndividualRequestsLogging** method of the **System** class for each of the nodes in the sample definition group and invokes the **NodeSamplIntervalCallBack** entry point with the **eventType** parameter of the entry point set to **2**.

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

Parameter	Description
samplingHandle	Identifies the sampling context
userNumber	Identifies the sample in the corresponding user library invocations
userText	In conjunction with the **userNumber** parameter, identifies the sample

The following code fragment shows an example of the **endIndividualRequestsLogging** method and its parameters.

```
system.endIndividualRequestsLogging(samplingHandle, 4, "End logging of
                                    remote requests");
```

The JADE-supplied library writes an **endInterval** record (type **12**) containing your specified user number and text to the output file.

For details, see "End Interval" and "End Process" under "Statistics File Format", later in this chapter.

endSample

Signature `endSample(samplingHandle: Integer;);`

The **endSample** method of the **System** class terminates the sampling of statistics on each of the nodes in the sample definition group for the context identified by the **samplingHandle** parameter and invokes the following entry points.

- **NodeSampleNodeInfoCallBack**, passing it information about the local node and setting the **eventType** parameter in the user library entry point to **2**.
- **NodeSampleInfoCallBack**, which your user library should consider the last call for the node sampling context.

The JADE-supplied library automatically writes the record trailer (type **20**) and then closes and releases the current sampling file, which you can then analyze.

You can produce multiple files during a node lifetime, by using the **System** class **beginSample** and **endSample** methods.

endSampleGroupDefinition

Signature `endSampleGroupDefinition(samplingHandle: Integer;);`

The **endSampleGroupDefinition** method of the **System** class terminates the sampling context identified in the **samplingHandle** parameter for the group of nodes.

For details about opening a new remote sampling context for a group of nodes, see "beginSampleGroupDefinition", earlier in this chapter.

Sampling Library Interface

The JADE Object Manager provides a mechanism for sampling data capture by calling user-supplied entry points at different times during normal operation. You can specify the operations that cause calls to the entry points.

The overhead involved in data capturing is proportional to the level of detail required, the amount of processing in the node being sampled, and the amount of processing that your user library performs on each invocation.

For details about direct node sampling and initializing a sampling library (including when direct node sampling using the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted), see "Direct Retrieval of Node Sampling Statistics", later in this chapter.

The names of the entry points for your user-supplied library are listed in the following table.

Entry Point	Invoked…
nodeSampleInfoCb	Once, at the time a sampling context begins.
nodeSampleNodeInfoCb	Once, at the time a sampling context begins or ends and after (for begins) or before (for ends) the **NodeSampleInfoCallBack** entry point has been invoked.
nodeSampleProcessInfoCb	Every time a process begins or ends and once for every existing process at the time sampling begins.
nodeSampleIntervalCb	Every time an individual request interval begins or ends.

Entry Point	Invoked…
nodeSampleIndividualRequestCb	Every time an individual request occurs, only within those intervals that requested specific levels of details.
nodeSampleRequestStatisticsCb	When it is explicitly requested by using the **logRequestStatistics** method or when a process ends.
nodeSampleCacheInfoCb	When it is explicitly requested by using the **logObjectCaches** method.
nodeSampleObjectBufferCb	When it is explicitly requested by using the **logObjectCaches** method. It is also invoked every time a cache activity occurs, only within those intervals that requested specific levels of details.

The definitions of these entry points are in the **jomSamplingTypes.h** file in the **include** directory on your JADE release medium. In addition, a full example of a user library is provided in the **examples\sampling** directory. This example library implements the same functionality of the **filesmpl** library.

JADE Sampling Libraries

JADE provides **filesmpl** and **tcpsmpl** libraries that implement the JADE Object Manager sampling callback facilities.

The **filesmpl** library sends all sampling output to a specified file, and the **tcpsmpl** library sends all sampling output to a specified TCP/IP connection.

If the **SamplingNode** parameter in the initialization file is set to **true** and the default **filesmpl** library is used, automatic sampling is output to the JADE installation directory (that is, the location of the **jade.exe** executable program) **sampling.smp** file.

Each line of sampling that is output to a file is terminated with carriage return/line feed characters (**0x0d/0x0a**).

If you are doing automatic sampling with output sent to a file, you can specify the following parameters in the [JadeClient] section of the JADE initialization file for the JADE sampling callback library.

Parameter	Value	Description
SamplingLibraryName	filesmpl	Name of the library, without the file name extension.
SamplingLibraryInitialization	sampling.smp	Name of the file to which output is sent. The file name has the following macro definitions, or escape characters, which enable you to generate unique file names for each node to avoid duplication. %p Process id %d Date in *ddMMyyyy* format %t Time in *hhmmss* format %c Computer name %% Percent character For example, **c:\perftest%c\sampling%p%%.smp** generates **c:\perftestwilbur\sampling123%.smp**.

Parameter	Value	Description
(continued)		When multiple sampling files are to be created and you use the macros (that is, **%p**, **%d**, **%t**, and **%c**), they are substituted with the process id, date, time, and computer name, respectively. For example, if you sample from multiple nodes and you do not specify **%p** macro in the file name to distinguish the sampling file of each node, an error message is generated when you output sampling to a file, advising you *No %p specified*.
SamplingFilterFile	Not specified	Optional object filter command file name, in a user-supplied library, which is read each time the **beginSample** method is invoked. For more details, see "Sampling Filtering", later in this chapter.

If you are doing automatic sampling with output sent to a TCP/IP connection, you must specify the following parameters in the [JadeClient] section of the JADE initialization file for the JADE sampling callback library.

Parameter	Value	Description
SamplingLibraryName	tcpsmpl	Name of the library, without the filename extension.
SamplingLibraryInitialization	*host,port[,localport]*	TCP/IP connection parameters, which are a comma-separated list of parameters. No spaces are allowed before or after the comma separator. The first parameter is the host name or IP address of the host to connect to, the second parameter is the port on the remote host to connect to, and the optional third parameter is the port number or port name on the client (local) node through which you want to connect.
SamplingFilterFile	Optional	Optional object filter command file name, in a user-supplied library, which is read each time the **beginSample** method is invoked. For more details, see "Sampling Filtering", later in this chapter.

The sampling output of the **tcpsmpl** library is the same as the output of the **filesmpl** library, except that every line of sampling is preceded with a 4-character field indicating the length of the following data.

Caution If you want to retain an existing sampling file (for example, you have a sampling context that was output to the default file), ensure that you define a unique name for the next sampling context, as existing files are cleared if sampling is started to a file whose path and name already exist.

The **tcpsmpl** and **filesmpl** libraries also allow **"<null>"** or **""** as a **SamplingLibraryInitialization** parameter value. Specifying **"<null>"** or **""** causes output to be suppressed.

Sampling Filtering

JADE provides a mechanism to filter object output records so that you can specify the objects in which you are interested. Write a sampling filter command file by using a text editor (for example, **Notepad**).

The **filesmpl** library can take object filter commands from a sampling filter command file.

Specify the command file in the **SamplingFilterFile** parameter in the [JadeClient] section or [JadeServer] section of the JADE initialization file. (For details, see "Filter Commands", in the following subsection.)

The **tcpsmpl** library can take object filter commands from a sampling filter command line at startup, as well as from the TCP/IP connection at run time.

When performing manual sampling by using the **Node** class or **System** class methods, the contents of the **SamplingLibraryName** and **SamplingLibraryInitialization** parameters in the [JadeClient] section or [JadeServer] section of the JADE initialization file must be passed as the **libraryName** and **initializationParameter** parameters in the appropriate **beginSample** method.

In addition, you can supply the contents of the **SamplingFilterFile** parameter one command at a time, by using the **command** parameter in the **logUserCommand** method in the **Node** class or **System** class. (You can subscribe to an exception in a sampling library by specifying the **SamplingExceptionEvent** number in the **command** parameter of the **Node** class **logUserCommand** method.)

Filter Commands

You can specify the following filter commands.

- **clear**

  Clears all filter entries.

- **add** record *recordNumber* class [not] *classSpec*[*classSpec* …]

  Adds a class filter for the specified record type and the specified class number.

- **delete** record *recordNumber* class [not] *classSpec*[*classSpec* …]

  Deletes a class filter for the specified record type and the specified class number.

- **;**

  Any line that starts with a semicolon character is ignored.

The following is an example of sampling filter commands in a filter file.

```
add record 2 class not 135 150-175 200
add record 10 class 25 50 75 100-105
;delete record 14 class not 10-50
delete record * class 350
```

Record Specification

The *recordNumber* variable can be any integer value, but filtering currently works only on record types **2**, **10**, and **14**, as these are the only callbacks that have **DskObjectId** output values. An asterisk character (*****) specified for the *recordNumber* value is recognized as "all record types".

Record types **10** and **14** that indicate a **BEGIN_TRANSACTION**, **END_TRANSACTION**, or **ABORT_TRANSACTION** cannot be filtered.

Class Specification

The *classSpec* variable can be an integer value representing the class number (obtained from the Class browser in the JADE development environment, for example), two or more integer values separated by spaces, or it can be two integer values separated by a hyphen character (-) to indicate a class range.

To exclude the class number or range, precede the class number or class range specification with a **not** operator.

If your class specification is inclusive (that is, the command does not contain the **not** operator), all other class numbers are excluded unless you specify them in the class specification of another filter command.

One or more *classSpec* values can follow a *recordNumber* value, using the same record number (for example, **add record 14 class 135 150-175 200**).

Sampling Exception Handling

Sampling libraries cannot raise exceptions, but you can specify a user notification event number if you want to subscribe to an exception in a sampling library.

If an error occurs in the sampling library, a user notification event is caused on the **Node** object in which the exception occurs. If you want to be notified of a sampling exception, subscribe to an exception in a sampling library by specifying the **SamplingExceptionEvent** number in the **command** parameter of the **Node** class **logUserCommand** method, in the following format.

```
SamplingExceptionEvent user-notification-event-number
```

In this format, the *user-notification-event-number* must be in the range **User_Base_Event** through **User_Max_Event** of the global constants in the **UserEvents** category, listed in the following table.

Global Constant	Integer Value
User_Base_Event	16
User_Max_Event	Max_Integer (#7FFFFFFF, equates to 2147483647)

If you subscribe to a sampling exception event by using the **SamplingExceptionEvent** and an exception occurs in the sampling library, an event is caused on the **Node** object to notify the subscriber that a sampling exception occurred. Sampling for that node is then terminated.

The userInfo parameter in the **Object** class **userNotification** or a **userNotify** event method of the Windows class or subclass is then set by the **Class** class **causeClassEvent** or **Object** class **causeEvent** method to one of the following values.

- SAMPLING_OPEN_FAILED (1), indicating that the opening of the connection or file failed
- SAMPLING_WRITE_FAILED (2), for example, the disk is full or the connection lost

Sampling for the current session is terminated, regardless of whether an exception event was caused (that is, if you subscribed to a sampling exception by specifying a user notification event number in the **command** parameter of the **Node** class **logUserCommand** method or not).

Statistics File Format

This section describes the file format produced be the JADE-supplied libraries. The statistics output file contains the records described in the following subsections. Fields are separated by spaces, with the exception of user text, which is enclosed in quote characters.

The statistics file logs a line for each type of statistic that is captured, indicated by the record type values; for example, the local request statistics record is record type **8**.

Tip If you read 64-bit numbers into a JADE local variable from a statistics file or TCP/IP connection, read them into an **Integer64** primitive type value, as an **Integer** primitive type value may be too small.

For details, see:

File Header

The **file header** record (type **0**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record.
Node number	Node number relative to the system.
Node type	Integer value representing the type of node on which records are sampled. The value can be **1** (database server), **2** (application server), **3** (application server and database server), **4** (standard, or fat, client), **5** (standard, or fat, client and database server).
Computer name	Name of the computer where the node is executing.
JADE version number	Number of the JADE version from which statistics are sampled.
Date	Date on which the sample is executed converted to the internal (Julian day) format for internal storage and computation.
Time	Time in milliseconds from midnight UTC (that is, 00:00) on that date until the sample is executed.
UTC bias	Coordinated Universal Time (UTC) bias of the machine on which the node sample is executed.
Timestamp string	Text representation of the date and time as a timestamp (in the local time of the workstation from which the sample was executed).

For more details about Coordinated Universal Time (UTC), see "File Trailer", later in this chapter.

Begin Process

The **begin process** record (type **6**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record.
Node number	Node number relative to the system.
Process number	Process number relative to the system.
Clock ticks	64-bit number of microseconds elapsed since the node was initialized.
Node CPU time	64-bit number of microseconds of CPU time charged for the node.
Node ticks	64-bit number of logical operations performed by the node.
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process.
Process ticks	64-bit number of logical operations performed by the process in the node.
Process logical clock	Sequential 64-bit number for inter-node process context switches.
Starting process id	Starting process number identification.
Start clock ticks	64-bit number of microseconds elapsed since the process was initialized.
Start node CPU time	64-bit number of microseconds of CPU time charged for the node when the process started.

Statistic	Description
Start node ticks	64-bit number of logical operations performed by the node when the process started.
Start process CPU time	64-bit number of microseconds of CPU time charged for the process when the process started.
Start process ticks	64-bit number of logical operations performed by the process when the process started.
Start process logical clock	64-bit number of inter-node process context switches when the process started.
Instance id	Starting process instance identification, unique across nodes. (Available only for processes already executing when sampling begins. The corresponding **end process** record is always available.)
Process location	Integer value representing the location in which the process starts. For details, see "Process Location", later in this chapter.
Schema name	Name of the schema in which statistics are sampled.
Application name	Name of the application in which statistics are sampled.

End Process

The **end process** record (type **7**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record.
Node number	Node number relative to the system.
Process number	Process number relative to the system.
Clock ticks	64-bit number of microseconds elapsed since the node was initialized.
Node CPU time	64-bit number of microseconds of CPU time charged for node.
Node ticks	64-bit number of logical operations performed by the node.
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process.
Process ticks	64-bit number of logical operations performed by the process.
Process logical clock	Sequential 64-bit number for inter-node process context switches.
Ending process Id	Ending process number identification.
End clock ticks	64-bit number of microseconds elapsed when the process ended.
End node CPU time	64-bit number of microseconds of CPU time charged for the node when the process ended.
End node ticks	64-bit number of logical operations performed by the node when the process ended.
End process CPU time	64-bit number of microseconds of CPU time charged for the process when the process ended.
End process ticks	64-bit number of logical operations performed by the process when the process ended.

Statistic	Description
End process logical clock	64-bit number of inter-node process context switches when the process ended.
Instance id	Ending process instance identification, unique across nodes.
Process location	Integer value representing the location in which the process ends. For details, see "Process Location", in the following subsection.
Schema name	Name of the schema in which statistics are sampled.
Application name	Name of the application in which statistics are sampled.

Process Location

The process location values that can be output in the begin and end process records (that is, record types **6** and **7**) are listed in the following table.

Process Location	Integer Value
LOCAL_SAMPLING	1
REMOTE_SAMPLING	2

Begin Interval

The **begin interval** record (type **11**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request
Individual local requests	Not yet implemented (0)
Individual remote requests	To be written in the interval (0 or 1)
Individual persistent cache activities	To be written in the interval (0 or 1)
Individual transient cache activities	To be written in the interval (0 or 1)
Individual remote transient cache activities	To be written in the interval (0 or 1)

End Interval

The **end interval** record (type **12**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request

Local Request Statistics Format

The **local request statistics format** record (type **8**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request
Statistics process number	Statistics process number relative to the system

This information is then followed by an array of statistics, as listed in the following table.

