6

Java Message Service for Transactional Event Queues and Advanced Queuing

This chapter contains the following topics:

- Java Messaging Service Interface for Oracle Transactional Event Queues and Advanced Queuing
- Oracle Java Message Service Basic Operations
- Oracle Java Message Service Point-to-Point
- Oracle Java Message Service Publish/Subscribe
- Oracle Java Message Service Shared Interfaces
- Oracle Java Message Service Types Examples

Java Messaging Service Interface for Oracle Transactional Event Queues and Advanced Queuing

The following topics describe the Oracle Java Message Service (JMS) interface to Oracle Database Advanced Queuing (AQ).

- General Features of JMS and Oracle JMS
- Structured Payload/Message Types in JMS
- Buffered Messaging in JMS
- JMS Point-to-Point Model Features
- JMS Publish/Subscribe Model Features
- JMS Message Producer Features
- JMS Message Consumer Features
- JMS Propagation
- Message Transformation with JMS AQ
- JMS Streaming
- Java EE Compliance

General Features of JMS and Oracle JMS

This section contains these topics:

- JMS Connection and Session
- JMS Destination
- System-Level Access Control in JMS
- Destination-Level Access Control in JMS

- Retention and Message History in JMS
- Supporting Oracle Real Application Clusters in JMS
- Supporting Statistics Views in JMS

JMS Connection and Session

This section contains these topics:

- ConnectionFactory Objects
- Using AQjmsFactory to Obtain ConnectionFactory Objects
- Using JNDI to Look Up ConnectionFactory Objects
- JMS Connection
- JMS Session

ConnectionFactory Objects

A `ConnectionFactory` encapsulates a set of connection configuration parameters that has been defined by an administrator. A client uses it to create a connection with a JMS provider. In this case Oracle JMS, part of Oracle Database, is the JMS provider.

The three types of `ConnectionFactory` objects are:

- `ConnectionFactory`
- `QueueConnectionFactory`
- `TopicConnectionFactory`

Using AQjmsFactory to Obtain ConnectionFactory Objects

You can use the `AQjmsFactory` class to obtain a handle to a `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` object.

To obtain a `ConnectionFactory`, which supports both point-to-point and publish/subscribe operations, use `AQjmsFactory.getConnectionFactory()`. To obtain a `QueueConnectionFactory`, use `AQjmsFactory.getQueueConnectionFactory()`. To obtain a `TopicConnectionFactory`, use `AQjmsFactory.getTopicConnectionFactory()`.

The `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` can be created using hostname, port number, and SID driver or by using JDBC URL and properties.

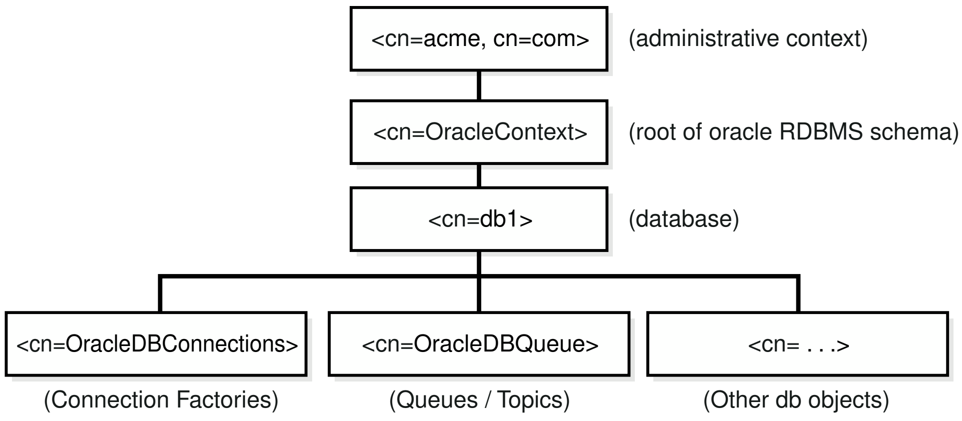
Using JNDI to Look Up ConnectionFactory Objects

A JMS administrator can register `ConnectionFactory` objects in a Lightweight Directory Access Protocol (LDAP) server. The following setup is required to enable Java Naming and Directory Interface (JNDI) lookup in JMS:

1. Register Database

   When the Oracle Database server is installed, the database must be registered with the LDAP server. This can be accomplished using the Database Configuration Assistant (DBCA). Figure 6-1 shows the structure of Oracle Database Advanced Queuing entries in the LDAP server. `ConnectionFactory` information is stored under `<cn=OracleDBConnections>`, while topics and queues are stored under `<cn=OracleDBQueues>`.

Figure 6-1 Structure of Oracle Database Advanced Queuing Entries in LDAP Server



2. Set Parameter `GLOBAL_TOPIC_ENABLED`.

   The `GLOBAL_TOPIC_ENABLED` system parameter for the database must be set to `TRUE`. This ensures that all queues and topics created in Oracle Database Advanced Queuing are automatically registered with the LDAP server. This parameter can be set by using `ALTER SYSTEM SET GLOBAL_TOPIC_ENABLED = TRUE`.

3. Register `ConnectionFactory` Objects

   After the database has been set up to use an LDAP server, the JMS administrator can register `ConnectionFactory`, `QueueConnectionFactory`, and `TopicConnectionFactory` objects in LDAP by using `AQjmsFactory.registerConnectionFactory()`.

   The registration can be accomplished in one of the following ways:

   - Connect directly to the LDAP server

     The user must have the `GLOBAL_AQ_USER_ROLE` to register connection factories in LDAP.

     To connect directly to LDAP, the parameters for the `registerConnectionFactory` method include the LDAP context, the name of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory`, hostname, database SID, port number, JDBC driver (thin or oci8) and factory type (queue or topic).

   - Connect to LDAP through the database server

     The user can log on to Oracle Database first and then have the database update the LDAP entry. The user that logs on to the database must have the `AQ_ADMINISTRATOR_ROLE` to perform this operation.

     To connect to LDAP through the database server, the parameters for the `registerConnectionFactory` method include a JDBC connection (to a user having `AQ_ADMINISTRATOR_ROLE`), the name of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory`, hostname, database SID, port number, JDBC driver (thin or oci8) and factory type (queue or topic).

JMS Connection

A JMS `Connection` is an active connection between a client and its JMS provider. A JMS `Connection` performs several critical services:

- Encapsulates either an open connection or a pool of connections with a JMS provider
- Typically represents an open TCP/IP socket (or a set of open sockets) between a client and a provider's service daemon
- Provides a structure for authenticating clients at the time of its creation
- Creates `Sessions`
- Provides connection metadata
- Supports an optional `ExceptionListener`

A JMS `Connection` to the database can be created by invoking `createConnection()`, `createQueueConnection()`, or `createTopicConnection()` and passing the parameters `username` and `password` on the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` object respectively.

Some of the methods that are supported on the `Connection` object are

- `start()`

  This method starts or restart delivery of incoming messages.
- `stop()`

  This method temporarily stops delivery of incoming messages. When a `Connection` object is stopped, delivery to all of its message consumers is inhibited. Also, synchronous receive's block and messages are not delivered to message listener.
- `close()`

  This method closes the JMS session and releases all associated resources.
- `createSession(true, 0)`

  This method creates a JMS `Session` using a JMS `Connection` instance.
- `createQueueSession(true, 0)`

  This method creates a `QueueSession`.
- `createTopicSession(true, 0)`

  This method creates a `TopicSession`.
- `setExceptionListener(ExceptionListener)`

  This method sets an exception listener for the `Connection`. This allows a client to be notified of a problem asynchronously. If a `Connection` only consumes messages, then it has no other way to learn it has failed.
- `getExceptionListener()`

  This method gets the `ExceptionListener` for this `Connection`.

A JMS client typically creates a `Connection`, a `Session` and several `MessageProducer` and `MessageConsumer` objects. In the current version only one open `Session` for each `Connection` is allowed, except in the following cases:

- If the JDBC oci8 driver is used to create the JMS connection
- If the user provides an `OracleOCIConnectionPool` instance during JMS connection creation

When a `Connection` is created it is in stopped mode. In this state no messages can be delivered to it. It is typical to leave the `Connection` in stopped mode until setup is complete. At that point the `Connection` `start()` method is called and messages begin arriving at the `Connection` consumers. This setup convention minimizes any client confusion that can result from asynchronous message delivery while the client is still in the process of setup.

It is possible to start a `Connection` and to perform setup subsequently. Clients that do this must be prepared to handle asynchronous message delivery while they are still in the process of setting up. A `MessageProducer` can send messages while a `Connection` is stopped.

JMS Session

A JMS `Session` is a single threaded context for producing and consuming messages. Although it can allocate provider resources outside the Java Virtual Machine (JVM), it is considered a lightweight JMS object.

A `Session` serves several purposes:

- Constitutes a factory for `MessageProducer` and `MessageConsumer` objects
- Provides a way to get a handle to destination objects (queues/topics)
- Supplies provider-optimized message factories
- Supports a single series of transactions that combines work spanning session `MessageProducer` and `MessageConsumer` objects, organizing these into units
- Defines a serial order for the messages it consumes and the messages it produces
- Serializes execution of `MessageListener` objects registered with it

In Oracle Database 20c, you can create as many JMS `Sessions` as resources allow using a single JMS `Connection`, when using either JDBC thin or JDBC thick (OCI) drivers.

Because a provider can allocate some resources on behalf of a `Session` outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough. The same is true for `MessageProducer` and `MessageConsumer` objects created by a `Session`.

Methods on the `Session` object include:

- `commit()`

  This method commits all messages performed in the transaction and releases locks currently held.

- `rollback()`

  This method rolls back any messages accomplished in the transaction and release locks currently held.

- `close()`

  This method closes the `Session`.

- `getDBConnection()`

  This method gets a handle to the underlying JDBC connection. This handle can be used to perform other SQL DML operations as part of the same `Session`. The method is specific to Oracle JMS.

- `acknowledge()`

  This method acknowledges message receipt in a nontransactional session.

- `recover()`

This method restarts message delivery in a nontransactional session. In effect, the series of delivered messages in the session is reset to the point after the last acknowledged message.

The following are some Oracle JMS extensions:

- `createQueueTable()`

  This method creates a queue table.
- `getQueueTable()`

  This method gets a handle to an existing queue table.
- `createQueue()`

  This method creates a queue.
- `getQueue()`

  This method gets a handle to an existing queue.
- `createTopic()`

  This method creates a topic.
- `getTopic()`

  This method gets a handle to an existing topic.

The `Session` object must be cast to `AQjmsSession` to use any of the extensions.

Note:

The JMS specification expects providers to return null messages when receives are accomplished on a JMS `Connection` instance that has not been started.

After you create a `javax.jms.Connection` instance, you must call the `start()` method on it before you can receive messages. If you add a line like `t_conn.start();` any time after the connection has been created, but before the actual receive, then you can receive your messages.

JMS Destination

A `Destination` is an object a client uses to specify the destination where it sends messages, and the source from which it receives messages. A `Destination` object can be a `Queue` or a `Topic`. In Oracle Database Advanced Queuing, these map to a *`schema.queue`* at a specific database. `Queue` maps to a single-consumer queue, and `Topic` maps to a multiconsumer queue.

Using a JMS Session to Obtain Destination Objects

`Destination` objects are created from a `Session` object using the following domain-specific `Session` methods:

- `AQjmsSession.getQueue(queue_owner, queue_name)`

  This method gets a handle to a JMS queue.
- `AQjmsSession.getTopic(topic_owner, topic_name)`

This method gets a handle to a JMS topic.

Using JNDI to Look Up Destination Objects

The database can be configured to register schema objects with an LDAP server. If a database has been configured to use LDAP and the GLOBAL_TOPIC_ENABLED parameter has been set to TRUE, then all JMS queues and topics are automatically registered with the LDAP server when they are created. The administrator can also create aliases to the queues and topics registered in LDAP. Queues and topics that are registered in LDAP can be looked up through JNDI using the name or alias of the queue or topic.

See Also:

"Adding an Alias to the LDAP Server"

JMS Destination Methods

Methods on the `Destination` object include:

- `alter()`

  This method alters a `Queue` or a `Topic`.
- `schedulePropagation()`

  This method schedules propagation from a source to a destination.
- `unschedulePropagation()`

  This method unschedules a previously scheduled propagation.
- `enablePropagationSchedule()`

  This method enables a propagation schedule.
- `disablePropagationSchedule()`

  This method disables a propagation schedule.
- `start()`

  This method starts a `Queue` or a `Topic`. The queue can be started for enqueue or dequeue. The topic can be started for publish or subscribe.
- `stop()`

  This method stops a `Queue` or a `Topic`. The queue is stopped for enqueue or dequeue. The topic is stopped for publish or subscribe.
- `drop()`

  This method drops a `Queue` or a `Topic`.

System-Level Access Control in JMS

Oracle8*i* or higher supports system-level access control for all queuing operations. This feature allows an application designer or DBA to create users as queue administrators. A queue administrator can invoke administrative and operational JMS interfaces on any queue in the database. This simplifies administrative work, because all administrative scripts for the queues in a database can be managed under one schema.

When messages arrive at the destination queues, sessions based on the source queue schema name are used for enqueuing the newly arrived messages into the destination queues. This means that you must grant enqueue privileges for the destination queues to schemas of the source queues.

To propagate to a remote destination queue, the login user (specified in the database link in the address field of the agent structure) should either be granted the `ENQUEUE_ANY` privilege, or be granted the rights to enqueue to the destination queue. However, you are not required to grant any explicit privileges if the login user in the database link also owns the queue tables at the destination.

See Also:

"Oracle Enterprise Manager Support"

Destination-Level Access Control in JMS

Oracle8*i* or higher supports access control for enqueue and dequeue operations at the queue or topic level. This feature allows the application designer to protect queues and topics created in one schema from applications running in other schemas. You can grant only minimal access privileges to the applications that run outside the schema of the queue or topic. The supported access privileges on a queue or topic are `ENQUEUE`, `DEQUEUE` and `ALL`.

See Also:

"Oracle Enterprise Manager Support"

Retention and Message History in JMS

Messages are often related to each other. For example, if a message is produced as a result of the consumption of another message, then the two are related. As the application designer, you may want to keep track of such relationships. Oracle Database Advanced Queuing allows users to retain messages in the queue table, which can then be queried in SQL for analysis.

Along with retention and message identifiers, Oracle Database Advanced Queuing lets you automatically create message journals, also called tracking journals or event journals. Taken together, retention, message identifiers and SQL queries make it possible to build powerful message warehouses.

Supporting Oracle Real Application Clusters in JMS

A transactional event queue (TxEventQ) is a single logical queue that is divided into multiple, independent, physical queues through system-maintained partitioning. TxEventQs are the preferred JMS queues for queues used across Oracle RAC instances, for queues with high enqueue or dequeue rates, or for queues with many subscribers. See "Transactional Event Queues and Oracle Real Application Clusters (Oracle RAC)" for more information.

For AQ queues, Oracle Real Application Clusters (Oracle RAC) can be used to improve Oracle Database Advanced Queuing performance by allowing different queues to be managed by different instances. You do this by specifying different instance affinities (preferences) for the queue tables that store the queues. This allows queue operations (enqueue/dequeue) or topic operations (publish/subscribe) on different queues or topics to occur in parallel.