Statistic	Number of…
Transient object creates	Transient objects created
Transient object deletes	Transient objects deleted
Transient object clones	Transient objects cloned
Transient object copies	Transient objects copied
Transient object get properties	Transient object properties obtained
Transient object set properties	Transient object properties set
Transient object JADE methods	JADE methods for transient objects
Transient object external methods	External methods for transient objects
Transient object locks	Transient objects locked
Transient object unlocks	Transient objects unlocked
Transient object remove locks	Transient object locks removed
Transient object gets	JADE method for transient objects
Transient object puts	Transient objects written to the transient database
Transient buffer swaps	Transient buffers swapped
Persistent object creates	Persistent objects created
Persistent object deletes	Persistent objects deleted
Persistent object clones	Persistent objects cloned
Persistent object copies	Persistent objects copied
Persistent object get properties	Persistent object properties obtained
Persistent object set properties	Persistent object properties set
Persistent object JADE methods	JADE methods for persistent objects
Persistent object external methods	External methods for persistent objects
Persistent object locks	Persistent objects locked
Persistent object unlocks	Persistent objects unlocked
Persistent object remove locks	Persistent object locks removed
Persistent object gets	Persistent objects obtained
Persistent object puts	Persistent objects written to the transient database
Persistent buffer swaps	Persistent buffers swapped
Primitive JADE methods	Primitive type JADE methods
Primitive external methods	Primitive type external methods
Transient begin transactions	Transient **beginTransientTransaction** instructions
Transient end transactions	Transient **commitTransientTransaction** instructions
Transient abort transactions	Transient **abortTransientTransaction** instructions

Statistic	Number of...
Transient begin notifications	Transient **beginNotification** method calls
Transient end notifications	Transient **endNotification** method calls
Transient cause events	Transient **causeEvent** method calls
Transient return notes	Transient notifications returned
Received notes from local	Notifications received from the local node
User requests	User level-zero (**0**) kernel requests
Persistent begin transactions	Persistent **beginTransaction** instructions
Persistent end transactions	Persistent **commitTransaction** instructions
Persistent abort transactions	Persistent **abortTransaction** instructions
Persistent begin notifications	Persistent **beginNotification** method calls
Persistent end notifications	Persistent **endNotification** method calls
Persistent cause events	Persistent **causeEvent** method calls
Persistent return notes	Persistent notifications returned
Thin client messages sent by the application server	Messages sent by the application server to the presentation client
Thin client bytes sent by the application server	Bytes sent by the application server to the presentation client
Thin client messages sent by the client	Messages sent by the presentation client to the application server
Thin client bytes sent by the client	Bytes sent by the presentation client to the application server
Thin client logic message wait time	Milliseconds that logic had to wait for the reply to a message sent by the application server to the presentation client (this does not include waiting for modal forms, message boxes, common dialogs, or exception dialogs)

Note that the **Thin client messages sent by the application server** value will not necessarily be the same as the **Thin client messages sent by the client** value, because of keep-alive messages that are only sent one way.

Remote Requests Statistics Format

The **remote request statistics format** record (type **9**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node

Statistic	Description
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request
Statistics process number	Statistics process number relative to the system

This information is then followed by an array of statistics, as listed in the following table.

Statistic	Description
RPC new buffer get objects	Number of get object operations for new buffers
RPC updated buffer get objects	Number of get object operations for updated buffers
RPC non-updated buffer get objects	Number of get object operations for non-updated buffers
RPC temporary buffer get objects	Number of get object operations for temporary buffers
RPC new buffer get object groups	Number of get object in-group operations for new buffers
RPC updated buffer get object groups	Number of get object in-group operations for updated buffers
RPC non-updated buffer get object groups	Number of get object in-group operations for non-updated buffers
RPC temporary buffer get object groups	Number of get object in-group operations for temporary buffers
RPC get object groups	Number of get object group operations
RPC new buffer lock objects	Number of lock object operations for new buffers
RPC updated buffer lock objects	Number of lock object operations for updated buffers
RPC non-updated buffer lock objects	Number of lock object operations for non-updated buffers
RPC temporary buffer lock objects	Number of lock object operations for temporary buffers
RPC new buffer lock object groups	Number of lock object in-group operations for new buffers
RPC updated buffer lock object groups	Number of lock object in-group operations for updated buffers
RPC non-updated buffer lock object groups	Number of lock object in-group operations for non-updated buffers
RPC temporary buffer lock object groups	Number of lock object in-group operations for temporary buffers
RPC lock object groups	Number of lock object group operations
RPC create objects	Number of create object operations
RPC update objects	Number of update object operations

Statistic	Description
RPC delete objects	Number of delete object operations
RPC unlock objects	Number of unlock object operations
RPC unlock object groups	Number of unlock object group operations
RPC remove locks	Number of remove lock operations
RPC get editions	Number of get edition operations
RPC get oids	Number of get oid operations
RPC begin transactions	Number of Remote Procedure Call **beginTransaction** instructions
RPC end transactions	Number of Remote Procedure Call **commitTransaction** instructions
RPC abort transactions	Number of Remote Procedure Call **abortTransaction** instructions
RPC begin notifications	Number of Remote Procedure Call **beginNotification** method calls
RPC end notifications	Number of Remote Procedure Call **endNotification** method calls
RPC cause events	Number of Remote Procedure Call **causeEvent** method calls
RPC return notes	Number of Remote Procedure Call notifications returned
Received notes from remote	Number of notifications received from the remote nodes
RPC server execution	Number of Remote Procedure Call server execution methods
RPC send database message	Number of Remote Procedure Call send database messages
RPC new buffer get objects time	Elapsed time for get object operations for new buffers
RPC updated buffer get objects time	Elapsed time for get object operations for updated buffers
RPC non-updated buffer get objects time	Elapsed time for get object operations for non-updated buffers
RPC temporary buffer get objects time	Elapsed time for get object operations for temporary buffers
RPC new buffer get objects group time	Elapsed time for get object in-group operations for new buffers
RPC updated buffer get objects group time	Elapsed time for get object in-group operations for updated buffers
RPC non-updated buffer get objects group time	Elapsed time for get object in-group operations for non-updated buffers
RPC temporary buffer get objects group time	Elapsed time for get object in-group operations for temporary buffers
RPC get object group time	Elapsed time for get object group operations
RPC new buffer lock objects time	Elapsed time for lock object operations for new buffers

Statistic	Description
RPC updated buffer lock objects time	Elapsed time for lock object operations for updated buffers
RPC non-updated buffer lock objects time	Elapsed time for lock object operations for non-updated buffers
RPC temporary buffer lock objects time	Elapsed time for lock object operations for temporary buffers
RPC new buffer lock objects group time	Elapsed time for lock object in-group operations for new buffers
RPC updated buffer lock objects group time	Elapsed time for lock object in-group operations for updated buffers
RPC non-updated buffer lock objects group time	Elapsed time for lock object in-group operations for non-updated buffers
RPC temporary buffer lock objects group time	Elapsed time for lock object in-group operations for temporary buffers
RPC lock objects group time	Elapsed time for lock object group operations
RPC lock queue waits time	Total wait time for queued requests
RPC put objects time	Elapsed time for put object operations
RPC unlock object time	Elapsed time for unlock object operations
RPC unlock object groups time	Elapsed time for unlock object group operations
RPC remove locks time	Elapsed time for remove lock operations
RPC get editions time	Elapsed time for get edition operations
RPC get oids time	Elapsed time for get oid operations
RPC begin transactions time	Elapsed time for begin transaction operations
RPC end transactions time	Elapsed time for end transaction operations
RPC abort transactions time	Elapsed time for abort transaction operations
RPC begin notifications time	Elapsed time for begin notification operations
RPC end notifications time	Elapsed time for end notification operations
RPC cause events time	Elapsed time for cause event operations
RPC return notes time	Elapsed time for return notification operations
RPC server execution time	Elapsed time for server method executions
RPC send database message time	Elapsed time for database method executions
All request bytes sent	Total bytes sent for all remote requests
All request bytes received	Total bytes received for all remote requests
All requests packets sent	Total packets sent for all remote requests
All requests packets received	Total packets received for all remote requests
All receipts bytes received	Total bytes received for all remote-initiated requests
All receipts packets received	Total packets received for all remote-initiated requests

Individual Local Requests

The **individual local requests** record (type **14**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
Request count	Sequential number for the sampled requests
Request level	Number of nested requests
Request type	For details, see "Statistics File Request Values", in the following section
Mode	For details, see "Statistics File Request Values", in the following section (event type for notifications)
Duration	Number of milliseconds elapsed performing the request (**0** for notifications)
Size	Object record size (**0** when not available)
Flag	For details, see the Buffer Flag table, later in this chapter
Result	Result of the request; that is, zero (**0**) for a successful result or the JADE error code, documented under JADE Error Messages and System Messages in the JADE online help, from the **jomerrs.h** file that represents the specific error (for example, **1027** for a locked object)
Wait time	Zero (**0**) if the request was not queued or the time in milliseconds that the request was queued
Process instance id	Process instance id with which a contention occurred in the case of object locks or deadlocks
Class id	Class number for the object identifier
Instance id	Instance number for the object identifier
Owner class id	Owner object class number for the object identifier
Sub level	Class hierarchy level number for the object identifier
Sub id	Subobject number for the object identifier
Edition	Edition number for the object identifier
Invocation mode	For details, see "Statistics File Request Values" in the following section

Statistic	Description
Method class id	Object identifier of the executing method
Method instance id	Instance number for the object identifier of the executing method
Method owner class id	Owner object class number for the object identifier of the executing method
Method sub level	Class hierarchy level number for the object identifier of the executing method
Method sub id	Subobject number for the object identifier of the executing method
Method edition	Edition number for the object identifier of the executing method
Receiver class id	Object identifier of the receiver
Receiver instance id	Instance number for the object identifier of the receiver
Receiver owner class id	Owner object class number for the object identifier of the receiver
Receiver sub level	Class hierarchy level number for the object identifier of the receiver
Receiver sub id	Subobject number for the object identifier of the receiver
Receiver edition	Edition number for the object identifier of the receiver
Volatility	For details, see "Statistics File Request Values" in the following section
Lock Kind	For details, see "Statistics File Request Values" in the following section

Individual Remote Requests

The **individual remote requests** record (type **10**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
Request count	Sequential number for the sampled requests
Request level	Number of nested requests
Request type	For details, see "Statistics File Request Values", in the following section
Mode	For details, see "Statistics File Request Values", in the following section (event type for notifications)
Duration	Number of milliseconds elapsed performing the request (**0** for notifications)

Statistic	Description
Size	Object record size (**0** when not available)
Flag	For details, see the Buffer Flag table, later in this chapter
Result	Result of the request; that is, zero (**0**) for a successful result or the JADE error code, documented under JADE Error Messages and System Messages in the JADE online help, from the **jomerrs.h** file that represents the specific error (for example, **1027** for a locked object)
Wait time	Zero (**0**) if the request was not queued or the time in milliseconds that the request was queued
Process instance id	Process instance id with which a contention occurred in the case of object locks or deadlocks
Remote node id	Number (instance id) for the remote node to which the request is sent
Last request bytes sent	64-bit number of bytes sent in the last request (**0** for notifications)
Last request bytes received	64-bit number of bytes received in the last request (last receipt bytes for notifications)
Last request packets sent	64-bit number of packets sent in the last request (**0** for notifications)
Last request packets received	64-bit number of packets received in the last request (last receipt packets for notifications)
Class id	Class number for the object identifier
Instance id	Instance number for the object identifier
Owner class id	Owner object class number for the object identifier
Sub level	Class hierarchy level number for the object identifier
Sub id	Subobject number for the object identifier
Edition	Edition number for the object identifier
Invocation mode	For details, see "Statistics File Request Values", in the following section
Method class id	Object identifier of the executing method
Method instance id	Instance number for the object identifier of the executing method
Method owner class id	Owner object class number for the object identifier of the executing method
Method sub level	Class hierarchy level number for the object identifier of the executing method
Method sub id	Subobject number for the object identifier of the executing method
Method edition	Edition number for the object identifier of the executing method
Receiver class id	Object identifier of the receiver
Receiver instance id	Instance number for the object identifier of the receiver
Receiver owner class id	Owner object class number for the object identifier of the receiver
Receiver sub level	Class hierarchy level number for the object identifier of the receiver
Receiver sub id	Subobject number for the object identifier of the receiver
Receiver edition	Edition number for the object identifier of the receiver
Volatility	For details, see "Statistics File Request Values", in the following section
Lock Kind	For details, see "Statistics File Request Values", in the following section

Statistics File Request Values

The request values that can be output to the statistics file during the sampling of individual requests are described in the following subsections.

Request Types

The request type values that can be output in the request statistic are listed in the following table.

Request Type	Integer Value
GET_OBJECT	1
LOCK_OBJECT	2
UNLOCK_OBJECT	3
GET_OBJECT_GROUP	4
LOCK_OBJECT_GROUP	5
UNLOCK_OBJECT_GROUP	6
PUT_OBJECT	7
GET_OID	8
GET_EDITION	9
BEGIN_NOTIFICATION	11
END_NOTIFICATION	12
CAUSE_EVENT	13
RETURN_NOTES	14
RECEIVED_NOTIFICATION	15
BEGIN_TRANSACTION	16
END_TRANSACTION	17
ABORT_TRANSACTION	18
REMOTE_METHOD_CALL	19

Buffer Read Mode

The mode values that can be output in the **GET_OBJECT** and **LOCK_OBJECT** requests are listed in the following table.

Mode Number	Integer Value
NEW_BUFFER_REQUEST	1
NON_UPDATE_BUFFER_REQUEST	2
UPDATE_BUFFER_REQUEST	3
TEMPORARY_BUFFER_REQUEST	4

Buffer Put Mode

The mode values that can be output in the **PUT_OBJECT** requests are listed in the following table.

Mode Number	Integer Value
BUFFER_CREATED	1
BUFFER_UPDATED	2
BUFFER_DELETED	3

Buffer Unlock Mode

The mode values that can be output in the **UNLOCK_OBJECT** requests are listed in the following table.

Mode Number	Integer Value
UNLOCK_INDIVIDUAL	1
REMOVE_LOCKS	2

Remote Method Call Mode

The mode values that can be output in the **UNLOCK_OBJECT** requests are listed in the following table.

Mode Number	Integer Value
STANDARD_METHOD	1
DATABASE_METHOD	2

Buffer Flag

The mode values that can be output in the **flag** field of requests are listed in the following table.

Mode Number	Integer Value
EMPTY	0
INUSE	1
TEMPORARY	2
UPDATED	3
CREATED	4
PERMANENT	5
DELETED	6
VERSIONED	7

Invocation Mode

The invocation mode values that can be output in the request statistics are listed in the following table.

Invocation Mode	Integer Value
NORMAL (for example, an inherited object)	0
CONSTRUCTOR	1
DESTRUCTOR	2
MAP_GET	3
MAP_SET	4

Object Volatility

The object volatility values that can be output in the **GET_OBJECT**, **LOCK_OBJECT**, and **PUT_OBJECT** requests are listed in the following table.

Volatility Value	Integer Value
OBJECT_VOLATILITY_VOLATILE	0
OBJECT_VOLATILITY_FROZEN	4
OBJECT_VOLATILITY_STABLE	8

Lock Kind

The lock kind values that can be output in the **LOCK_OBJECT** and **UNLOCK_OBJECT** requests are listed in the following table.