The Oracle Database Advanced Queuing queue monitor process continuously monitors the instance affinities of the queue tables. The queue monitor assigns ownership of a queue table to the specified primary instance if it is available, failing which it assigns it to the specified secondary instance.

If the owner instance of a queue table terminates, then the queue monitor changes ownership to a suitable instance such as the secondary instance.

Oracle Database Advanced Queuing propagation can make use of Oracle Real Application Clusters, although it is transparent to the user. The affinities for jobs submitted on behalf of the propagation schedules are set to the same values as that of the affinities of the respective queue tables. Thus, a `job_queue_process` associated with the owner instance of a queue table is handling the propagation from queues stored in that queue table, thereby minimizing pinging.

See Also:

- "Transactional Event Queues"
- "Scheduling a Queue Propagation"
- *Oracle Real Application Clusters Administration and Deployment Guide*

Supporting Statistics Views in JMS

Each instance keeps its own Oracle Database Advanced Queuing statistics information in its own System Global Area (SGA), and does not have knowledge of the statistics gathered by other instances. Then, when a `GV$AQ` view is queried by an instance, all other instances funnel their statistics information to the instance issuing the query.

The `GV$AQ` view can be queried at any time to see the number of messages in waiting, ready or expired state. The view also displays the average number of seconds messages have been waiting to be processed.

See Also:

"V$AQ: Number of Messages in Different States in Database"

Structured Payload/Message Types in JMS

JMS messages are composed of a header, properties, and a body.

The header consists of header fields, which contain values used by both clients and providers to identify and route messages. All messages support the same set of header fields.

Properties are optional header fields. In addition to standard properties defined by JMS, there can be provider-specific and application-specific properties.

The body is the message payload. JMS defines various types of message payloads, and a type that can store JMS messages of any or all JMS-specified message types.

This section contains these topics:

JMS Message Headers

A JMS message header contains the following fields:

- `JMSDestination`

  This field contains the destination to which the message is sent. In Oracle Database Advanced Queuing this corresponds to the destination queue/topic. It is a `Destination` type set by JMS after the `Send` method has completed.

- `JMSDeliveryMode`

  This field determines whether the message is logged or not. JMS supports `PERSISTENT` delivery (where messages are logged to stable storage) and `NONPERSISTENT` delivery (messages not logged). It is a `INTEGER` set by JMS after the `Send` method has completed. JMS permits an administrator to configure JMS to override the client-specified value for `JMSDeliveryMode`.

- `JMSMessageID`

  This field uniquely identifies a message in a provider. All message IDs must begin with the string `ID:`. It is a `String` type set by JMS after the `Send` method has completed.

- `JMSTimeStamp`

  This field contains the time the message was handed over to the provider to be sent. This maps to Oracle Database Advanced Queuing message enqueue time. It is a `Long` type set by JMS after the `Send` method has completed.

- `JMSCorrelationID`

  This field can be used by a client to link one message with another. It is a `String` type set by the JMS client.

- `JMSReplyTo`

  This field contains a `Destination` type supplied by a client when a message is sent. Clients can use `oracle.jms.AQjmsAgent`; `javax.jms.Queue`; or `javax.jms.Topic`.

- `JMSType`

  This field contains a message type identifier supplied by a client at send time. It is a `String` type. For portability Oracle recommends that the `JMSType` be symbolic values.

- `JMSExpiration`

  This field is the sum of the enqueue time and the `TimeToLive` in non-Java EE compliance mode. In compliant mode, the `JMSExpiration` header value in a dequeued message is the sum of `JMSTimeStamp` when the message was enqueued (Greenwich Mean Time, in milliseconds) and the `TimeToLive` (in milliseconds). It is a `Long` type set by JMS after the `Send` method has completed. JMS permits an administrator to configure JMS to override the client-specified value for `JMSExpiration`.

- `JMSPriority`

  This field contains the priority of the message. It is a `INTEGER` set by JMS after the `Send` method has completed. In Java EE-compliance mode, the permitted values for priority are `0`–`9`, with `9` the highest priority and `4` the default, in conformance with the Sun Microsystem JMS 1.1 standard. Noncompliant mode is the default. JMS permits an administrator to configure JMS to override the client-specified value for `JMSPriority`.
- `JMSRedelivered`

  This field is a Boolean set by the JMS provider.

See Also:

"Java EE Compliance"

JMS Message Properties

JMS properties are set either explicitly by the client or automatically by the JMS provider (these are generally read-only). Some JMS properties are set using the parameters specified in `Send` and `Receive` operations.

Properties add optional header fields to a message. Properties allow a client, using a `messageSelector`, to have a JMS provider select messages on its behalf using application-specific criteria. Property names are strings and values can be: `Boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `string`.

JMS-defined properties, which all begin with `"JMSX"`, include the following:

- `JMSXUserID`

  This field is the identity of the user sending the message. It is a `String` type set by JMS after the `Send` method has completed.
- `JMSXAppID`

  This field is the identity of the application sending the message. It is a `String` type set by JMS after the `Send` method has completed.
- `JMSXDeliveryCount`

  This field is the number of message delivery attempts. It is an `Integer` set by JMS after the `Send` method has completed.
- `JMSXGroupid`

  This field is the identity of the message group that this message belongs to. It is a `String` type set by the JMS client.
- `JMSXGroupSeq`

  This field is the sequence number of a message within a group. It is an `Integer` set by the JMS client.
- `JMSXRcvTimeStamp`

  This field is the time the message was delivered to the consumer (dequeue time). It is a `String` type set by JMS after the `Receive` method has completed.
- `JMSXState`

This field is the message state, set by the provider. The message state can be `WAITING`, `READY`, `EXPIRED`, or `RETAINED`.

Oracle-specific JMS properties, which all begin with `JMS_Oracle`, include the following:

- `JMS_OracleExcpQ`

  This field is the queue name to send the message to if it cannot be delivered to the original destination. It is a `String` type set by the JMS client. Only destinations of type `EXCEPTION` can be specified in the `JMS_OracleExcpQ` property.

- `JMS_OracleDelay`

  This field is the time in seconds to delay the delivery of the message. It is an `Integer` set by the JMS client. This can affect the order of message delivery.

- `JMS_OracleOriginalMessageId`

  This field is set to the message identifier of the message in the source if the message is propagated from one destination to another. It is a `String` type set by the JMS provider. If the message is not propagated, then this property has the same value as `JMSMessageId`.

A client can add additional header fields to a message by defining properties. These properties can then be used in a `messageSelector` to select specific messages.

JMS Message Bodies

JMS provides five forms of message body:

- StreamMessage
- BytesMessage
- MapMessage
- TextMessage
- ObjectMessage
- AdtMessage

StreamMessage

A `StreamMessage` object is used to send a stream of Java primitives. It is filled and read sequentially. It inherits from `Message` and adds a `StreamMessage` body. Its methods are based largely on those found in `java.io.DataInputStream` and `java.io.DataOutputStream`.

The primitive types can be read or written explicitly using methods for each type. They can also be read or written generically as objects. To use `StreamMessage` objects, create the queue table with the `SYS.AQ$_JMS_STREAM_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

`StreamMessage` objects support the conversions shown in Table 6-1. A value written as the row type can be read as the column type.