Lock Kind Value	Integer Value
LOCK_KIND_PROCESS_LOCK	0
LOCK_KIND_NODE_LOCK	2

Cache Statistics

The **cache statistics** record (type **1**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process

Statistic	Description
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request
Cache type	For details, see "Cache Type", later in this chapter
Hits	64-bit number of cache requests to existing buffers
Misses	64-bit number of cache requests to non-existing buffers
Top of LRU hits	64-bit number of requests where the buffer was at the top of the LRU (least-recently used) chain
Created buffers	64-bit number of buffers created
Clean swapped buffers	64-bit number of buffers swapped with no put operation
Dirty swapped buffers	64-bit number of buffers swapped with put operation
Resized buffers	64-bit number of buffers resized
Maximum buffer size	Maximum size (in bytes) allowed for the cache
Total number of buffers	Number of current buffers in cache
Available buffer size	Current available size (in bytes)
Maximum overdraft buffer size	Maximum size (in bytes) allowed for the process overdrafts
Overdraft buffer size	Current overdraft size (in bytes) for process overdrafts
Total cumulative operations	64-bit number of total cumulative logical operations for all buffers (a logical operation is equivalent to an access in cache)
Total current operations	64-bit number of total logical operations for all buffers currently in cache
Deleted buffers	64-bit number of buffers deleted (that is, removed from cache)
Dead buffers	64-bit number of total cumulative buffers marked as dead (that is, pending deletion)
Copied buffers	64-bit number of buffers that were copied due to being resized
New buffers	64-bit number of buffers created for new objects
Total fetches	64-bit number of times an object was fetched from the database to be placed in cache
Duplicate fetches	64-bit number of times an object was fetched from the database but not copied into the cache because the same or later edition was already in cache
Stubbed buffers	64-bit number of times buffers containing an uncommitted update were converted to stubs (that is, minimally-sized buffers) instead of being swapped out of cache
Swapped buffers	64-bit number of times buffers were removed from cache to make room for other buffers

Statistic	Description
Total operations when swapped	64-bit number of total logical operations for all swapped buffers (that is, removed from cache to make room for other buffers)
Minimum operations when swapped	64-bit number
Maximum operations when swapped	64-bit number
Total age when swapped	64-bit number of total age for all swapped buffers (age is measured in node ticks elapsed since the buffer was created)
Minimum age when swapped	64-bit number
Maximum age when swapped	64-bit number
LRU traversals	64-bit number of recordings for the top of LRU processing to the bottom position
Total LRU traversal ticks	64-bit number of total elapsed node ticks for all LUR traversals
Latest LRU traversal ticks	64-bit number of node ticks elapsed for the latest recording of top of LRU processing to the bottom position
Total cache coherency notifications	Total number of cache coherency notifications received
Cache coherency notification hits	Number of cache coherency notifications that resulted in objects in cache being made obsolete
Cache coherency updated objects	Number of objects that have been updated
Cache coherency object hits	Number of objects listed in cache coherency notifications that were found in cache
Cache coherency object misses	Number of objects listed in cache coherency notifications that were not found in cache
Cache coherency range requests	*Only* for internal use by JADE
Node lock remove requests received	Number of requests received to remove a node lock
Node lock remove requests sent	Number of requests sent to remove a node lock
Node lock swap out requests sent	Number of requests sent to remove a node lock due to a stable object being removed from the cache

For details about the **JadeDynamicObject** property names of these statistics for the **Node** class **getObjectCaches** method, see "Getting Cache Activity Information", earlier in this chapter.

An *LRU traversal* represents the time, measured in elapsed node ticks, it would take a buffer at the top if the LRU chain to progress to the bottom position, assuming it is not referenced again.

Cache Buffer Activity

The **cache buffer activity** record (type **2**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system

Statistic	Description
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
Cache type	For details, see "Cache Type", later in this chapter
Buffer activity	For details, see "Cache Activity", later in this chapter
Order	Order in the LRU (least-recently used) list for the buffer
Birth time	64-bit number of node ticks when the buffer was created
Operations	64-bit number of requests (hashes) for the buffer during its lifetime
Class id	Class number for the object identifier
Instance id	Instance number for the object identifier
Owner class id	Owner object class number for the object identifier
Sub level	Class hierarchy level number for the object identifier
Sub id	Subobject number for the object identifier
Edition	Edition number for the object identifier
Object size	Object record size
Flag	For details, see "Buffer Flag", earlier in this chapter
Status	For details, see "Object Buffer Status", later in this chapter
Semantic category	For details, see "Semantic Category", later in this chapter
Physical category	For details, see "Physical Category", later in this chapter
Buffer size	Total number of bytes occupied in cache for the object
Buffer process number	Process number associated with the buffer
Shared	*Only* for internal use by JADE
Replica	*Only* for internal use by JADE
Nail count	*Only* for internal use by JADE
I/O count	*Only* for internal use by JADE
Internal flags	*Only* for internal use by JADE
Lives	*Only* for internal use by JADE

For type **2** records, the order value for non-shared transient objects is always zero (**0**). In addition, the process number, process CPU time, process ticks, and process logical clock values for the type **2** records for non-shared transient objects are those of the process that creates the object, rather than those of the process making the sampling request.

Cache Type

The cache type statistics that can be output in record type **1** are listed in the following table.

Cache Type	Integer Value
PERSISTENT_CACHE	1
TRANSIENT_CACHE	2
REMOTE_TRANSIENT_CACHE	3

Cache Activity

The cache buffer activity statistics that can be output in record type **2** are listed in the following table.

Activity Number	Integer Value
BUFFER_CREATED (for objects created in the node)	1
BUFFER_ALLOCATED (for object to be read into the node)	2
BUFFER_REMOVED (removed from cache other than swapped)	3
BUFFER_SWAPPED (buffer still wanted but cannot be fitted in cache)	4
BUFFER_REALLOCATED (variable length buffer reallocation)	5
BUFFER_SNAPSHOT (contents logged as part of a full cache dump)	6

Object Buffer Status

The object buffer status activity statistics that can be output in record type **2** are listed in the following table.

Buffer Status	Integer Value
UNKNOWN	0
CURRENT_SHARED	1
CURRENT_RESERVED	2
CURRENT_EXCLUSIVE	3
OBSOLETE	4

Semantic Category

The semantic category activity statistics that can be output in record type **2** are listed in the following table.

Semantic Category	Integer Value
SYSTEM_META_OBJECT	1
SYSTEM_GUI_OBJECT	2
SYSTEM_OBJECT	3
USER_META_OBJECT	4
USER_GUI_OBJECT	5

Semantic Category	Integer Value
USER_OBJECT	6
UNKNOWN_OBJECT	7

Physical Category

The physical category activity statistics that can be output in record type **2** are listed in the following table.

Physical Category	Integer Value
Normal Object	1
Collection Header	2
Collection Block	3
Blob or Slob	4

User Command

The **user command** record (type **13**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system
Process number	Process number relative to the system
Clock ticks	64-bit number of microseconds elapsed since the node was initialized
Node CPU time	64-bit number of microseconds of CPU time charged for node
Node ticks	64-bit number of logical operations performed by the node
Process CPU time	64-bit number of microseconds of CPU time (including start process CPU time and end process CPU time) charged for the process
Process ticks	64-bit number of logical operations performed by the process
Process logical clock	Sequential 64-bit number for inter-node process context switches
User number	Number supplied by the write sample request
User text	Text supplied by the write sample request
User command	Text supplied by the user command request

File Trailer

The **file trailer** record (type **20**) is listed in the following table.

Statistic	Description
Record type	Integer value representing the type of statistics sampling record
Node number	Node number relative to the system

Statistic	Description
Date	Date on which the sample terminates converted to the internal (Julian day) format for internal storage and computation
Time	Time in milliseconds from midnight UTC (that is, 00:00) on that date until the sample terminates
Timestamp string	Text representation of the date and time as a timestamp (in the local time of the workstation from which the sample was executed)

For details about obtaining UTC times, see the **Application** class **currentUTCBias** and **getUTCTime** methods, in Chapter 1 of your JADE *Encyclopaedia of Classes*. For details about converting UTC times, see the **TimeStamp** type **localToUTCTime**, **localToUTCTimeUsingBias**, **utcToLocalTime**, and **utcToLocalTimeUsingBias** methods, in your *JADE Encyclopaedia of Primitive Types*.

The UTC bias is the UTC time minus local time, in minutes. For example, the New Zealand UTC bias of –720 indicates that a machine in New Zealand executing node sampling is 12 hours ahead of Coordinated Universal Time.

Note Greenwich Mean Time, or GMT, has been replaced as the world standard time by Coordinated Universal Time, or 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.)

Considerations and Restrictions

In thin client modes of operation (that is, JADE Smart thin client and HTML thin client modes), statistics are gathered only from the application server on which the operations are processed. (As the presentation client or HTML client is not a JADE node and therefore does not execute processes, statistics sampling has no meaning on these remote connections.)

Note If the file cannot be opened, no sample records are written, and an exception is raised only if you subscribed to an exception in a sampling library by specifying the **SamplingExceptionEvent** number in the **command** parameter of the **Node** class **logUserCommand** method.

The total request time is rounded down, as some operating system implementations have a granularity of more than one millisecond.

Example of Code to Manually Sample Node Statistics

The following is an example of a method that manually samples node statistics.

```
testManualSamplingFullInterval();
vars
    cust           : Customer;
    samplingHandle : Integer;
begin
    samplingHandle := node.beginSample("filesmpl",
                      "c:\temp\fullInterval%p.txt");
    node.beginIndividualRequestsLogging(samplingHandle, false,
                     true, true, true, false, 557, "fullInterval");
    foreach cust in Customer.instances do
        write cust.name;
    endforeach;
    node.endIndividualRequestsLogging(samplingHandle, 557, "fullInterval");
```

```
        node.endSample(samplingHandle);
    end;
```

Direct Retrieval of Node Sampling Statistics

You can directly retrieve node sampling information, by using the **Node** class **getObjectCaches** method to retrieve recorded node sampling values relating to cache activity or the **Process** class **getRequestStatistics** method to retrieve recorded node sampling values relating to the current process.

For more details, see "Getting Cache Activity Information" and "Process::*getRequestStatistics* Method", earlier in this chapter.

Initializing the *filesmpl* and *tcpsmpl* Sampling Libraries

When using the **filesmpl** or **tcpsmpl** JADE sampling library, you can specify **"<null>"** or **""** for the **SamplingLibraryInitialization** parameter in the [JadeClient] and [JadeServer] sections of the JADE initialization file and the **initializationParameter** parameter of the **Node**::**beginSample** and **System**::**beginSample** methods.

If you specify a value of **"<null>"** or **""**, the **filesmpl.dll** and **tcpsmpl.dll** JADE sampling libraries (documented under "JADE Sampling Libraries", earlier in this chapter) will not output any sample values.

For **filesmpl**, the values will not be written to a file. For **tcpsmpl**, the values will not be sent to a TCP/IP connection. Use this option in situations where node sampling needs to be enabled for the **Process** class **getRequestStatistics** method but no file or TCP/IP output is wanted.

Using the JADE Sampling Application

The JADE Monitor provides functionality that enables you to initiate node sampling and to analyze results.

The separate JADE Sampling application is also available for initiating node sampling.

Note For a full explanation of all the functionality available with JADE node sampling, refer to the earlier part of this chapter, which you should read before you use this JADE Sampling application.

Accessing the JADE Sampling Application

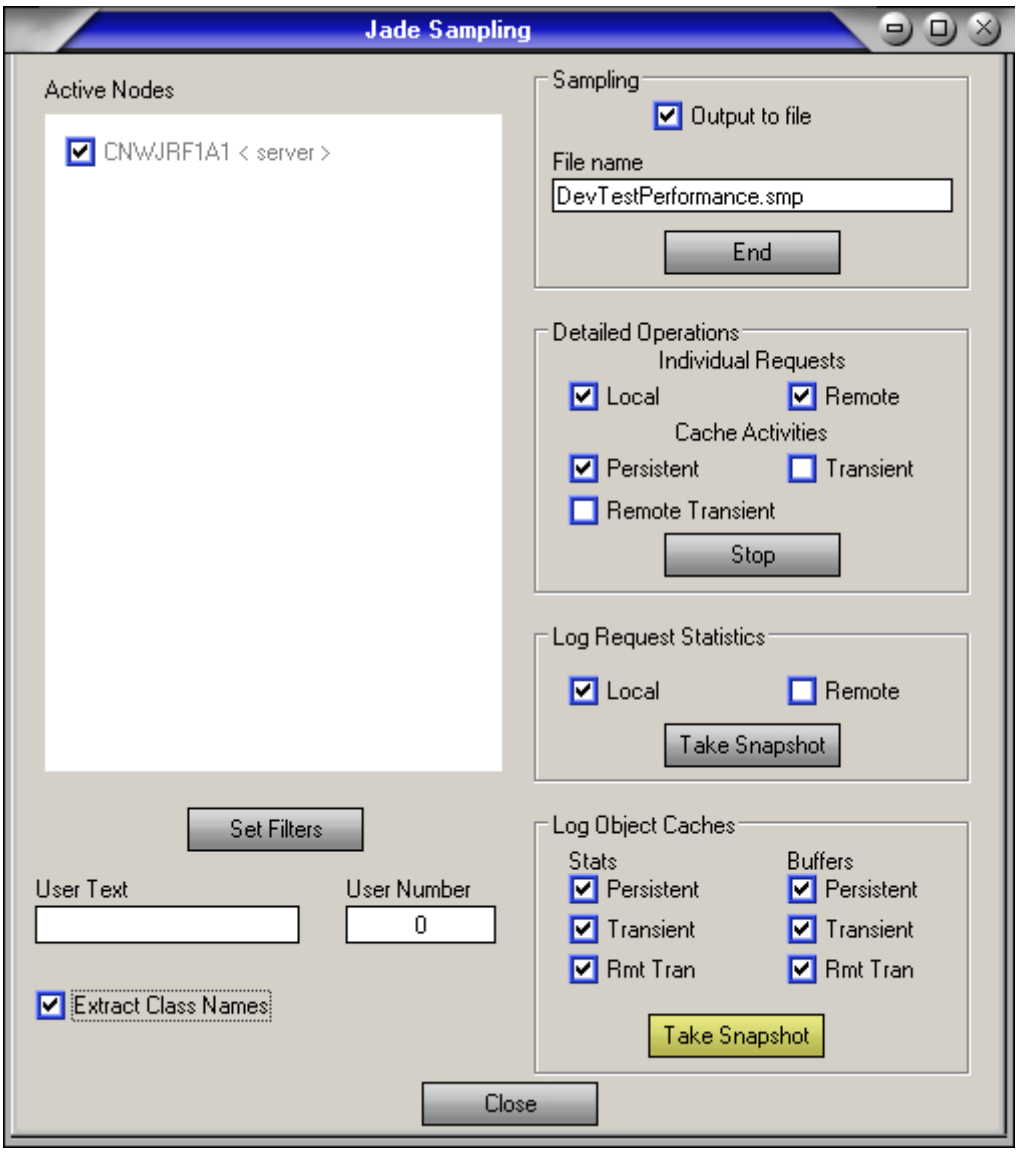
» To access the JADE Sampling application

1. Set up an icon, or shortcut, in the same way that you would for the JADE Monitor.
2. In the command line **app** parameter, change the application name to **JadeSampling**; for example:

   ```
   D:\JADE\bin\jade.exe path=d:\jade\system schema=JadeMonitorSchema
   app=JadeSampling
   ```

3. To run the standalone JADE Sampling application, select the **JadeSampling** program icon from your JADE program folder.

The Jade Sampling dialog, shown in the following image, is then displayed.



Selecting Your JADE Sampling Options

The JADE Sampling application provides the Jade Sampling and the Jade Sampling Class Selector dialogs, to enable you to select your JADE sampling options.

For details, see the following subsections.

Using the Jade Sampling Dialog

The Jade Sampling dialog enables you to select the options that you require for sampling, as follows.

1. The **Active Nodes** panel displays all currently active nodes on the system and is dynamically updated whenever a node begins or shuts down.

To select the nodes that are to be sampled, simply check the appropriate check boxes. (As node sampling is disabled by default, no active nodes are checked when the JADE Sampling application in invoked.)

2. By default, all classes in all schemas are sampled. The Jade Sampling Class Selector dialog enables you to select specific classes that are included in the sampling context, by clicking the **Set Filters** button.

   For details about selecting the classes that are whose statistics are sampled, see "Using the Jade Sampling Class Selector Dialog", in the following section.

3. The **Sampling** group box enables you to choose between sending your output to a file on disk or through a TCP/IP connection.

   Unless you have written your own TCP/IP interface, use the default **Output to file** option, which is checked by default.

   Tip The **examples\sampling** directory on your JADE release medium contains sample code.

   If you do not specify a file name in the **File name** text box, your remote node sampling is output to the **jomstats.smp** file in the JADE installation directory by default.

   Caution If you want to retain an existing sampling file (for example, you have a sampling context that was output to the default file), ensure that you define a unique name for the next sampling context, as existing files are cleared if sampling is started to a file whose path and name already exist.

   To ensure that file names do not conflict, you can include one or more macro definitions, or escape characters, within the file name; for example, **c:\jade\performance\stats%p%d%t.log**. The macro definitions that you can use are listed in the following table.