Table 6-1 StreamMessage Conversion

Input	Boolean	byte	short	char	int	long	float	double	String	byte[]
`Boolean`	X	-	-	-	-	-	-	-	X	-
`byte`	-	X	X	-	X	X	-	-	X	-
`short`	-	-	X	-	X	X	-	-	X	-

Table 6-1 (Cont.) StreamMessage Conversion

Input	Boolean	byte	short	char	int	long	float	double	String	byte[]
`char`	-	-	-	X	-	-	-	-	X	-
`int`	-	-	-	-	X	X	-	-	X	-
`long`	-	-	-	-	-	X	-	-	X	-
`float`	-	-	-	-	-	-	X	X	X	-
`double`	-	-	-	-	-	-	-	X	X	-
`string`	X	X	X	X	X	X	X	X	X	-
`byte[]`	-	-	-	-	-	-	-	-	-	X

BytesMessage

A `BytesMessage` object is used to send a message containing a stream of uninterpreted bytes. It inherits `Message` and adds a `BytesMessage` body. The receiver of the message interprets the bytes. Its methods are based largely on those found in `java.io.DataInputStream` and `java.io.DataOutputStream`.

This message type is for client encoding of existing message formats. If possible, one of the other self-defining message types should be used instead.

The primitive types can be written explicitly using methods for each type. They can also be written generically as objects. To use `BytesMessage` objects, create the queue table with `SYS.AQ$_JMS_BYTES_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

MapMessage

A `MapMessage` object is used to send a set of name-value pairs where the names are `String` types, and the values are Java primitive types. The entries can be accessed sequentially or randomly by name. The order of the entries is undefined. It inherits from `Message` and adds a `MapMessage` body. The primitive types can be read or written explicitly using methods for each type. They can also be read or written generically as objects.

To use `MapMessage` objects, create the queue table with the `SYS.AQ$_JMS_MAP_MESSAGE` or `AQ$_JMS_MESSAGE` payload types. `MapMessage` objects support the conversions shown in Table 6-2. An "X" in the table means that a value written as the row type can be read as the column type.

Table 6-2 MapMessage Conversion

Input	Boolean	byte	short	char	int	long	float	double	String	byte[]
`Boolean`	X	-	-	-	-	-	-	-	X	-
`byte`	-	X	X	-	X	X	-	-	X	-
`short`	-	-	X	-	X	X	-	-	X	-
`char`	-	-	-	X	-	-	-	-	X	-
`int`	-	-	-	-	X	X	-	-	X	-
`long`	-	-	-	-	-	X	-	-	X	-
`float`	-	-	-	-	-	-	X	X	X	-

Table 6-2 (Cont.) MapMessage Conversion

Input	Boolean	byte	short	char	int	long	float	double	String	byte[]
`double`	-	-	-	-	-	-	-	X	X	-
`string`	X	X	X	X	X	X	X	X	X	-
`byte[]`	-	-	-	-	-	-	-	-	-	X

TextMessage

A `TextMessage` object is used to send a message containing a `java.lang.StringBuffer`. It inherits from `Message` and adds a `TextMessage` body. The text information can be read or written using methods `getText()` and `setText(...)`. To use `TextMessage` objects, create the queue table with the `SYS.AQ$_JMS_TEXT_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

ObjectMessage

An `ObjectMessage` object is used to send a message that contains a serializable Java object. It inherits from Message and adds a body containing a single Java reference. Only serializable Java objects can be used. If a collection of Java objects must be sent, then one of the collection classes provided in JDK 1.4 can be used. The objects can be read or written using the methods `getObject()` and `setObject(...)`.To use `ObjectMessage` objects, create the queue table with the `SYS.AQ$_JMS_OBJECT_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

AdtMessage

An `AdtMessage` object is used to send a message that contains a Java object that maps to an Oracle object type. These objects inherit from `Message` and add a body containing a Java object that implements the `CustomDatum` or `ORAData` interface.

To use `AdtMessage` objects, create the queue table with payload type as the Oracle object type. The `AdtMessage` payload can be read and written using the `getAdtPayload` and `setAdtPayload` methods.

You can also use an `AdtMessage` object to send messages to queues of type `SYS.XMLType`. You must use the `oracle.xdb.XMLType` class to create the message.

For `AdtMessage` objects, the client can get:

- `JMSXDeliveryCount`
- `JMSXRecvTimeStamp`
- `JMSXState`
- `JMS_OracleExcpQ`
- `JMS_OracleDelay`

See Also:

Oracle Database Java Developer's Guide for information about the `CustomDatum` and `ORAData` interfaces

Using Message Properties with Different Message Types

The following message properties can be set by the client using the `setProperty` call. For `StreamMessage`, `BytesMessage`, `ObjectMessage`, `TextMessage`, and `MapMessage` objects, the client can set:

- `JMSXAppID`
- `JMSXGroupID`
- `JMSXGroupSeq`
- `JMS_OracleExcpQ`
- `JMS_OracleDelay`

For `AdtMessage` objects, the client can set:

- `JMS_OracleExcpQ`
- `JMS_OracleDelay`

The following message properties can be obtained by the client using the `getProperty` call. For `StreamMessage`, `BytesMessage`, `ObjectMessage`, `TextMessage`, and `MapMessage` objects, the client can get:

- `JMSXuserID`
- `JMSXAppID`
- `JMSXDeliveryCount`
- `JMSXGroupID`
- `JMSXGroupSeq`
- `JMSXRecvTimeStamp`
- `JMSXState`
- `JMS_OracleExcpQ`
- `JMS_OracleDelay`
- `JMS_OracleOriginalMessageID`

Buffered Messaging with Oracle JMS

Users can send a nonpersistent JMS message by specifying the `deliveryMode` to be `NON_PERSISTENT` when sending a message. JMS nonpersistent messages are not required to be logged to stable storage, so they can be lost after a JMS system failure. JMS nonpersistent messages are similar to the buffered messages available in Oracle Database Advanced Queuing, but there are also important differences between the two.

Note:

Do not confuse Oracle JMS nonpersistent messages with Oracle Database Advanced Queuing nonpersistent queues, which are deprecated in Oracle Database 10*g* Release 2 (10.2).

See Also:

- "Buffered Messaging"
- Nonpersistent Queues

Transaction Commits and Client Acknowledgments

The JMS `deliveryMode` is orthogonal to the transaction attribute of a message. JMS nonpersistent messages can be sent and received by either a transacted session or a nontransacted session. If a JMS nonpersistent message is sent and received by a transacted session, then the effect of the JMS operation is only visible after the transacted session commits. If it is received by a nontransacted session with `CLIENT_ACKNOWLEDGE` acknowledgment mode, then the effect of receiving this message is only visible after the client acknowledges the message. Without the acknowledgment, the message is not removed and will be redelivered if the client calls `Session.recover`.

Oracle Database Advanced Queuing buffered messages, however, do not support these transaction or acknowledgment concepts. Both sending and receiving a buffered message must be in the `IMMEDIATE` visibility mode. The effects of the sending and receiving operations are therefore visible to the user immediately, no matter whether the session is committed or the messages are acknowledged.

Different APIs

Messages sent with the regular JMS send and publish methods are treated by Oracle Database Advanced Queuing as persistent messages. The regular JMS receive methods receive only AQ persistent messages. To send and receive buffered messages, you must use the Oracle extension APIs `bufferSend`, `bufferPublish`, and `bufferReceive`.

See Also:

Oracle Database Advanced Queuing Java API Reference for more information on `bufferSend`, `bufferPublish`, and `bufferReceive`

Payload Limits

The Oracle Database Advanced Queuing implementation of buffered messages does not support `LOB` attributes. This places limits on the payloads for the five types of standard JMS messages:

- JMS `TextMessage` payloads cannot exceed 4000 bytes.

  This limit might be even lower with some database character sets, because during the Oracle JMS character set conversion, Oracle JMS sometimes must make a conservative choice of using `CLOB` instead of `VARCHAR` to store the text payload in the database.
- JMS `BytesMessage` payloads cannot exceed 2000 bytes.
- JMS `ObjectMessage`, `StreamMessage`, and `MapMessage` data serialized by JAVA cannot exceed 2000 bytes.
- For all other Oracle JMS ADT messages, the corresponding Oracle database ADT cannot contain `LOB` attributes.

Different Constants

The Oracle Database Advanced Queuing and Oracle JMS APIs use different numerical values to designate buffered and persistent messages, as shown in Table 6-3.

Table 6-3 Oracle Database AQ and Oracle JMS Buffered Messaging Constants

API	Persistent Message	Buffered Message
Oracle Database Advanced Queuing	`PERSISTENT := 1`	`BUFFERED :=2`
Oracle JMS	`PERSISTENT := 2`	`NON_PERSISTENT := 1`

Buffered Messaging in JMS

Buffered messaging fully supports JMS messaging standards. Oracle JMS extends those standards in several ways.

See Also:

"Buffered Messaging"

Enqueuing JMS Buffered Messages

Oracle JMS allows applications to send buffered messages by setting `JMSDeliveryMode` for individual messages, so persistent and buffered messages can be enqueued to the same JMS queue/topic.

Oracle JMS buffered messages can be ordered by enqueue time, priority, or both. The ordering does not extend across message types. So a persistent message sent later, for example, can be delivered before an buffered message sent earlier. Expiration is also supported for buffered messages in Oracle JMS.

See Also:

"JMS Message Headers"

Dequeuing JMS Buffered Messages

JMS does not require subscribers to declare interest in just persistent messages or just buffered messages, so JMS subscribers can be interested in both message types.

Oracle JMS supports fast and efficient dequeue of messages by `JMSMessageID`, selectors on message headers, and selectors on message properties. The Oracle JMS dequeue call checks for both persistent and buffered messages.

> **Note:**
>
> Oracle JMS persistent messages have unique message identifiers. Oracle JMS buffered message identifiers are unique only within a queue/topic.

If concurrent dequeue processes are dequeuing from the same queue as the same subscriber, then they will skip messages that are locked by the other process.

> **See Also:**
>
> - "MessageSelector"
> - "Receiving Messages "

Transactions Support

If buffered messages are enqueued in a transacted session, then JMS requires transaction support for them. Oracle JMS guarantees that transacted sessions involving buffered messages meet the following standards:

- Atomicity

  Both persistent and buffered messages within an Oracle JMS transaction are committed or rolled back atomically. Even if buffered messages were written to disk, as in the case of messages involving LOBs, rollback nevertheless removes them.
- Consistency

  If persistent and buffered messaging operations interleave in a transaction, then all Oracle JMS users share a consistent view of the affected queues/topics. All persistent and buffered messages enqueued by a transaction become visible at commit time. If a process ends in the middle of a transaction, then both persistent and buffered messages are undone. Oracle JMS users see either all persistent and buffered messages in a transaction or none of them.
- Isolation

  An buffered enqueue operation in a transaction is visible only to the owner transaction before the transaction is committed. It is visible to all consumers after the transaction is committed.

Messages locked by dequeue transaction may be browsed.

Acknowledging Message Receipt

Three values are defined for the `ack_mode` parameter for acknowledging message receipt in nontransacted sessions:

- `DUPS_OK_ACKNOWLEDGE`

  In this mode, duplicate messages are allowed.
- `AUTO_ACKNOWLEDGE`

  In this mode, the session automatically acknowledges messages.
- `CLIENT_ACKNOWLEDGE`

In this mode, the client explicitly acknowledges messages by calling the message producer acknowledge method. Acknowledging a message acknowledges all previously consumed messages.

See Also:

"Creating a Session"

Buffered Messaging Quality of Service

JMS requires providers to support at-most-once delivery of unpropagated buffered messages. If recovery of buffered messages is disabled, then Oracle JMS meets this standard.

Duplicate delivery of messages is possible with the current implementation of message propagation. But this does not violate the JMS standard, because message propagation is an extension offered by Oracle JMS.

See Also:

"Propagating Buffered Messages" for the causes of duplicate delivery of buffered messages

JMS Types Support for Buffered Messages

Oracle JMS maps the JMS-defined types to Oracle user-defined types and creates queues of these user-defined types for storing JMS messages. Some of these types have LOB attributes, which Oracle JMS writes to disk whether the message is persistent or buffered.

The user-defined type `SYS.AQ$_JMS_TEXT_MESSAGE` for JMS type `JMSTextMessage`, for example, stores text strings smaller than 4k in a `VARCHAR2` column. But it has a CLOB attribute for storing text strings larger than 4k.

Because JMS messages are often larger than 4k, Oracle JMS offers a new ADT that allows larger messages to be stored in memory. The disk representation of the ADT remains unchanged, but several `VARCHAR2`/`RAW` attributes allow for JMS messages of sizes up to 100k to be stored in memory. Messages larger than 100k can still be published as buffered messages, but they are written to disk.

See Also:

"Enqueuing Buffered Messages"

JMS Point-to-Point Model Features

In the point-to-point model, clients exchange messages from one point to another. Message producers and consumers send and receive messages using single-consumer queues. An administrator creates the single-consumer queues with the `createQueue` method in `AQjmsSession`. Before they can be used, the queues must be enabled for enqueue/dequeue using the `start` call in `AQjmsDestination`. Clients obtain a handle to a previously created queue using the `getQueue` method on `AQjmsSession`.

In a single-consumer queue, a message can be consumed exactly once by a single consumer. If there are multiple processes or operating system threads concurrently dequeuing from the same queue, then each process dequeues the first unlocked message at the head of the queue. A locked message cannot be dequeued by a process other than the one that has created the lock.

After processing, the message is removed if the retention time of the queue is 0, or it is retained for a specified retention time. As long as the message is retained, it can be either queried using SQL on the queue table view or dequeued by specifying the message identifier of the processed message in a `QueueBrowser`.

QueueSender

A client uses a `QueueSender` to send messages to a queue. It is created by passing a queue to the `createSender` method in a client `Session`. A client also has the option of creating a `QueueSender` without supplying a queue. In that case a queue must be specified on every send operation.

A client can specify a default delivery mode, priority and `TimeToLive` for all messages sent by the `QueueSender`. Alternatively, the client can define these options for each message.

QueueReceiver

A client uses a `QueueReceiver` to receive messages from a queue. It is created using the `createQueueReceiver` method in a client `Session`. It can be created with or without a `messageSelector`.

QueueBrowser

A client uses a `QueueBrowser` to view messages on a queue without removing them. The browser method returns a `java.util.Enumeration` that is used to scan messages in the queue. The first call to `nextElement` gets a snapshot of the queue. A `QueueBrowser` can be created with or without a `messageSelector`.

A `QueueBrowser` can also optionally lock messages as it is scanning them. This is similar to a "`SELECT`... `for UPDATE`" command on the message. This prevents other consumers from removing the message while they are being scanned.

MessageSelector

A `messageSelector` allows the client to restrict messages delivered to the consumer to those that match the `messageSelector` expression. A `messageSelector` for queues containing payloads of type `TextMessage`, `StreamMessage`, `BytesMessage`, `ObjectMessage`, or `MapMessage` can contain any expression that has one or more of the following:

- JMS message identifier prefixed with "ID:"

  ```
  JMSMessageID ='ID:23452345'
  ```

- JMS message header fields or properties

  ```
  JMSPriority < 3 AND JMSCorrelationID = 'Fiction'
  ```

  ```
  JMSCorrelationID LIKE 'RE%'
  ```

- User-defined message properties

  ```
  color IN ('RED', BLUE', 'GREEN') AND price < 30000
  ```

The `messageSelector` for queues containing payloads of type `AdtMessage` can contain any expression that has one or more of the following:

- Message identifier without the "ID:" prefix

  ```
  msgid = '23434556566767676'
  ```

- Priority, correlation identifier, or both

  ```
  priority < 3 AND corrid = 'Fiction'
  ```

- Message payload

  ```
  tab.user_data.color = 'GREEN' AND tab.user_data.price < 30000
  ```

JMS Publish/Subscribe Model Features

This section contains these topics:

- JMS Publish/Subscribe Overview
- DurableSubscriber
- RemoteSubscriber
- TopicPublisher
- Recipient Lists
- TopicReceiver
- TopicBrowser
- Setting Up JMS Publish/Subscribe Operations

JMS Publish/Subscribe Overview

JMS enables flexible and dynamic communication between applications functioning as publishers and applications playing the role of subscribers. The applications are not coupled together; they interact based on messages and message content.