Definition	Description	Example
%p	Process id	123
%d	Date	31/07/2000
%t	Time	07:05:57
%c	Computer name	statsjeken

4. To select the detailed operations that are to be sampled, check the appropriate check boxes in the Detailed Operations group box. You can select one or both of local or remote individual requests and the cache activities that you want to sample. (For details, see the **Node**::**beginIndividualRequestsLogging** method or **System**::**beginIndividualRequestsLogging**, earlier in this chapter.)

   The **Start** button is enabled only when sampling has begun, to enable you to start or stop detailed operations at any time during an active sampling context. When sampling has ended, the caption of this button is displayed as **Stop**.

5. To take a snapshot of statistics for local or remote activity or both local and remote activity (record types **8** and **9**), check the appropriate check boxes in the Log Request Statistics group box. (For details, see **Node**::**logRequestStatistics** or **System**::**logRequestStatistics**, earlier in this chapter.)

   The **Take Snapshot** button is enabled only when sampling has begun, to enable you to take snapshots of log request statistics at any time during the active sampling context. When sampling has ended, the button is disabled.

6. To take a snapshot of statistics for cache statistics or cache buffers or for both cache statistics and buffers (record types **1** and **2**), check the appropriate check boxes in the Log Objects Caches group box. (For details, see **Node**::**logObjectCaches** or **System**::**logObjectCaches**, earlier in this chapter.)

You can sample any combination of persistent, transient, or remote transient cache statistics. All buffers containing non-shared transient objects are listed when node sampling snapshots are requested.

The **Take Snapshot** button is enabled only when sampling has begun, to enable you to take snapshots of log object caches at any time during the active sampling context.

When sampling has ended, the button is disabled.

7. The **User Text** and **User Number** text boxes enable you to specify any user information that you require, which is recorded on each record. (For details, see "System Class External Methods", earlier in this chapter.)

8. The **Extract Class Names** check box, which is checked by default the first time you access the Jade Sampling dialog, enables you to extract the class names and numbers and the schema name, class name, method name, and oid of each method in all schemas to a file that defaults to **jomstats.txt** in the JADE installation directory; for example:

```
z:\jade\bin\jomstats.txt
```

9. When you have selected all of the sampling options that you require in the dialog check boxes, click the **Begin** button in the Sampling group box.

   The **Start** button in the Details Operations group box is then enabled.

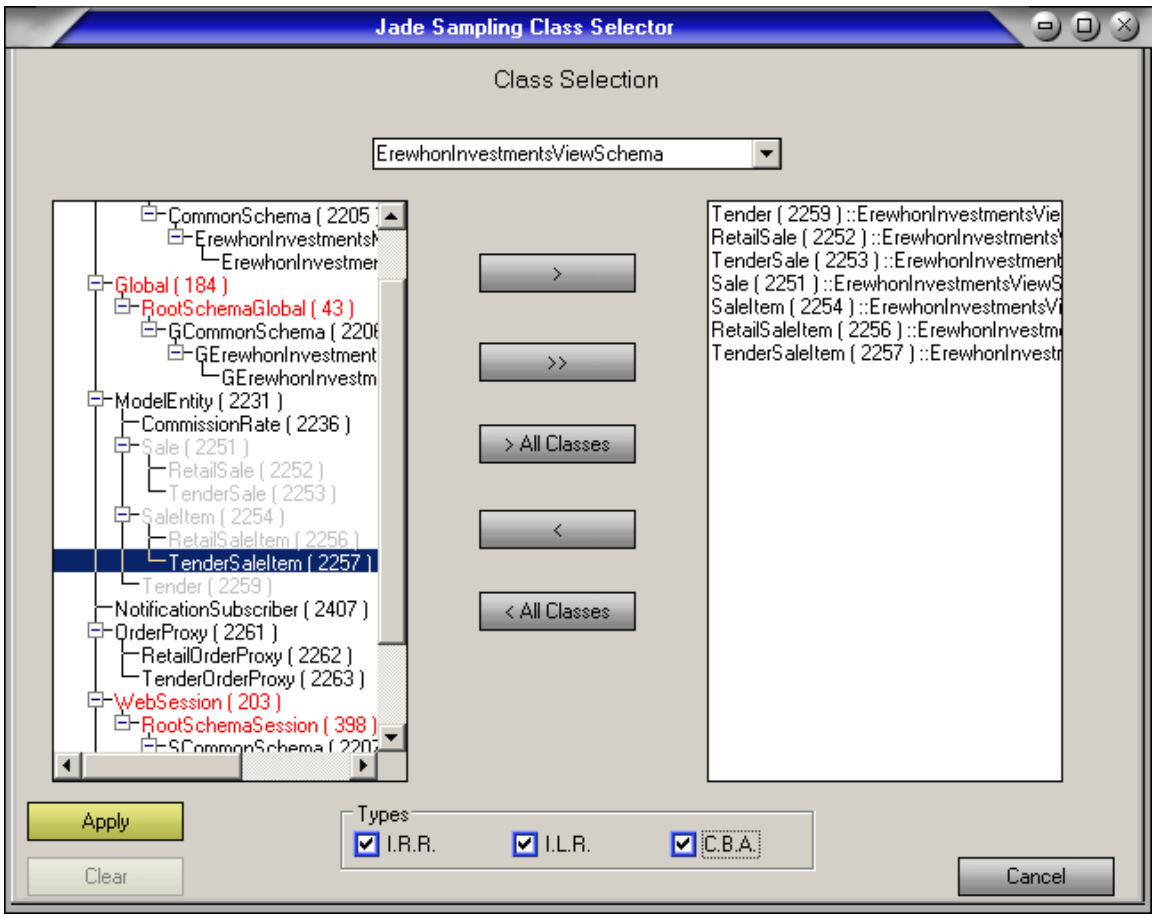
10. To start the sampling context, click the **Start** button in the Detailed Operations group box.

    The sampling context for your selected criteria is then started, the caption of the button in the Sampling group box changes to **End**, and the **Take Snapshot** buttons in the Log Request Statistics and Log Object Caches group boxes are enabled.

Using the Jade Sampling Class Selector Dialog

The Jade Sampling Class Selector dialog, accessed by clicking the **Set Filters** button in the Jade Sampling dialog, enables you to select the schema whose classes you want to filter for sampling and then select only the classes whose statistics you want sampled.

An example of the Jade Sampling Class Selector dialog is shown in the following image.



To select only the classes that you want included in the sample

1. In the drop-down list box in the top center of the dialog, select the schema whose classes you want to select for inclusion in the sample. The class hierarchy for that schema is then displayed in the list box at the left of the dialog.

2. From the class hierarchy, select one or more classes that you want included in the sampling context, and then click the **>** button if you want to add the selected class to the sample, or click the **>>** button if you want to add the selected class and all of its subclasses to the sample.

   Although all classes in all schemas are sampled by default, when you select a schema in this dialog, all classes are automatically excluded from the sampling context; that is, they are displayed in the class hierarchy in the left list box and are included in the sample only when you select the appropriate classes and click a button to move them to the right list box for inclusion in the sampling context.

   Alternatively, click the **> All Classes** button if you want to sample all classes.

3. Reverse the actions described above in step 2 if you want to remove a class or classes from the sampling context.

4. Select at least one of the sample types in the check boxes in the Types group box.

These types are:

- **I.R.R** (Individual Remote Requests)
- **I.L.R** (Individual Local Requests)
- **C.B.A** (Cache Buffer Activity)

5. Select other schemas and add additional classes to the list on the right of the dialog for inclusion in the sampling context, if required.
6. Click the **Apply** button to retain your selections. Alternatively, click the **Clear** button to clear all previously applied selections. (The **Clear** button is enabled only if you previously selected classes in this dialog and you retained your selected classes by clicking the **Apply** button.)
7. To return to the Jade Sampling dialog, click the **Cancel** button.

   All selected classes that you applied are then retained for sampling when you next begin a sampling context. (If you did not apply your selected classes, any selections that you made are abandoned.)

Note The filtering selections are invoked when you click the **Begin** button in the Sampling group box. This enables you to set up another class filter selection while a sampling context is active.

Chapter 5 JADE Logical Certifier Diagnostic Utility

This chapter covers the following topics.

Overview

The **RootSchema** provides the JADE Logical Certifier diagnostic utility, which performs various database consistency checks, particularly the validation of inverse references and collections. You can check the referential integrity of user data and the user meta schema definition.

The logical certification process analyzes each class in a schema and performs various cross-checks on each instance, to verify its logical integrity.

Certify your schemas in production systems regularly as data is updated, so that any problems can be detected as soon as possible. (You can certify schemas in a development environment less frequently, as data is generally more stable.)

Certify meta data in:

- Production systems occasionally, especially following the deployment of schema changes.
- Your development environment regularly, especially during the development phase. The development environment in which meta data is certified must be a stable system (that is, one in which no structural changes are being made).

Although you could certify meta data in a backed up version of your JADE development environment, any fixes that are required apply only to a system in the same structural state.

Note You should check the meta-structure of schemas only when schemas are not versioned and you are running in single user mode. If entities are versioned, only the current version is checked for integrity.

You should use the JADE Logical Certifier utility in addition to the **Certify Files** command in the JADE Database utility, which checks the physical integrity of your database files.

Any errors that are detected are reported in the **_logcert.err** file in the specified output log directory.

If you want to perform an online repair of a collection while it is concurrently updated, call the **repairCollection** method of the **Application** class to remove invalid object references and fix up dictionary keys in a specified collection, including dynamic dictionaries.

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

Note 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.

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

The JADE Logical Certifier:

- Writes meta data certification output to **_metacert.*** files, except for the fix file, which is output as **_logcert.fix**.
- Appends output to the **_logcert.log** file and the **_repair.log** file, rather than overwriting them.
- Uses the **JadeLog** directory instance as the default value of the **Log File Directory** text box on the Jade Logical Certifier dialog.

Installation Requirements

The JADE Logical Certifier utility generates a transient method to certify each class. This method can be large, depending on the number of references, inverses, and subobjects that are defined in the class. It is therefore important that you configure a sufficiently large method cache to run the certifier efficiently.

As a rough guide, to certify a class containing approximately 200 references, inverses, and subobjects, the method generated would require approximately 10M bytes of transient cache and approximately 10M bytes of method cache to execute efficiently. You should ensure that your transient cache size (specified in the **TransientCacheSizeLimit** parameter in the [JadeClient] section of the JADE initialization file) and your method cache size (specified in the **MethodCacheLimit** parameter in the [JadeInterpreter] section of the initialization file) are of sufficient size to accommodate certification overheads and the affect of multiple certify workers, if used.

Running the Diagnostic Tool

You can run the JADE Logical Certifier utility as a **RootSchema** application, by using the GUI **JadeLogicalCertifier** application or the non-GUI **JadeLogicalCertifierNonGui** application from the **jadclient** program. For details about the **jadclient** program, see "Running a Non-GUI Client Application using jadclient", in Chapter 1 of the *JADE Runtime Application Guide*.

Running a Non-GUI JADE Logical Certifier

To run the JADE Logical Certifier as a non-GUI application, specify the following parameters in the **jadclient** program.

```
jadclient path=database-path
          ini=c:\jade\system\jade.ini
          schema=RootSchema
          app=JadeLogicalCertifierNonGui
          server=SingleUser
          endJade
          operation=certify|certifyMeta|repair
```

```
logDir=log-file-disk-path
[progress=true|false]
[instances=number-of-progress-instances-output]
[workers=number-of-worker-processes-available]
```

Running the **jadclient** program with these parameters initiates a certification of all schemas in the database or the meta-structure of all schemas.

To avoid possible inconsistencies and contention with running applications, the JADE Logical Certifier tool must be run in single user mode when certifying user schemas. It can be run in multiuser mode to perform **meta certify** or **repair** operations.

Note It is not advisable to perform **repair** operations to meta data in multiuser mode.

The **logDir** parameter specifies the directory to which the log files are output (for example, **logDir=c:\jade\system\diagnostics**). An exception is raised if this directory does not exist or you do not specify a value. Output is appended to the **_logcert.log** file and the **_repair.log** file.

Notes Path names must be valid and cannot contain spaces.

The **jadclient** program treats processing arguments enclosed in double (**""**) or single (**''**) quotation marks after the **endJade** parameter as single-string entries in the huge string array. The handling of strings in this huge string array is application-specific. For example, **dir= "program files"** is treated as a two-string entry and **"dir= program files"** is treated as a one-string entry. How these entries are handled is determined by your application.

The values of the **operation** and **logDir** parameters that follow **endJade** are case-sensitive; that is, **certify**, **certifyMeta**, or **repair**.

By default, a progress output window is displayed as the classes are analyzed and instances validated. The optional **progress** parameter controls whether certify progress messages are output to the progress output window. Specify this parameter with a value of **false** if you do not want certify progress messages output.

You can control the frequency of certify progress messages by specifying the number of instances in the optional **instances** parameter. By default, messages are output every 10,000 instances.

For the **certify** operation only, and only when no **_logcert.in** input file is used, you can control the number of worker processes available if you want all classes to be processed in parallel by specifying the optional **workers** parameter. By default, a single worker process is used. When you specify that more than one worker process can be used, the elapsed time to certify a system *may* be reduced. (This parameter is ignored if it is not a **certify** or **meta certify** operation, or a **_logcert.in** input file is used.)

Note The number of worker processes will depend on a multitude of different factors, including things such as available I/O bandwidth, CPU cores, the number of instances, and the number of inverses in each class. As a guide, the number of workers should not exceed the number of CPU cores minus one.

When performing a certify operation, you can create a **_logcert.in** input file in the output directory, adding lines in the following format to specify your logical certification filter criteria, as follows.

- Individual classes to be certified, by specifying the following lines.

```
Schema <schema-name-1>
Class <class-name-1>
Class <class-name-2>
:
:
Schema <schema-name-2>
Class <class-name-3>
```

```
Class <class-name-4>
:
```

When you specify the **Class** command with the *class-name* parameter value, all instances of the specified class are checked (for example, **Class Customer**). You can filter the instances of the class specified in the *class-name* parameter by specifying the instance or instances that you want to check, as listed in the following table.

Parameter	Checks…
Class *class-name first-instance*	Instances of the specified class whose instance id is the value of the ***first-instance*** parameter or higher (for example, **Class Customer 23467**)
Class *class-name first-instance last-instance*	Instances of the specified class whose instance id is between the specified values of the ***first-instance*** and ***last-instance*** parameters (for example, **Class Customer 23467 33000**)

For a meta-certify operation, you can use the **_metacert.in** file to specify the **RootSchema** classes that are to be certified.

Note The format of the **_metacert.in** file is the same as that of the **_logcert.in** file except that the **Schema** and **AllSchemas** values are not valid, because the certification of meta data is performed at **RootSchema** level so you cannot filter the schema selection.

- All classes in a specific schema, by specifying the following lines.

```
Schema <schema-name>
AllClasses
```

- All classes in all schemas, by specifying the following line.

```
AllSchemas
```

- A specific class and all of its subclasses, by specifying the following line.

```
Subclasses <class-name>
```

- Exclusion of a specific class, by specifying the following lines that apply only when you have specified the **AllClasses** or **Subclasses** line.

```
ExcludeClass <class-name-1>
ExcludeClass <class-name-2>
ExcludeClass <class-name-3>
:
```

- Exclusion of a specific class and all of its subclasses, by specifying the following lines that apply only when you have specified the **AllClasses** or **Subclasses** line.

```
ExcludeSubclasses <class-name-1>
ExcludeSubclasses <class-name-2>
ExcludeSubclasses <class-name-3>
:
```

You can specify the following commands *before* any **AllSchemas**, **Schema**, **AllClasses**, **Class**, **Subclasses**, **ExcludeClass**, or **ExcludeSubclasses** command that applies to all classes that are certified.

Command	Checks instances of classes that are created on or after the…
DateRange *start-date*	Start date specified in the ***start-date*** parameter (for example, **DateRange 14/12/2007**)
DateRange *start-date end-date*	Start date specified in the ***start-date*** parameter and on or before the end date specified in the ***end-date*** parameter (for example, **DateRange 12-Dec-2007 14-Dec-2007**)

Note In this table, the instances checked are those *created* in the specified period; not those that are *updated* in that period.

Lines that start with two virgules, or forward slash separators (that is, *//*), are treated as comments and are ignored.

When you run the logical certifier, it creates the **_logcert.cls** file, listing classes that have errors. (This file has the same format as the **.in** file.) This enables you to check those classes after repairs have been made, by copying it to the **_logcert.in** file and then running the certification again.

If you want to certify all schemas, remember to delete any existing **_logcert.in** input file.

Running the JADE Logical Certifier from a Dialog

You can also run the diagnostic tool from a dialog interface, which enables you to interactively specify schemas or classes that you want to certify or repair. The **JADE Logical Certifier** icon is installed in your JADE program folder when you install JADE or upgrade to this release.

» To run the standalone JADE Logical Certifier application, perform one of the following actions

- From the JADE development environment:

  a. Select the **RootSchema** in the Schema Browser of the current schema version (if applicable).

  b. Select the **Run** command from the File menu or the **Applications** command from the Browse menu.

     Alternatively, click the **Run Application** toolbar button.

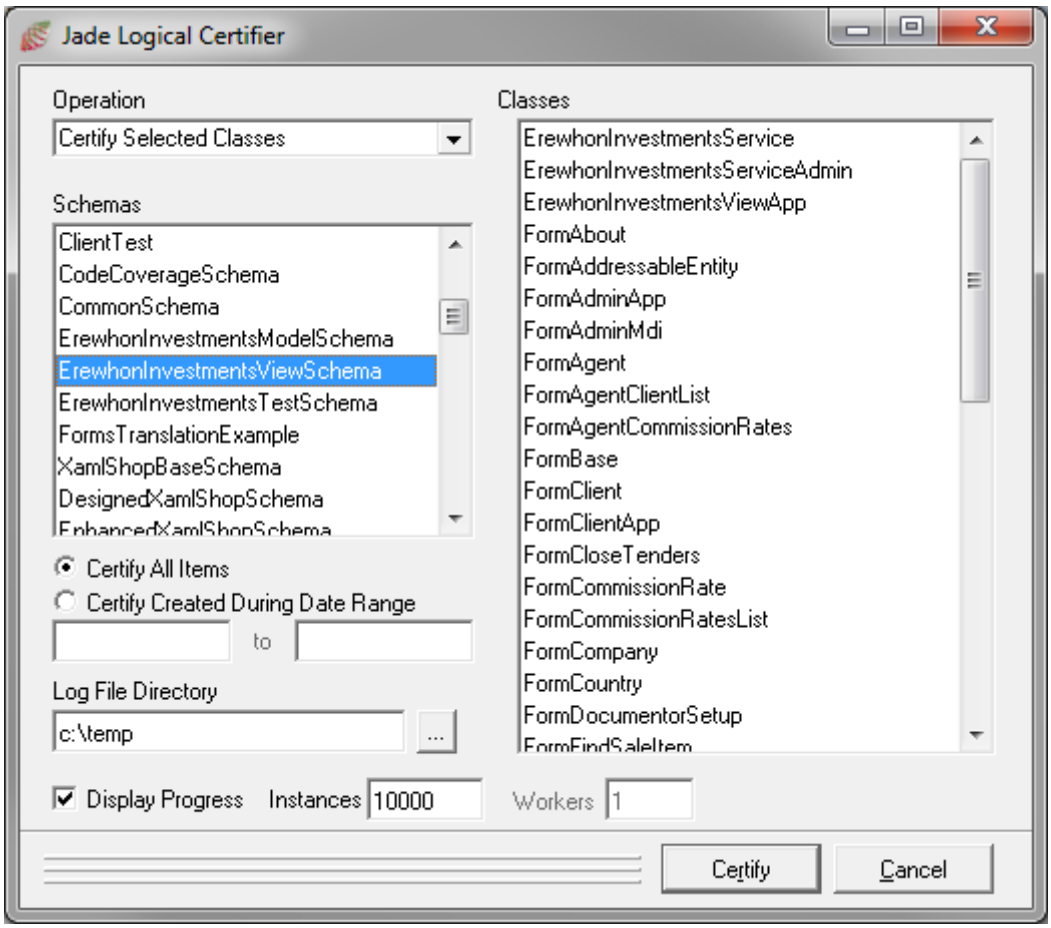
  c. From the Run Application dialog that is then displayed, select the **JadeLogicalCertifier** application the **Application Name** text box.

- Select the **JADE Logical Certifier** program icon from your JADE program folder.

  The following is an example of the command line required to run the JADE Logical Certifier tool.

```
c:\jade\bin\jade.exe path=c:\jade\system schema=RootSchema
ini=c:\jade\system\jade.ini app=JadeLogicalCertifier server=SingleUser
```

The Jade Logical Certifier dialog, shown in the following image, is then displayed.



In this example, the **ErewhonInvestmentsViewSchema** schema is selected in the **Schemas** list box, with the **Classes** list box populated with all classes in that schema so that only those classes whose referential integrity requires checking can be selected.

Although all schemas are selected for logical certification by default (that is, **Certify All Schemas** is selected in the **Operation** combo box), you can expand the certify and meta certify nodes in the **Operation** combo box to tailor the logical certification or user or meta data. Select the:

- **Certify Selected Schemas** operation if you want to select the schemas whose classes are logically certified from the list in the **Schemas** list box.
- **Certify Selected Classes** operation if you want to select one or more classes displayed in the **Classes** list box for the schema selected in the **Schemas** list box. (You can select one schema only, when using this operation.)
- **Meta Certify All Schemas** operation if you want to check the referential integrity of the schema definition. (As this operation checks the schema definition of *all* user schemas, you cannot select individual schemas in the **Schemas** list box.)
- **Meta Certify Selected Classes** operation if you want to select one or more meta classes displayed in the **Classes** list box for all user schemas. (As the certification of meta data in selected classes is performed at the **RootSchema** level, you cannot filter the schema selection for this operation.)
- **Repair** operation after you have edited the **_logcert.fix** file, to run the repair of detected errors. (As this

operation repairs all errors for which fixes are available, you cannot filter the selection of the schemas or classes for which the repair operation is run.)

When you certify all schemas or you meta-certify all schemas (that is, you have selected the **Certify All Schemas** or **Meta Certify All Schemas** option in the **Operation** combo box), the **Workers** check box is enabled, so that you can control the number of worker processes available if you want all classes to be processed in parallel. By default, a single worker process is used. When you specify that more than one worker process can be used, the elapsed time to certify a system *may* be reduced.

Note The number of worker processes will depend on a multitude of different factors, including things such as available I/O bandwidth, CPU cores, the number of instances, and the number of inverses in each class. As a guide, the number of workers should not exceed the number of CPU cores minus one.

Entities in the **Schema** and **Classes** list boxes are disabled when their selection does not apply to the selected logical certify operation.

To select a group or range of classes or schemas displayed in the **Classes** or **Schemas** list box, use the Shift key or Ctrl key, respectively.

Although all items in the selected operation are certified by default, you can select the **Certify Created During Date Range** option button and then enter the start and end creation dates of the items that you want certified. The **JADE Logical Certifier** application then checks only the items that were *created* within the specified date range (that is, updated instances are not certified).

The **JadeLog** directory instance is used as the default value for output from the logical certification. If you want to direct output to another directory, specify the required valid absolute path in the **Log File Directory** text box. Alternatively, click the adjacent browse button (indicated by the … points of ellipsis symbol) to display the common Browse for Folder dialog that enables the selection of the log file directory in which certified, checked, or repaired files will be located.

Certification data is appended to the **_logcert.log** file and the **_repair.log** file. Meta data certification output is written to **_metacert.*** files, except for the fix file, which is output as **_logcert.fix**.

If you do not want certify progress messages to be output to the Jade Interpreter Output Viewer window, uncheck the **Display Progress** check box. By default, certify progress messages are output to the viewer window. You can control the frequency with which these messages are output, by specifying the number of instances in the **Instances** text box. By default, messages are output every 10,000 instances.

Repairing Errors

After errors have been identified, you can use the JADE Logical Repairer to assist in repairing the errors.

Although many errors can be repaired without your input, some situations require you to decide between alternative repairs. In addition, situations involving duplicate keys require you to write a script to repair the error.

For large production systems, it may not be realistic to take the system off-line for the length of time taken to certify and then repair the system. In this case, you can run the JADE Logical Certifier utility on a backed up database and run the repair process on the actual database as soon as it is practical to do so.

Even though the errors that are reported and the errors that are produced relate to the state of the database when the diagnostic tool was run, the repairs contain checks that specify conditions that must still apply for the repair to be performed.

Caution It is important that no reorganization of the database has subsequently been performed, as the reorganization may invalidate the repairs.

The repair process is driven by the **_logcert.fix** file that is created when the JADE Logical Certifier utility is run. This file contains a list of repairs that need to be made and is in the following format.

```
FIXnnn: fixcommand param1 param2 ... [CHECK: checkexpression]
```

The optional **CHECK** part of the command contains conditions that must apply for the fix to be applied. You can use the ***FIXnnn*** number to relate the fixes to the report in the **_logcert.err** file. For details about the errors and repairs, see "Logical Certifier Errors and Repairs", later in this chapter.

The **fixcommands** that can be output to the **_logcert.fix** file are as follows.

- null obj prop

  This sets the **prop** reference on the **obj** object to null.

- set obj prop refobj

  This sets the **prop** property on the **obj** object to reference **refobj**.

- nullKP obj prop *or* setKP obj prop refobj

  This sets the **prop** reference on the **obj** object to null or reference **refobj** where the **prop** reference is part of a key path. Key maintenance is turned off when this repair is performed, to avoid problems with *object not found* errors that could result if key maintenance was performed.

  A **rebuild** command to rebuild any collections where this reference was part of a key should also be generated, to update the keys.

  Any **nullKP** and **setKP** command is performed before any **rebuild** command.

- rebuild coll

  As the **coll** collection contains references to objects that have been deleted or it has invalid keys, the collection is rebuilt.

- add coll obj

  This adds the **obj** object to the **coll** collection.

- remove coll obj

  This removes the **obj** object from the **coll** collection.

- setBit obj subobj

  This detects that **subobj**, a sub-object of **obj**, has been created but the creation bit of **obj** is not set so it sets the bit.

- orphan oid

  This detects collections whose parents no longer exist in any file in the database.

- orphanBlock filename

  This detects collection blocks whose parents no longer exist in any file in the database, both in user data files and user meta schema files.

- delete obj

  This deletes the **obj** object.

- deleteInstances fileName className classNumber

This deletes the abstract class name and class number instance from the database map file.

- mappingCount obj count

  This resets the mapping count of the **obj** object to count.

- fixSlobOrBlob obj prop

  This repairs the incorrect length or edition of the parent object relative to the String Large Object (slob) or Binary Large Object (blob).

- orphanSlobOrBlob imap

  This detects String Large Objects (slobs) or Binary Large Objects (blobs) whose parents no longer exist in any file in the database, both in user data files and user meta schema files.

The JADE Logical Certifier utility decides which fix is appropriate, by treating the **Manual** side of an inverse reference as being correct.

Note Orphan blocks may be created when a dictionary with missing blocks is deleted by a delete or an orphan repair. If any unreachable blocks exist prior to the collection being deleted, they will not be reported as orphans until after the collection header is deleted. A message is logged to the Logical Certify **repair.log** file if a collection that contains missing blocks is deleted. Run the Logical Certifier again to determine whether any orphan collection blocks have been created.

As the correct repair cannot be determined for **Manual/Automatic** inverse references, the repair is output as three or more lines that are commented out, as shown in the following example.

```
//FIX1: Choose either first fix if man/auto reference Class1::prop1 acts
        as auto else following 1 line(s)
//FIX1: fixcommand1 ....
//FIX1: fixcommand2 ....
```

You should choose the first repair (**fixcommand1**) if **Class1::prop1** should be treated as the automatic reference and the second repair (**fixcommand2**) if **Class2::prop2** is the automatic side of the reference. To edit these, remove the two virgule, or forward slash, separators (**//**) that act as comments from the correct repair and then delete the other two lines.

When certifying meta data, fixes are provided for errors in the 98 category; for example:

```
FIX1: set 1291.67 schemaType 1281.185 CHECK: '1281.185'.asOid.getPropertyValue
('properties').Object.Collection.includes ('1291.67'.asOid)

FIX1: delete 1281.794 CHECK: '1281.794'.asOid.Type.schema=null
```

Errors 88 and 99 are meta data errors for which no fix is provided, which is stated in the **_logcert.err** file.

After editing the **_logcert.err** file, you should run the repair by performing one of the following actions.

- Access the standalone JADE Logical Certifier utility again and selecting **Repair** in the **Operation** combo box and then clicking the **Repair** button in the Jade Logical Certifier dialog to fix the errors in those schemas or classes.

- Run the **jadclient** program with the following parameters.

```
jadclient path=database-path
          ini=c:\jade\system\jade.ini
          schema=RootSchema
          app=JadeLogicalCertifierNonGui
          server=SingleUser
```

```
endJade
operation=repair
logDir=log-file-disk-path
```

Running the **jadclient** program with these parameters, all of which you must specify, initiates a repair of all schemas in the database. To avoid possible inconsistencies and contention with running applications, the JADE Logical Certifier utility must be run in single user mode.

The **logDir** parameter specifies the directory to which the log files are output (for example, **logDir=c:\jade\system\diagnostics**). An exception is raised if this directory does not exist or you do not specify a value. A progress output window is displayed as the classes are analyzed and instances validated.

The repair process uses your selected fixes in the **_logcert.fix** file and results in a **_repair.log** file containing the repairs that are made. Any errors that are encountered are logged in the **_repair.err** file and any repairs that cannot be performed are output to the **_repair.fix** file.

When the repair completes, the **_repair.fix** file is renamed **_logcert.fix** and the existing **_logcert.fix** file is renamed **_logcertBackup.fix**. This file contains any repairs that could not be actioned, including repairs that you did not edit and any repair that produced an exception.

Exceptions raised from repairs are output to the **_repair.err** file, which contains a log of the errors.

The most-likely exceptions that are raised are 1310 exceptions that result from duplicate keys being detected when the collection is rebuilt or an object is added to a collection. The most-likely exceptions raised from repairs are output in the following format.

```
rebuildDUPLICATE coll
addDUPLICATE coll obj
```

For other exceptions, the fix is output with ERROR appended to the **fixcommand**, as follows.

```
setERROR ...
```

In such cases, you must write a JADE script to correct the data.

A warning may be given if an **add** is performed to a collection and that object already exists. This can happen if a rebuild of a collection exposes an object that is hidden because of broken keys. You can ignore these warnings.

In addition, it is possible for the **CHECK** conditions associated with the repair to no longer apply, in which case a warning is given. This can occur because the database has changed since the JADE Logical Certifier utility was run or because of another repair. You can ignore these warnings.

Rerun the JADE Logical Certifier utility after the repair has been completed, to confirm that you have repaired all errors correctly.

Logical Certifier Errors and Repairs

This section describes the errors that can be detected and repaired when using the JADE Logical Certifier utility to check the consistency of your JADE database, particularly the validation of inverse references and collections. For details, see "Overview", earlier in this chapter.

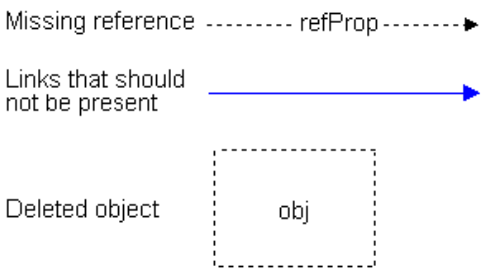
The errors that can be detected when using the Logical Certifier utility to check the referential integrity of user meta (system) schema data are:

- Error 98 – Errors to meta schema structure with fixes
- Error 99 – Errors to meta schema structure without fixes (that is, no repair is provided and it is your responsibility to repair this error yourself)

- Error 88 – Errors to versioning structure without fixes (that is, no repair is provided and it is your responsibility to repair this error yourself)

In the meta data errors, entity names are output where possible. However, in some cases the fully qualified name may not be available because the required meta schema entities required to output the full name may be null.

When using this section, the key for user data errors is as follows.