In distributing messages, publisher applications are not required to handle or manage message recipients explicitly. This allows new subscriber applications to be added dynamically without changing any publisher application logic.

Similarly, subscriber applications receive messages based on message content without regard to which publisher applications are sending messages. This allows new publisher applications to be added dynamically without changing any subscriber application logic.

Subscriber applications specify interest by defining a rule-based subscription on message properties or the message content of a topic. The system automatically routes messages by computing recipients for published messages using the rule-based subscriptions.

In the publish/subscribe model, messages are published to and received from topics. A topic is created using the `CreateTopic()` method in an `AQjmsSession`. A client can obtain a handle to a previously-created topic using the `getTopic()` method in `AQjmsSession`.

DurableSubscriber

A client creates a `DurableSubscriber` with the `createDurableSubscriber()` method in a client `Session`. It can be created with or without a `messageSelector`.

A `messageSelector` allows the client to restrict messages delivered to the subscriber to those that match the selector. The syntax for the selector is described in detail in `createDurableSubscriber` in *Oracle Database Advanced Queuing Java API Reference*.

When subscribers use the same name, durable subscriber action depends on the Java EE compliance mode set for an Oracle Java Message Service (Oracle JMS) client at runtime.

In noncompliant mode, two durable `TopicSubscriber` objects with the same name can be active against two different topics. In compliant mode, durable subscribers with the same name are not allowed. If two subscribers use the same name and are created against the same topic, but the selector used for each subscriber is different, then the underlying Oracle Database Advanced Queuing subscription is altered using the internal `DBMS_AQJMS.ALTER_SUBSCRIBER()` call.

If two subscribers use the same name and are created against two different topics, and if the client that uses the same subscription name also originally created the subscription name, then the existing subscription is dropped and the new subscription is created.

If two subscribers use the same name and are created against two different topics, and if a different client (a client that did not originate the subscription name) uses an existing subscription name, then the subscription is not dropped and an error is thrown. Because it is not known if the subscription was created by JMS or PL/SQL, the subscription on the other topic should not be dropped.

See Also:

- "MessageSelector"
- "Java EE Compliance"

RemoteSubscriber

Remote subscribers are defined using the `createRemoteSubscriber` call. The remote subscriber can be a specific consumer at the remote topic or all subscribers at the remote topic

A remote subscriber is defined using the `AQjmsAgent` structure. An `AQjmsAgent` consists of a name and address. The name refers to the `consumer_name` at the remote topic. The address refers to the remote topic:

```
schema.topic_name[@dblink]
```

To publish messages to a particular consumer at the remote topic, the `subscription_name` of the recipient at the remote topic must be specified in the name field of `AQjmsAgent`. The remote topic must be specified in the address field of `AQjmsAgent`.

To publish messages to all subscribers of the remote topic, the name field of `AQjmsAgent` must be set to null. The remote topic must be specified in the address field of `AQjmsAgent`.

TopicPublisher

Messages are published using `TopicPublisher`, which is created by passing a `Topic` to a `createPublisher` method. A client also has the option of creating a `TopicPublisher` without supplying a `Topic`. In this case, a `Topic` must be specified on every publish operation. A client can specify a default delivery mode, priority and `TimeToLive` for all messages sent by the `TopicPublisher`. It can also specify these options for each message.

Recipient Lists

In the JMS publish/subscribe model, clients can specify explicit recipient lists instead of having messages sent to all the subscribers of the topic. These recipients may or may not be existing subscribers of the topic. The recipient list overrides the subscription list on the topic for this message. Recipient lists functionality is an Oracle extension to JMS.

TopicReceiver

If the recipient name is explicitly specified in the recipient list, but that recipient is not a subscriber to the queue, then messages sent to it can be received by creating a `TopicReceiver`. If the subscriber name is not specified, then clients must use durable subscribers at the remote site to receive messages. `TopicReceiver` is an Oracle extension to JMS.

A `TopicReceiver` can be created with a `messageSelector`. This allows the client to restrict messages delivered to the recipient to those that match the selector.

See Also:

"MessageSelector"

TopicBrowser

A client uses a `TopicBrowser` to view messages on a topic without removing them. The browser method returns a `java.util.Enumeration` that is used to scan topic messages. Only durable subscribers are allowed to create a `TopicBrowser`. The first call to `nextElement` gets a snapshot of the topic.

A `TopicBrowser` can optionally lock messages as it is scanning them. This is similar to a `SELECT`... `for UPDATE` command on the message. This prevents other consumers from removing the message while it is being scanned.

A `TopicBrowser` can be created with a `messageSelector`. This allows the client to restrict messages delivered to the browser to those that match the selector.

`TopicBrowser` supports a purge feature. This allows a client using a `TopicBrowser` to discard all messages that have been seen during the current browse operation on the topic. A purge is equivalent to a destructive receive of all of the seen messages (as if performed using a `TopicSubscriber`).

For a purge, a message is considered seen if it has been returned to the client using a call to the `nextElement()` operation on the `java.lang.Enumeration` for the `TopicBrowser`. Messages that have not yet been seen by the client are not discarded during a purge. A purge operation can be performed multiple times on the same `TopicBrowser`.

The effect of a purge becomes stable when the JMS `Session` used to create the `TopicBrowser` is committed. If the operations on the session are rolled back, then the effects of the purge operation are also undone.

See Also:

- "Creating a TopicBrowser for Standard JMS Messages"
- "Creating a TopicBrowser for Standard JMS Messages_ Locking Messages"
- "MessageSelector"
- "Browsing Messages Using a TopicBrowser"

Setting Up JMS Publish/Subscribe Operations

Follow these steps to use the publish/subscribe model of communication in JMS:

1. Set up one or more topics to hold messages. These topics represent an area or subject of interest. For example, a topic can represent billed orders.
2. Enable enqueue/dequeue on the topic using the `start` call in `AQjmsDestination`.
3. Create a set of durable subscribers. Each subscriber can specify a `messageSelector` that selects the messages that the subscriber wishes to receive. A null `messageSelector` indicates that the subscriber wishes to receive all messages published on the topic.

   Subscribers can be local or remote. Local subscribers are durable subscribers defined on the same topic on which the message is published. Remote subscribers are other topics, or recipients on other topics that are defined as subscribers to a particular queue. In order to use remote subscribers, you must set up propagation between the source and destination topics. Remote subscribers and propagation are Oracle extensions to JMS.

   See Also:

   "Managing Propagations"

4. Create `TopicPublisher` objects using the `createPublisher()` method in the publisher `Session`. Messages are published using the `publish` call. Messages can be published to all subscribers to the topic or to a specified subset of recipients on the topic.
5. Subscribers receive messages on the topic by using the `receive` method.
6. Subscribers can also receive messages asynchronously by using message listeners.