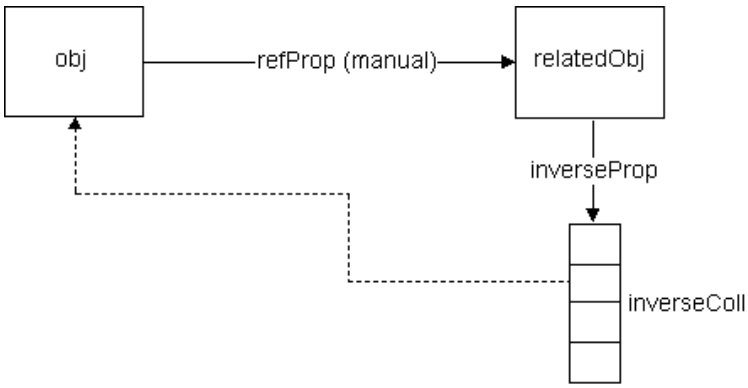


In addition, **manual repair** indicates that you can repair the error if the property **refProp** is maintained manually and **automatic repair** indicates that the error can be repaired if the property **refProp** is maintained automatically.

In each of the user data errors in the range 1 through 32 described in this section, the repair that is documented after each diagram corresponds to the **fixcommand** line in the **_logcert.fix** file.

Error 1 – Missing Reference to a Collection

An error **1** is detected when the collection does not include a reference to an inverse object.

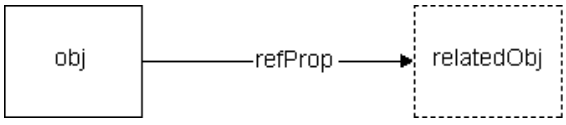


Repair:

```
add inverseColl obj
```

Error 2 – Property References an Invalid Object

An error **2** is detected when a property references an invalid object.



Repair:

If **refProp** is part of a **keyPath** then

```
nullKP obj refProp
```

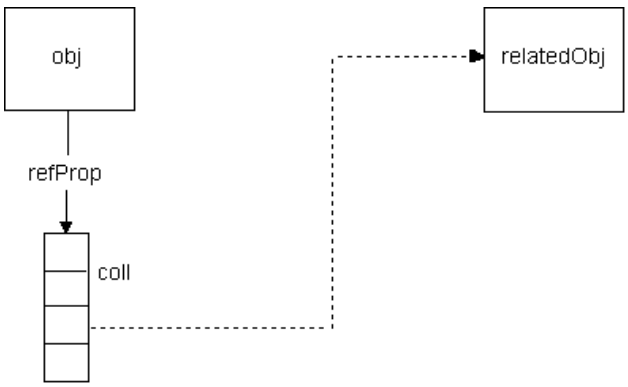
else

```
null obj refProp
```

Error 3 – Collection Contains Invalid Keys

An error **3** is detected when a collection contains an object at invalid keys.

Case A – Object was found via foreach relatedObj in coll but not via coll.includes(relatedObj)

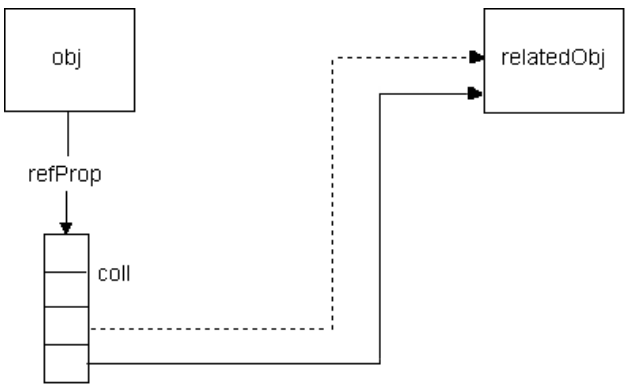


Repair:

```
rebuild coll
```

If the object has an invalid key path, it is your responsibility to repair the error yourself.

Case B – Object was found in the collection multiple times with valid and invalid keys



Repair:

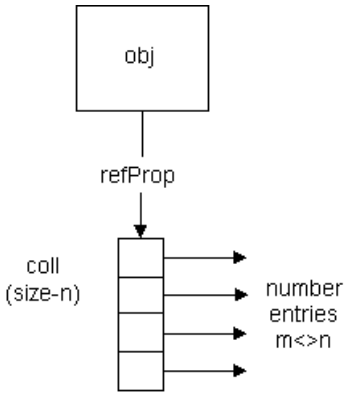
```
rebuild coll
```

If the object has an invalid key path, it is your responsibility to repair the error yourself.

Error 4 – Collection Size Mismatch

An error **4** is detected when a collection size header does not match the number of entries in the collection.

The number of entries is found when **foreach relatedObj in coll** does not agree with **coll.size**.

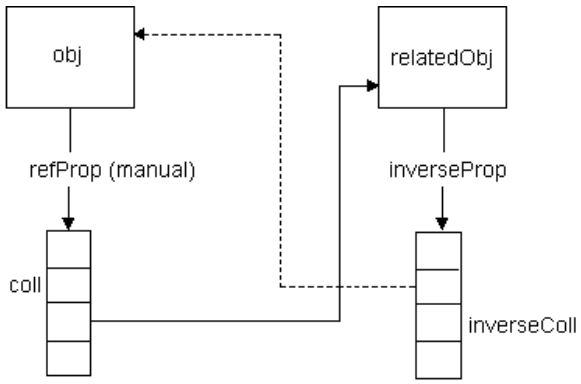


Repair:

```
rebuild coll
```

Error 5 – Collection Does Not Include Reference to Inverse Collection

An error **5** is detected when a collection does not include a reference to an inverse collection.

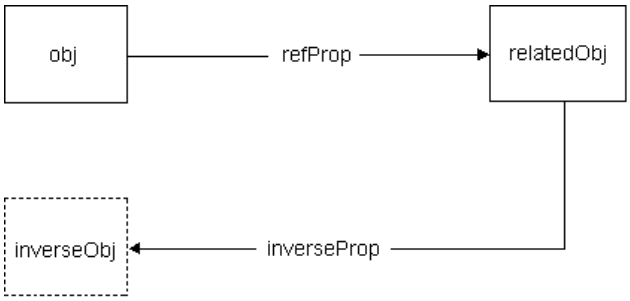


Repair:

```
add inverseColl obj
```

Error 6 – Property References an Invalid Inverse Object

An error **6** is detected when a property references an invalid inverse object.



Manual repair:

If **inverseProp** is part of a **keyPath** then

```
setKP relatedObj inverseProp obj
```

else

```
set relatedObj inverseProp obj
```

Automatic repair:

If **inverseProp** is part of a **keyPath** then

```
nullKP obj refProp
```

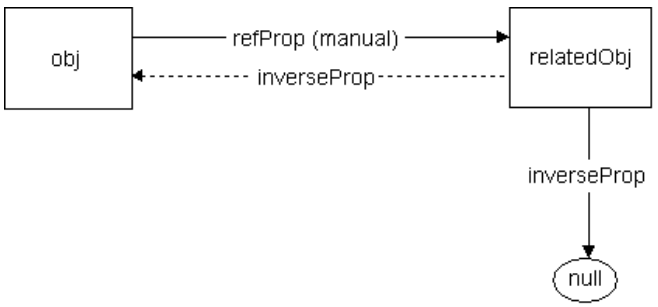
else

```
null obj refProp
```

Error 7 – Property Does Not Reference an Inverse Object

An error **7** is detected when a property does not reference an inverse object.

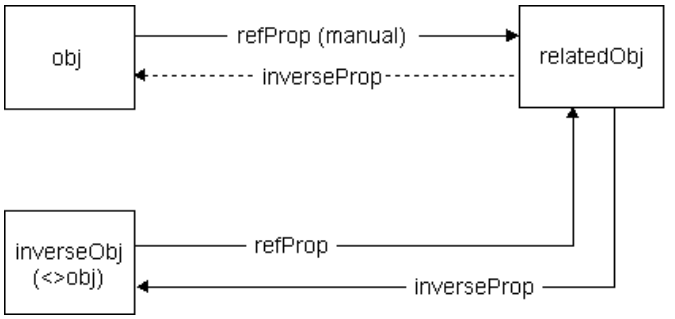
Case A – inverseObj = null



Repair:

```
set relatedObj inverseProp obj
```

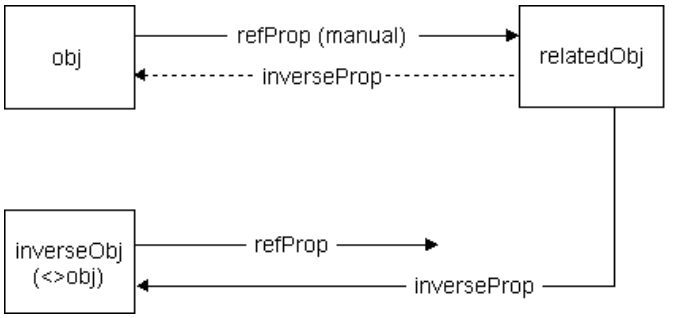
Case B – relatedObj.inverseProp <> obj and inverseObj.refProp = relatedObj



Repair:

```
null obj refProp
```

Case C – relatedObj.inverseProp <> obj and inverseObj.refProp <> relatedObj

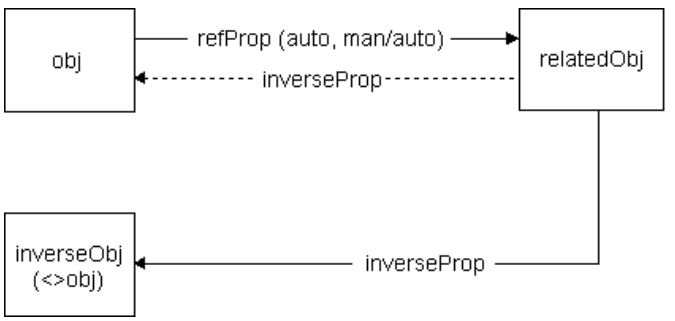


Repair:

```
set relatedObj inverseProp obj
```

Error 8 – No Inverses for Reference Property

An error **8** is detected when no inverses are found for a reference property.



Manual/automatic repair:

```
Select null or lines to populate all inverses.
```

Automatic repair:

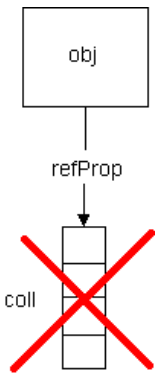
```
null obj refProp
```

Error 9

The JADE Logical Certify utility error **9** is reserved for future use.

Error 10 – Collection Has a Bad Root Block

An error **10** is detected when a collection has a bad root block.



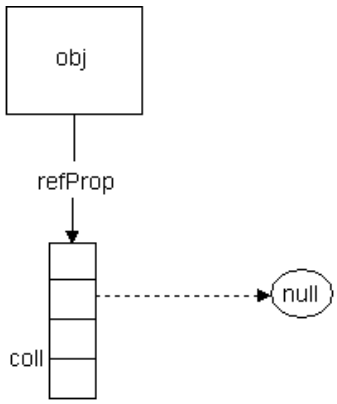
Repair:

```
rebuild coll
```

If the repair action is unsuccessful, contact your local JADE support center or JADE Support if your JADE licenses include support.

Error 11 – Collection Contains a Null Member

An error **11** is detected when a collection contains a **null** member. The collection is not an array.

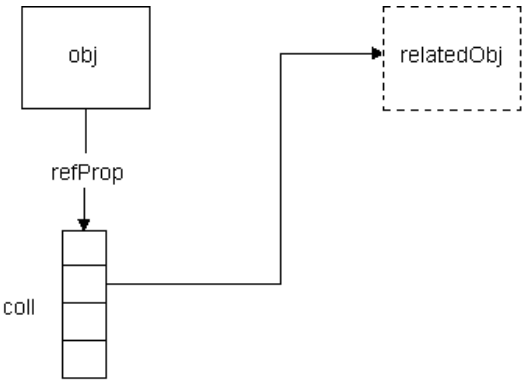


Repair:

```
rebuild coll
```

Error 12 – Collection Contains an Invalid Reference

An error **12** is detected when a collection contains an invalid reference.

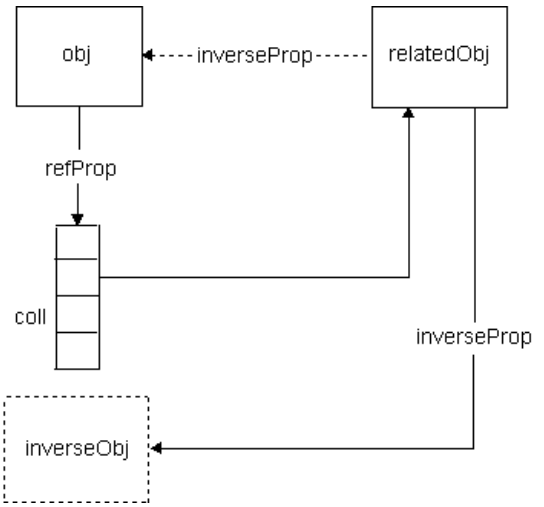


Repair:

```
rebuild coll
```

Error 13 – Collection Contains Invalid References to an Inverse Object

An error **13** is detected when a collection contains invalid references to an inverse object.



Manual repair:

```
set relatedObj inverseProp obj
```

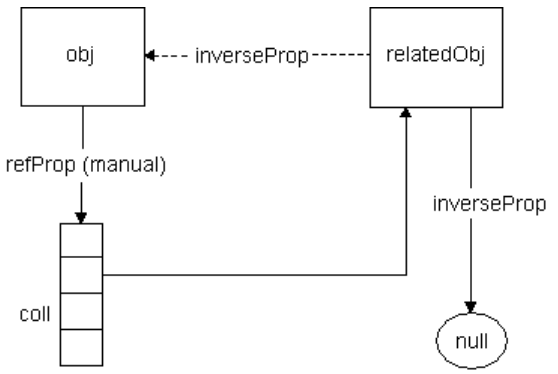
Automatic repair:

```
remove coll inverseProp
null relatedObj inverseProp
```

Error 14 – Collection Member Does Not Reference an Inverse Object

An error **14** is detected when a member of a collection does not reference an inverse object.

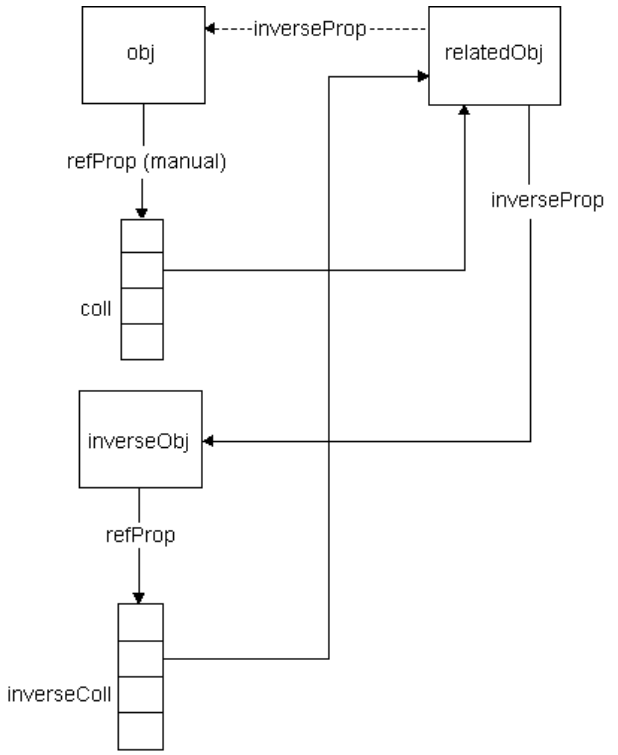
Case A – inverseObj.inverseProp = null



Repair:

```
set relatedObj inverseProp obj
```

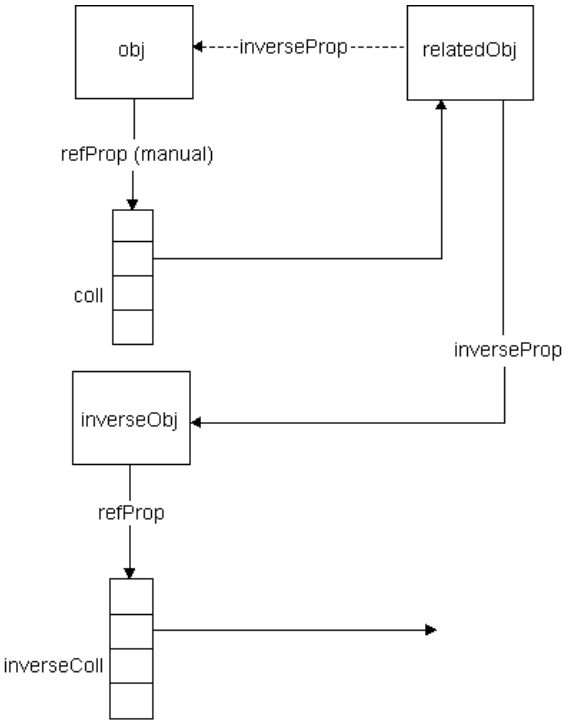
Case B – relatedObj.inverseProp <> obj and inverseObj.refProp.includes(relatedObj) = true



Repair:

```
remove coll relatedObj
```

Case C – relatedObj.inverseProp <> obj and inverseObj.refProp.includes(relatedObj) = false

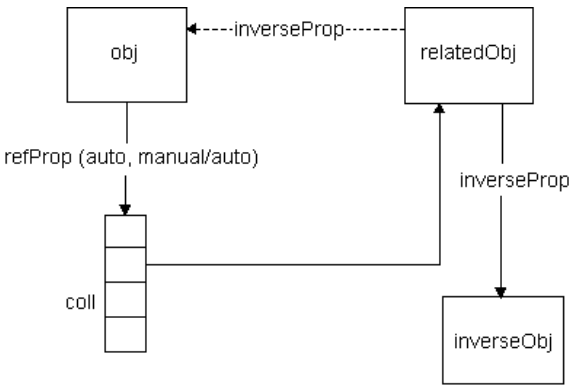


Repair:

```
set relatedObj inverseProp obj
```

Error 15 – No Inverses Found for Reference Property in a Collection

An error **15** is detected when no inverse references are found for a reference property in a collection.



Manual/automatic repair:

```
Select remove or lines to populate inverse.
```

Automatic repair:

```
remove coll relatedObj
```

Error 16 – Collection Exists but Subobject Not Set in Parent

An error **16** is detected when a collection exists but no sub-object is set in the parent object.

Repair:

```
setBit obj subObj
```

Error 17 – Object Instance References an Invalid Object

An error **17** is detected when an object instance references an invalid object.

Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 18 – Object Instance Contains a Null Reference

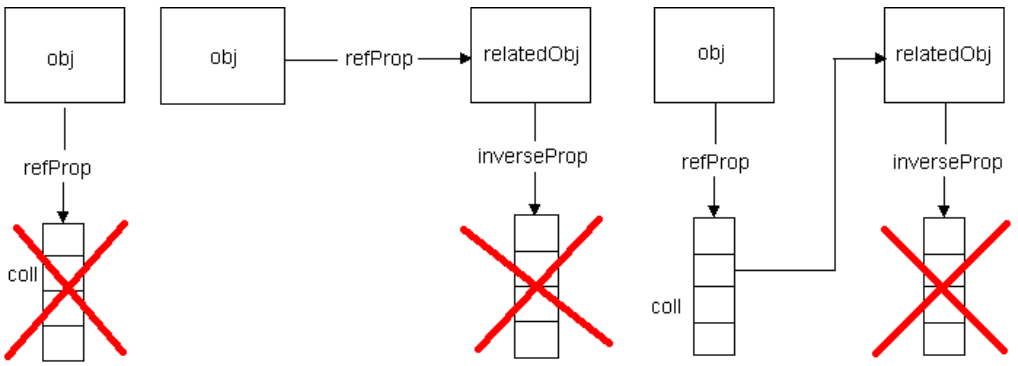
An error **18** is detected when an object instance contains a **null** reference.

Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 19 – Object Contains an Invalid Collection Reference

An error **19** is detected when an object contains an invalid collection reference.

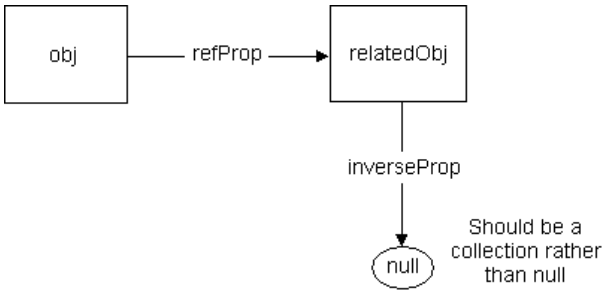


Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 20 – Object Contains a Null Collection Reference

An error **20** is detected when an object contains a **null** reference to a collection.

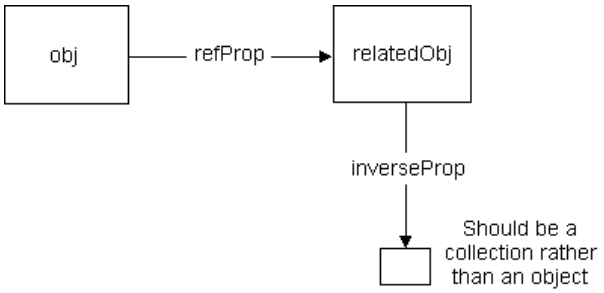


Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 21 – Object Contains Non-Collection Reference

An error **21** is detected when an object contains a reference that is not to a collection.

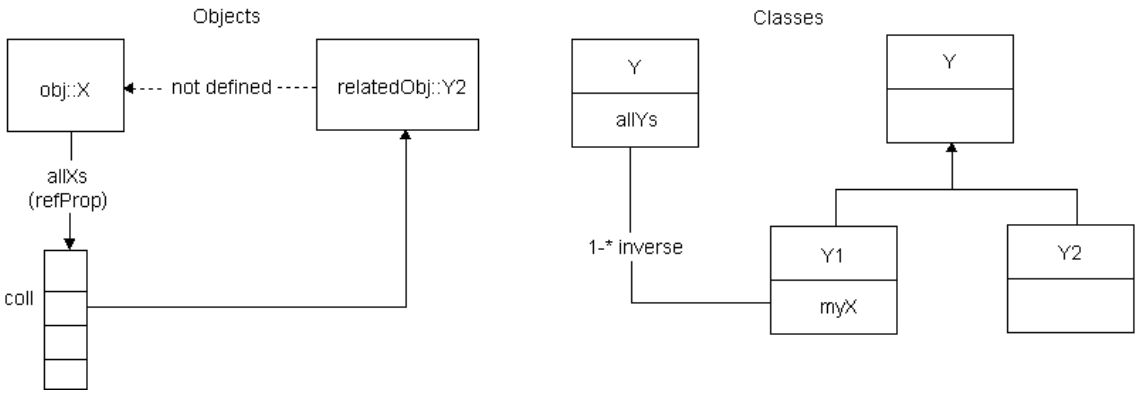


Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 22 – No Inverses are Possible in a Collection

An error **22** is detected when no inverse references are found in a collection (that is, when none is possible).

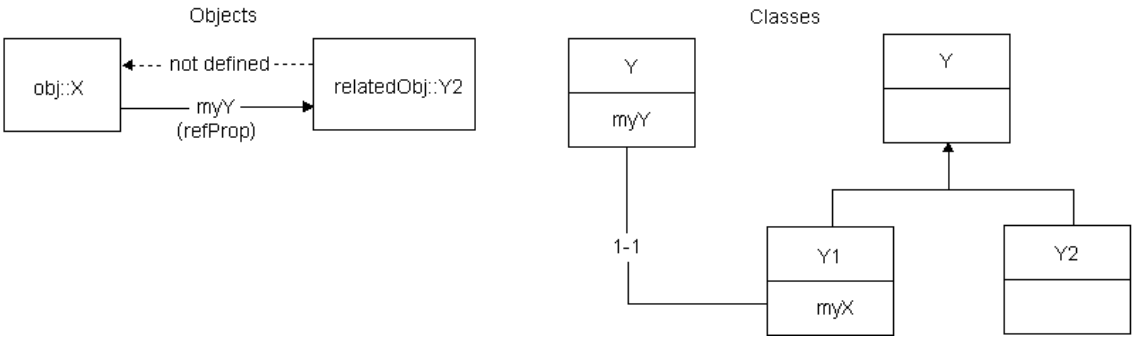


Repair:

```
remove coll relatedObj
```

Error 23 – No Inverses are Possible to a Property

An error **23** is detected when no inverse references are found to a property (that is, when none is possible).

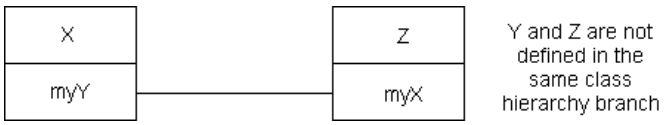


Repair:

```
null obj refProp
```

Error 24 – Invalid Inverse Definition

An error **24** is detected when an inverse has not been defined correctly in the schema.



Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Update your schema definition and remove the invalid inverse definition.

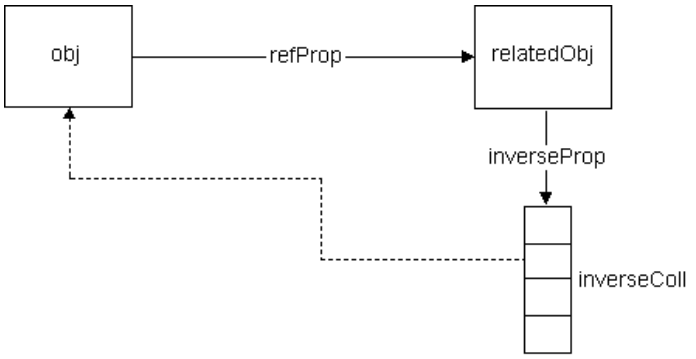
Error 25 – Exception on Includes

An error **25** is detected when an object is not found in a collection because **coll.includes(obj)** caused an exception. The possible causes are an invalid collection block or the object has an invalid key path.

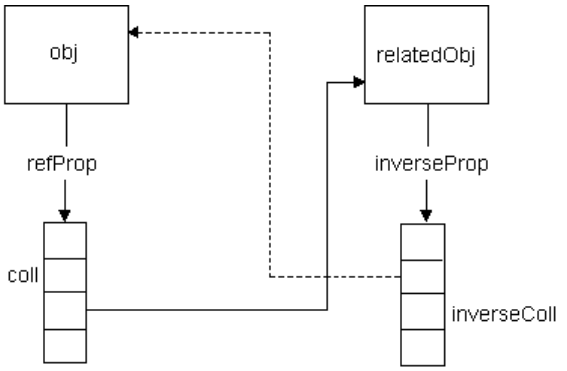
Repair:

```
rebuild inverseColl
```

Case A

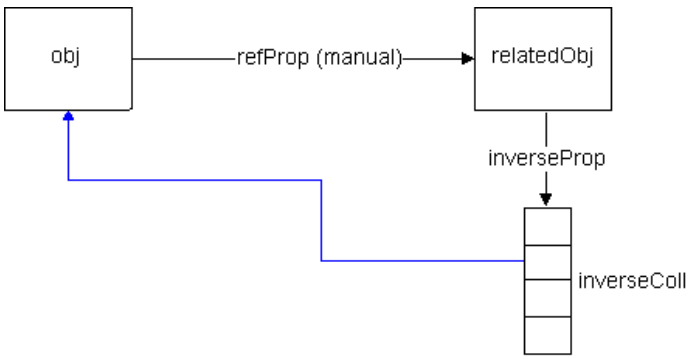


Case B



Error 26 – Collection Contains Object Not Meeting Constraint

An error **26** is detected when a collection contains a reference to an inverse object that does not meet the constraint.

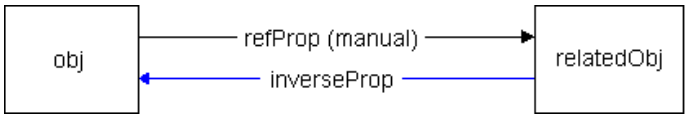


Repair:

```
remove inverseColl obj
```

Error 27 – Reference Set to Object that Does Not Satisfy Constraint

An error **27** is detected when a reference is set to an object that does not satisfy a constraint.



Repair:

```
null relatedObj inverseProp
```

Error 28 – Error in Slob or Blob

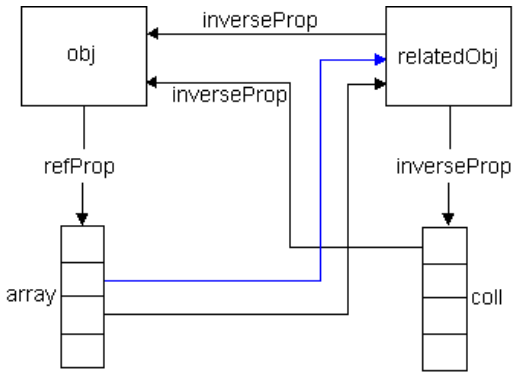
An error **28** is detected when a String Large Object (slob) or Binary Large Object (blob) contains an error.

Repair:

```
fixSlobOrBlob obj prop
```

Error 29 – Duplicate Entry in Array with Inverse Reference

An error **29** is detected when an array with inverse references contains a duplicate entry.



Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 30 – Invalid Inverse Definition for Shared Collection

An error **30** is detected when an illegal inverse is added to a shared collection.

Repair:

```
No repair provided.  It is your responsibility to repair this error yourself.
```

Error 31 – Invalid or Missing Collection Block

An error **31** is detected when a collection is found to have a missing or invalid collection block.

Repair:

```
rebuild coll
```

Error 32 – Invalid Instances in File

An error **32** is detected when instances of an abstract class are found, or when instances of a class are found in the wrong database map file.

Note This fix is commented out in the repair file that is generated. Instances of abstract classes should not exist. Instances of non-abstract classes should exist only in the database map file in which the class is defined.

Any class instances that the JADE Logical Certifier utility reports in the wrong file cannot be accessed directly. You should investigate the nature of these instances before they are deleted, and consider restoring these instances.

Repair:

```
deleteInstances fileName className classNumber
```

Error 33 – DynaDictionary Incomplete or Inconsistent

An error **33** is detected when a **DynaDictionary** references a missing or inconsistent class or property definition; for example, when the membership class of the **DynaDictionary** is deleted, or when a property that is used as a member key property of a **DynaDictionary** is deleted.

Repair:

```
delete oid
```

Error 40 – Orphan Dictionary Block

An error **40** is detected when a dictionary block is found but the parent instance that owns the dictionary does not exist.

Repair:

```
orphanBlock filename
```

Error 41 – Orphan Blob/Slob

An error **41** is detected when a blob or slob subobject is found but the parent instance that owns the Blob or Slob does not exist.

Repair:

```
orphanSlobOrBlob filename
```

Error 42 – Orphan Dynamic Property Cluster

An error **42** is detected when a dynamic property cluster is found but the parent instance that owns the cluster does not exist.

Repair:

```
orphanCluster filename
```

Error 43 – Orphan Subobject (Collection)

An error **43** is detected when a collection subobject is found but the parent instance that owns the collection does not exist.

Repair:

```
orphan oid
```

Chapter 6 C-Level Application Programming Interface (API)

JADE provides C-level API calls that help C programmers to write external methods and functions to obtain JADE initialization file information, directory information, and to convert JADE characters to an ANSI or Unicode string. This enables consistent behavior between the C-level code and the JADE runtime.

Function prototypes for these call, which are listed in the following table, are defined in the **joscalls.h** header in the **include** directory on the JADE release medium. The C functions return an **int** value, which will be zero (**0**) to indicate success or a JADE Object Manager error number.