   See Also:

   "Listening to One or More Queues"

JMS Message Producer Features

- Priority and Ordering of Messages
- Specifying a Message Delay
- Specifying a Message Expiration
- Message Grouping

Priority and Ordering of Messages

Message ordering dictates the order in which messages are received from a queue or topic. The ordering method is specified when the queue table for the queue or topic is created. Currently, Oracle Database Advanced Queuing supports ordering on message priority and enqueue time, producing four possible ways of ordering:

- First-In, First-Out (FIFO)

  If enqueue time was chosen as the ordering criteria, then messages are received in the order of the enqueue time. The enqueue time is assigned to the message by Oracle Database Advanced Queuing at message publish/send time. This is also the default ordering.
- Priority Ordering

  If priority ordering was chosen, then each message is assigned a priority. Priority can be specified as a message property at publish/send time by the `MessageProducer`. The messages are received in the order of the priorities assigned.
- FIFO Priority

  If FIFO priority ordering was chosen, then the topic/queue acts like a priority queue. If two messages are assigned the same priority, then they are received in the order of their enqueue time.
- Enqueue Time Followed by Priority

  Messages with the same enqueue time are received according to their priorities. If the ordering criteria of two message is the same, then the order they are received is indeterminate. However, Oracle Database Advanced Queuing does ensure that messages produced in one session with a particular ordering criteria are received in the order they were sent.

All ordering schemes available for persistent messages are also available for buffered messages, but only within each message class. Ordering among persistent and buffered messages enqueued/published in the same session is not currently supported.

Specifying a Message Delay

Messages can be sent/published to a queue/topic with delay. The delay represents a time interval after which the message becomes available to the message consumer. A message specified with a delay is in a waiting state until the delay expires. Receiving by message identifier overrides the delay specification.

Delay is an Oracle Database Advanced Queuing extension to JMS message properties. It requires the Oracle Database Advanced Queuing background process queue monitor to be started.

Specifying a Message Expiration

Producers of messages can specify expiration limits, or `TimeToLive` for messages. This defines the period of time the message is available for a Message Consumer.

`TimeToLive` can be specified at send/publish time or using the set `TimeToLive` method of a `MessageProducer`, with the former overriding the latter. The Oracle Database Advanced Queuing background process queue monitor must be running to implement `TimeToLive`.

Message Grouping

Messages belonging to a queue/topic can be grouped to form a set that can be consumed by only one consumer at a time. This requires the queue/topic be created in a queue table that is enabled for transactional message grouping. All messages belonging to a group must be created in the same transaction, and all messages created in one transaction belong to the same group.

Message grouping is an Oracle Database Advanced Queuing extension to the JMS specification.

You can use this feature to divide a complex message into a linked series of simple messages. For example, an invoice directed to an invoices queue could be divided into a header message, followed by several messages representing details, followed by the trailer message.

Message grouping is also very useful if the message payload contains complex large objects such as images and video that can be segmented into smaller objects.

The priority, delay, and expiration properties for the messages in a group are determined solely by the message properties specified for the first message (head) of the group. Properties specified for subsequent messages in the group are ignored.

Message grouping is preserved during propagation. The destination topic must be enabled for transactional grouping.

See Also:

"Dequeue Features" for a discussion of restrictions you must keep in mind if message grouping is to be preserved while dequeuing messages from a queue enabled for transactional grouping

JMS Message Consumer Features

This section contains these topics:

- Receiving Messages
- Message Navigation in Receive
- Browsing Messages
- Remove No Data
- Retry with Delay Interval
- Asynchronously Receiving Messages Using MessageListener
- Exception Queues

Receiving Messages

A JMS application can receive messages by creating a message consumer. Messages can be received synchronously using the `receive` call or asynchronously using a message listener.

There are three modes of receive:

- Block until a message arrives for a consumer

- Block for a maximum of the specified time
- Nonblocking

Message Navigation in Receive

If a consumer does not specify a navigation mode, then its first `receive` in a session retrieves the first message in the queue or topic, its second `receive` gets the next message, and so on. If a high priority message arrives for the consumer, then the consumer does not receive the message until it has cleared the messages that were already there before it.

To provide the consumer better control in navigating the queue for its messages, Oracle Database Advanced Queuing offers several navigation modes as JMS extensions. These modes can be set at the `TopicSubscriber`, `QueueReceiver` or the `TopicReceiver`.

Two modes are available for ungrouped messages:

- `FIRST_MESSAGE`

  This mode resets the position to the beginning of the queue. It is useful for priority ordered queues, because it allows the consumer to remove the message on the top of the queue.
- `NEXT_MESSAGE`

  This mode gets whatever message follows the established position of the consumer. For example, a `NEXT_MESSAGE` applied when the position is at the fourth message will get the fifth message in the queue. This is the default action.

Three modes are available for grouped messages:

- `FIRST_MESSAGE`

  This mode resets the position to the beginning of the queue.
- `NEXT_MESSAGE`

  This mode sets the position to the next message in the same transaction.
- `NEXT_TRANSACTION`

  This mode sets the position to the first message in the next transaction.

> **Note:**
>
> Transactional event queues do not support the three preceding modes.

The transaction grouping property can be negated if messages are received in the following ways:

- Receive by specifying a correlation identifier in the selector
- Receive by specifying a message identifier in the selector
- Committing before all the messages of a transaction group have been received

If the consumer reaches the end of the queue while using the `NEXT_MESSAGE` or `NEXT_TRANSACTION` option, and you have specified a blocking `receive()`, then the navigating position is automatically changed to the beginning of the queue.

By default, a `QueueReceiver`, `TopicReceiver`, or `TopicSubscriber` uses `FIRST_MESSAGE` for the first receive call, and `NEXT_MESSAGE` for subsequent `receive()` calls.

Browsing Messages

Aside from the usual `receive`, which allows the dequeuing client to delete the message from the queue, JMS provides an interface that allows the JMS client to browse its messages in the queue. A `QueueBrowser` can be created using the `createBrowser` method from `QueueSession`.

If a message is browsed, then it remains available for further processing. That does not necessarily mean that the message will remain available to the JMS session after it is browsed, because a `receive` call from a concurrent session might remove it.

To prevent a viewed message from being removed by a concurrent JMS client, you can view the message in the locked mode. To do this, you must create a `QueueBrowser` with the locked mode using the Oracle Database Advanced Queuing extension to the JMS interface. The lock on the message is released when the session performs a commit or a rollback.

To remove a message viewed by a `QueueBrowser`, the session must create a `QueueReceiver` and use the `JMSmesssageID` as the selector.

Remove No Data

The consumer can remove a message from a queue or topic without retrieving it using the `receiveNoData` call. This is useful when the application has already examined the message, perhaps using a `QueueBrowser`. This mode allows the JMS client to avoid the overhead of retrieving a payload from the database, which can be substantial for a large message.

Retry with Delay Interval

If a transaction receiving a message from a queue/topic fails, then it is regarded as an unsuccessful attempt to remove the message. Oracle Database Advanced Queuing records the number of failed attempts to remove the message in the message history.

An application can specify the maximum number of retries supported on messages at the queue/topic level. If the number of failed attempts to remove a message exceeds this maximum, then the message is moved to an exception queue.

Oracle Database Advanced Queuing allows users to specify a `retry_delay` along with `max_retries`. This means that a message that has undergone a failed attempt at retrieving remains visible in the queue for dequeue after `retry_delay` interval. Until then it is in the `WAITING` state. The Oracle Database Advanced Queuing background process time manager enforces the retry delay property.

The maximum retries and retry delay are properties of the queue/topic. They can be set when the queue/topic is created or by using the alter method on the queue/topic. The default value for `MAX_RETRIES` is `5`.

Note:

Transactional event queues do not support retry delay.

Asynchronously Receiving Messages Using MessageListener

The JMS client can receive messages asynchronously by setting the `MessageListener` using the `setMessageListener` method.

When a message arrives for the consumer, the `onMessage` method of the message listener is invoked with the message. The message listener can commit or terminate the receipt of the message. The message listener does not receive messages if the JMS `Connection` has been stopped. The `receive` call must not be used to receive messages once the message listener has been set for the consumer.