API Call	For details, see…	Description
jomDecimalCompare	Comparing Decimal Values	Compares two **Decimal** values
jomDecimalFromInt64Scale	Converting an Integer64 to a Decimal	Converts an **Integer64** value with scale factor to a **Decimal** value
jomDecimalFromReal	Converting a Real to a Decimal	Converts a **Real** value to a **Decimal** value
jomDecimalFromString	Converting a String to a Decimal	Converts a **String** containing a fixed-point value to a **Decimal** value
jomDecimalToReal	Converting a Decimal to a Real	Converts a **Decimal** value to a **Real** value
jomDecimalToString	Converting a Decimal to a String	Converts a **Decimal** value to its **String** representation
josDskDateToGregorian	Converting a Date Value to a Gregorian Value	Converts a Julian **Date** value to Gregorian **Integer** values
josDskTimeStampCompare	Comparing TimeStamp Values	Compares two **TimeStamp** values
josDskTimeStampToGregorianHMSm	Converting a TimeStamp Value to a Gregorian Value	Converts a **TimeStamp** value to Gregorian date and HMS (hours, minutes, seconds) Integer values
josIniFileGetFileName	Getting the Name of the Initialization File	Obtains the name of the JADE initialization file
josIniFileGetBoolean	Getting a Boolean Value form the Initialization File	Obtains a **Boolean** value from the JADE initialization file
josIniFileGetSInteger	Getting a Signed Integer Value from the Initialization File	Obtains a signed **Integer** value from the JADE initialization file
josIniFileGetUInteger	Getting an Unsigned Integer Value from the Initialization File	Obtains an unsigned **Integer** value from the JADE initialization file
josIniFileGetString	Getting a String Value from the Initialization File	Obtains a **String** value from the JADE initialization file
josIniFileSetBoolean	Setting a Boolean Value form the Initialization File	Sets the **Boolean** value of a JADE initialization file key

API Call	For details, see…	Description
josIniFileSetSInteger	Setting a Signed Integer Value from the Initialization File	Sets the signed **Integer** value of a JADE initialization file key
josIniFileSetUInteger	Setting an Unsigned Integer Value from the Initialization File	Sets the unsigned **Integer** value of a JADE initialization file key
josIniFileSetString	Setting a String Value from the Initialization File	Sets the **String** value of a JADE initialization file key
josGetDirectoryJade	Getting the JADE HOME Directory	Obtains the name of the JADE HOME directory
josGetDirectoryJadeBin	Getting the JADE Installation Directory	Obtains the name of the JADE installation directory
josGetDirectoryJadeLib	Getting the JADE Lib Directory	Obtains the name of the directory in which JADE shared objects and libraries are installed
josGetDirectoryJadeTemp	Getting the JADE Temp Directory	Obtains the name of the JADE **temp** directory
josCharacterToAnsi	Converting a JADE Character to an ANSI Value	Converts a JADE **Character** type to an ANSI value
josCharacterToUnicode	Converting a JADE Character to a Unicode Value	Converts a JADE **Character** type to a Unicode value

For details, see the following subsections.

Getting the Name of the Initialization File

The **josIniFileGetFileName** call, shown in the following example, obtains the name of the JADE initialization file.

```
int josIniFileGetFileName(Character fileName[],
                          Size      size);
```

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

Parameter	Description
fileName	Name of the JADE initialization file
size	Size of the buffer to hold the JADE initialization file, defined in characters

Getting a Boolean Value from the Initialization File

The **josIniFileGetBoolean** call, shown in the following example, obtains the current value of a **Boolean** key in the specified section of the JADE initialization file.

```
int josIniFileGetBoolean(const Character * pIniFileName,
                         const Character * pSection,
                         const Character * pKey,
```

```
                                bool            bDefault,
                                bool &          bValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the **Boolean** key in the JADE initialization file section
bDefault	**Boolean** default value for the specified key if the key cannot be found in the initialization file
bValue	Requested **Boolean** value from the initialization file

Getting a Signed Integer Value from the Initialization File

The **josIniFileGetSInteger** call, shown in the following example, obtains the current value of a signed **Integer64** key in the specified section of the JADE initialization file.

```
int josIniFileGetSInteger(const Character * pIniFileName,
                          const Character * pSection,
                          const Character * pKey,
                          Integer64         sDefault,
                          Integer64 &       sValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the signed **Integer64** key in the JADE initialization file section
sDefault	Signed **Integer64** default value for the specified key if the key cannot be found in the initialization file
sValue	Requested signed **Integer64** value from the initialization file

Getting an Unsigned Integer Value from the Initialization File

The **josIniFileGetUInteger** call, shown in the following example, obtains the current value of an unsigned **Integer64** key in the specified section of the JADE initialization file.

```
int josIniFileGetUInteger(const Character * pIniFileName,
                          const Character * pSection,
                          const Character * pKey,
                          UInteger64        uDefault,
                          UInteger64 &      uValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the unsigned **Integer64** key in the JADE initialization file section
uDefault	Unsigned **Integer64** default value for the specified key if the key cannot be found in the initialization file
uValue	Requested unsigned **Integer64** value from the initialization file

Getting a String Value from the Initialization File

The **josIniFileGetString** call, shown in the following example, obtains the current value of the **String** key in the specified section of the JADE initialization file.

```
int josIniFileGetString(const Character * pIniFileName,
                        const Character * pSection,
                        const Character * pKey,
                        const Character * pDefault,
                        Character       * pValue,
                        Size              size);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the **String** key in the JADE initialization file section
pDefault	String default value for the specified **String** key if the key cannot be found in the initialization file
pValue	Address of a Character buffer into which the requested string value from the initialization file is placed
size	Size of the Character buffer to which the **pValue** parameter points, defined in characters

Setting a Boolean Value in the Initialization File

The **josIniFileSetBoolean** call, shown in the following example, sets the **Boolean** key value in the specified section of the JADE initialization file.

```
int josIniFileSetBoolean(const Character * pIniFileName,
                         const Character * pSection,
                         const Character * pKey,
                         bool              bValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the **Boolean** key in the JADE initialization file section
bValue	**Boolean** value that is written to the specified key and section of the JADE initialization file

Setting a Signed Integer Value in the Initialization File

The **josIniFileSetSInteger** call, shown in the following example, sets the **Integer64** key value in the specified section of the JADE initialization file.

```
int josIniFileSetSInteger(const Character * pIniFileName,
                          const Character * pSection,
                          const Character * pKey,
                          bool              bMultipliers,
                          Integer64         sValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file.
pSection	Name of the section in the JADE initialization file.
pKey	Name of the signed **Integer64** key in the JADE initialization file section.
bMultipliers	If **true**, the **sValue** parameter is checked to see if it is an exact multiplier, if possible, and the value is written with the appropriate multiplier suffix. Multipliers are case-insensitive **K**, **M**, or **G** *prefix multipliers*. (For details, see "Handling of Parameter Values", in the *JADE Initialization File Reference*.)
sValue	Signed **Integer64** value that is written to the specified key and section of the JADE initialization file.

Setting an Unsigned Integer Value in the Initialization File

The **josIniFileSetUInteger** call, shown in the following example, sets the unsigned **Integer64** key value in the specified section in the JADE initialization file.

```
int josIniFileSetUInteger(const Character * pIniFileName,
                          const Character * pSection,
                          const Character * pKey,
                          bool              bMultipliers,
                          UInteger64        uValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file.
pSection	Name of the section in the JADE initialization file.

Parameter	Description
pKey	Name of the unsigned **Integer64** key in the JADE initialization file section.
bMultipliers	If **true**, the **uValue** parameter is checked to see if it is an exact multiplier, if possible, and the value is written with the appropriate multiplier suffix. Multipliers are case-insensitive **K**, **M**, or **G** *prefix multipliers*. (For details, see "Handling of Parameter Values", in the *JADE Initialization File Reference*.)
uValue	Unsigned **Integer64** value that is written to the specified key and section of the JADE initialization file.

Setting a String Value in the Initialization File

The **josIniFileSetString** call, shown in the following example, sets the **String** key value in the specified section of the JADE initialization file.

```
int josIniFileSetString(const Character * pIniFileName,
                        const Character * pSection,
                        const Character * pKey,
                        const Character * pValue);
```

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

Parameter	Description
pIniFileName	Name of the JADE initialization file
pSection	Name of the section in the JADE initialization file
pKey	Name of the **String** key in the section in the JADE initialization file
pValue	**String** value that is written to the specified key and section of the JADE initialization file

Getting the JADE HOME Directory

The **josGetDirectoryJade** call, shown in the following example, obtains the name of the JADE HOME directory, which is the parent directory of the installation directory (for example, if your installation directory is **Jade\bin**, your JADE HOME directory is **Jade**).

```
int josGetDirectoryJade(Character pathName[],
                        Size      size);
```

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

Parameter	Description
pathName	Buffer to receive the name of the JADE HOME directory
size	Size of the **pathName** parameter buffer, defined in characters

Getting the JADE Installation Directory

The **josGetDirectoryJadeBin** call, shown in the following example, obtains the name of the JADE installation directory (commonly known as the **bin** directory).

```
int josGetDirectoryJadeBin(Character pathName[],
                           Size      size);
```

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

Parameter	Description
pathName	Buffer to receive the name of the directory where the executable of the current executing program is located; that is, the directory in which the JADE binaries are installed
size	Size of the **pathName** parameter buffer, defined in characters

Getting the JADE Lib Directory

The **josGetDirectoryJadeLib** call, shown in the following example, obtains the name of the directory in which JADE shared objects and libraries are installed (commonly known as the **bin** directory).

```
int josGetDirectoryJadeLib(Character pathName[],
                           Size      size);
```

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

Parameter	Description
pathName	Buffer to receive the name of the directory where the shared objects and libraries of the current executing program are located
size	Size of the **pathName** parameter buffer, defined in characters

Getting the JADE Temp Directory

The **josGetDirectoryJadeTemp** call, shown in the following example, obtains the name of the **temp** directory; for example, the directory to which files are extracted or installation files are downloaded. This directory is often **Jade\temp**.

```
int josGetDirectoryJadeTemp(Character pathName[],
                            Size      size);
```

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

Parameter	Description
pathName	Buffer to receive the name of the JADE **temp** directory
size	Size of the **pathName** parameter buffer, defined in characters

Comparing Decimal Values

The **jomDecimalCompare** call, shown in the following example, compares two **Decimal** values.

```
int jomDecimalCompare(const DskDecimal* pDskDecimal1,
                      const DskDecimal* pDskDecimal2);
```

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

Parameter	Description
pDskDecimal1	First **Decimal** value that is to be compared
pDskDecimal2	Second **Decimal** value that is to be compared

The **jomDecimalCompare** call returns the values listed in the following table.

Returned Value	Description
0	**Decimal1** is equal to **Decimal2**
-1	**Decimal1** is less than **Decimal2**
1	**Decimal1** is greater than **Decimal2**

Comparing TimeStamp Values

The **josDskTimeStampCompare** call, shown in the following example, compares two **TimeStamp** values.

```
int josDskTimeStampCompare(const DskTimeStamp & datetimeA,
                           const DskTimeStamp & datetimeB);
```

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

Parameter	Description
DskTimeStamp & datetimeA	First **TimeStamp** type and date and time values that are to be compared
DskTimeStamp & datetimeB	Second **TimeStamp** type and date and time values that are to be compared

The **josDskTimeStampCompare** call returns the values listed in the following table.

Returned Value	Description
0	**datetimeA** is equal to **datetimeB**
-1	**datetimeA** is less than **datetimeB**
1	**datetimeA** is greater than **datetimeB**

Converting a JADE Character Value to an ANSI Value

The **josCharacterToAnsi** call, shown in the following example, converts a JADE **Character** primitive type value to an ANSI value.

```
int josCharacterToAnsi(Character * pSource,
                       char        target[],
```

```
                                Size        sizeOfTargetInChar);
```

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

Parameter	Description
pSource	String of JADE characters that are to be converted
target	Location of the buffer to receive the converted string
sizeOfTargetInChar	Size of the target buffer, defined in ANSI characters

Converting a JADE Character Value to a Unicode Value

The **josCharacterToUnicode** call, shown in the following example, converts a JADE **Character** type to a Unicode value using the UTF-16 wide-character encoding routine.

```
int josCharacterToUnicode(Character * pSource,
                          wchar_t     target[],
                          Size        sizeOfTargetInWChar);
```

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

Parameter	Description
pSource	String of JADE characters that is to be converted
target	Location of the buffer to receive the converted string
sizeOfTargetInWChar	Size of the target buffer, defined in **wchar_t** characters

Converting a Date Value to a Gregorian Value

The **josDskDateToGregorian** call, shown in the following example, converts a **Date** primitive type value to Gregorian calendar format.

```
void josDskDateToGregorian(const DskDate,
                           DayType &day,
                           MonthType &month,
                           YearType &year);
```

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

Parameter	Description
DskDate	Julian date to be converted to the Gregorian calendar format
DayType &day	Day number for the Gregorian date
MonthType &month	Month number for the Gregorian date
YearType &year	Year number for the Gregorian date

Converting a Decimal to a Real

The **jomDecimalToReal** call, shown in the following example, converts a **Decimal** primitive type value to a **Real** primitive type value.

```
bool jomDecimalToReal(const DskDecimal* pDskDecimal,
                      Real* pResult);
```

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

Parameter	Description
pDskDecimal	**Decimal** primitive type that is to be converted to a **Real** primitive type
pResult	**Real** value resulting from the **Decimal** conversion

Converting a Decimal to a String

The **jomDecimalToString** call, shown in the following example, converts a **Decimal** primitive type value to its **String** representation rounded to the specified number of decimal places.

```
bool jomDecimalToString(const DskDecimal* pDskDecimal,
                        Character* pResult,
                        Size size,
                        unsigned decimalPlaces);
```

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

Parameter	Description
pDskDecimal	**Decimal** primitive type value that is to be converted to its **String** representation
pResult	Character buffer to hold the result from the **Decimal** conversion
size	Length of the string result buffer
decimalPlaces	Number of decimal places to include in the converted value

Converting an Integer64 to a Decimal

The **jomDecimalFromInt64Scale** call, shown in the following example, converts an **Integer64** primitive type value to a **Decimal** primitive type, after applying the scale factor.

```
bool jomDecimalFromInt64Scale(Int64 unscaled,
                              Int32 scale,
                              DskDecimal* pResult);
```

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

Parameter	Description
unscaled	Length of the **Decimal** primitive type value to be converted
scale	Scale factor; for example, **-2** means divide the **Integer64** value by 100
pResult	**Decimal** primitive type value resulting from the conversion

The **jomDecimalFromInt64Scale** call returns **true** if the conversion is successful.

Converting a Real to a Decimal

The **jomDecimalFromReal** call, shown in the following example, converts a **Real** primitive type value to a **Decimal** primitive type value.

```
bool jomDecimalFromReal(Real real,
                        DskDecimal* pResult);
```

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

Parameter	Description
real	**Real** primitive type value to be converted
pResult	**Decimal** value resulting from the **Real** conversion

The **jomDecimalFromReal** call returns **true** if the conversion is successful.

Converting a String to a Decimal

The **jomDecimalFromString** call, shown in the following example, converts a **String** representation of a floating-point value to a **Decimal** value.

```
bool jomDecimalFromString(const Character string[],
                          DskDecimal* pResult);
```

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

Parameter	Description
string	Buffer containing the text value
pResult	**Decimal** primitive type value resulting from the **String** conversion

Only US-ASCII characters in the range **0** through **9** are recognized.

The string can contain an optional prefix of tab and space characters, an optional sign, a series of digit values, or an optional decimal point followed by a series of digits. The first unrecognized character ends the conversion. If the string cannot be converted to a **Decimal** primitive type, **false** is returned.

Converting a TimeStamp Value to Gregorian Format

The **josDskTimeStampToGregorianHMSm** call, shown in the following example, converts a **TimeStamp** primitive type value to Gregorian day, month, year and hour, minute, second format.

```
void josDskTimeStampToGregorianHMSm(const DskTimeStamp* pDatetime,
                                    DayType &day,
                                    MonthType &month,
                                    YearType &year,
                                    int &dayOfWeek,
                                    int &hour,
                                    int &minute,
                                    int &second,
                                    int &milliSec);
```

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

Parameter	Description
pDatetime	TimeStamp value to be converted to the Gregorian calendar format
day	Day number for the equivalent Gregorian date
month	Month for the equivalent Gregorian date
year	Year for the equivalent Gregorian date
dayOfWeek	Day of the week for the converted date (**0** is Sunday)
hour	Hour of the day for the converted time
minute	Minute of the hour for the converted time
second	Second of the minute for the converted time
milliSec	Milliseconds of the second