The JMS client can receive messages asynchronously for all consumers in the session by setting the `MessageListener` at the session. No other mode for receiving messages must be used in the session once the message listener has been set.

Exception Queues

An exception queue is a repository for all expired or unserviceable messages. Applications cannot directly enqueue into exception queues. However, an application that intends to handle these expired or unserviceable messages can receive/remove them from the exception queue.

To retrieve messages from exception queues, the JMS client must use the point-to-point interface. The exception queue for messages intended for a topic must be created in a queue table with multiple consumers enabled. Like any other queue, the exception queue must be enabled for receiving messages using the `start` method in the `AQOracleQueue` class. You get an exception if you try to enable it for enqueue.

Transactional event queues (TxEventQ) support exception queues through the `DBMS_AQADM.CREATE_EQ_EXCEPTION_QUEUE` API.

```
PROCEDURE CREATE_EQ_EXCEPTION_QUEUE(
  queue_name     IN VARCHAR2,
  exception_queue_name   IN VARCHAR2 DEFAULT NULL,
  multiple_consumers     IN BOOLEAN DEFAULT FALSE,
  storage_clause         IN VARCHAR2 DEFAULT NULL,
  sort_list              IN VARCHAR DEFAULT NULL,
  comment                IN VARCHAR2 DEFAULT NULL
  );
```

The exception queue is an Oracle-specific message property called "`JMS_OracleExcpQ`" that can be set with the message before sending/publishing it. If an exception queue is not specified, then the default exception queue is used. For AQ queues, the default exception queue is automatically created when the queue table is created and is named `AQ$_`*`queue_table_name`*`_E`. By default, no exception queue is created for TxEventQs.

Messages are moved to the exception queue under the following conditions:

- The message was not dequeued within the specified `timeToLive`.

  For messages intended for more than one subscriber, the message is moved to the exception queue if one or more of the intended recipients is not able to dequeue the message within the specified `timeToLive`.

- The message was received successfully, but the application terminated the transaction that performed the `receive` because of an error while processing the message. The message is returned to the queue/topic and is available for any applications that are waiting to receive messages.

A `receive` is considered rolled back or undone if the application terminates the entire transaction, or if it rolls back to a savepoint that was taken before the `receive`.

Because this was a failed attempt to receive the message, its retry count is updated. If the retry count of the message exceeds the maximum value specified for the queue/topic where it resides, then it is moved to the exception queue.

If a message has multiple subscribers, then the message is moved to the exception queue only when all the recipients of the message have exceeded the retry limit.

Note:

If a dequeue transaction failed because the server process died (including `ALTER SYSTEM KILL SESSION`) or `SHUTDOWN ABORT` on the instance, then `RETRY_COUNT` is not incremented.

JMS Propagation

This section contains these topics:

- RemoteSubscriber
- Scheduling Propagation
- Enhanced Propagation Scheduling Capabilities
- Exception Handling During Propagation

Note:

TxEventQ queues do not support RemoteSubscriber, Scheduling Propagation, Enhanced Propagation Scheduling Capabilities, and Exception Handling During Propagation.

RemoteSubscriber

Oracle Database Advanced Queuing allows a subscriber at another database to subscribe to a topic. If a message published to the topic meets the criterion of the remote subscriber, then it is automatically propagated to the queue/topic at the remote database specified for the remote subscriber. Propagation is performed using database links and Oracle Net Services. This enables applications to communicate with each other without having to be connected to the same database.

There are two ways to implement remote subscribers:

- The `createRemoteSubscriber` method can be used to create a remote subscriber to/on the topic. The remote subscriber is specified as an instance of the class `AQjmsAgent`.
- The `AQjmsAgent` has a name and an address. The address consists of a queue/topic and the database link to the database of the subscriber.

There are two kinds of remote subscribers:

- The remote subscriber is a topic.

  This occurs when no name is specified for the remote subscriber in the `AQjmsAgent` object and the address is a topic. The message satisfying the subscriber's subscription is propagated to the remote topic. The propagated message is now available to all the subscriptions of the remote topic that it satisfies.
- A specific remote recipient is specified for the message.

  The remote subscription can be for a particular consumer at the remote database. If the name of the remote recipient is specified (in the `AQjmsAgent` object), then the message satisfying the subscription is propagated to the remote database for that recipient only. The recipient at the remote database uses the `TopicReceiver` interface to retrieve its messages. The remote subscription can also be for a point-to-point queue.

Scheduling Propagation

Propagation must be scheduled using the `schedule_propagation` method for every topic from which messages are propagated to target destination databases.

A schedule indicates the time frame during which messages can be propagated from the source topic. This time frame can depend on several factors such as network traffic, the load at the source database, the load at the destination database, and so on. The schedule therefore must be tailored for the specific source and destination. When a schedule is created, a job is automatically submitted to the `job_queue` facility to handle propagation.

The administrative calls for propagation scheduling provide great flexibility for managing the schedules. The duration or propagation window parameter of a schedule specifies the time frame during which propagation must take place. If the duration is unspecified, then the time frame is an infinite single window. If a window must be repeated periodically, then a finite duration is specified along with a `next_time` function that defines the periodic interval between successive windows.

The propagation schedules defined for a queue can be changed or dropped at any time during the life of the queue. In addition there are calls for temporarily disabling a schedule (instead of dropping the schedule) and enabling a disabled schedule. A schedule is active when messages are being propagated in that schedule. All the administrative calls can be made irrespective of whether the schedule is active or not. If a schedule is active, then it takes a few seconds for the calls to be executed.

Job queue processes must be started for propagation to take place. At least 2 job queue processes must be started. The database links to the destination database must also be valid. The source and destination topics of the propagation must be of the same message type. The remote topic must be enabled for enqueue. The user of the database link must also have enqueue privileges to the remote topic.

See Also:

"Scheduling a Propagation"

Enhanced Propagation Scheduling Capabilities

Catalog views defined for propagation provide the following information about active schedules:

- Name of the background process handling the schedule

- SID (session and serial number) for the session handling the propagation
- Instance handling a schedule (if using Oracle RAC)
- Previous successful execution of a schedule
- Next planned execution of a schedule

The following propagation statistics are maintained for each schedule, providing useful information to queue administrators for tuning:

- The total number of messages propagated in a schedule
- Total number of bytes propagated in a schedule
- Maximum number of messages propagated in a window
- Maximum number of bytes propagated in a window
- Average number of messages propagated in a window
- Average size of propagated messages
- Average time to propagated a message

Propagation has built-in support for handling failures and reporting errors. For example, if the database link specified is invalid, or if the remote database is unavailable, or if the remote topic/queue is not enabled for enqueuing, then the appropriate error message is reported. Propagation uses an exponential backoff scheme for retrying propagation from a schedule that encountered a failure. If a schedule continuously encounters failures, then the first retry happens after 30 seconds, the second after 60 seconds, the third after 120 seconds and so forth. If the retry time is beyond the expiration time of the current window, then the next retry is attempted at the start time of the next window. A maximum of 16 retry attempts are made after which the schedule is automatically disabled.

Note:

Once a retry attempt slips to the next propagation window, it will always do so; the exponential backoff scheme no longer governs retry scheduling. If the date function specified in the `next_time` parameter of `DBMS_AQADM.SCHEDULE_PROPAGATION()` results in a short interval between windows, then the number of unsuccessful retry attempts can quickly reach 16, disabling the schedule.

When a schedule is disabled automatically due to failures, the relevant information is written into the alert log. It is possible to check at any time if there were failures encountered by a schedule and if so how many successive failures were encountered, the error message indicating the cause for the failure and the time at which the last failure was encountered. By examining this information, an administrator can fix the failure and enable the schedule.

If propagation is successful during a retry, then the number of failures is reset to 0.

Propagation has built-in support for Oracle Real Application Clusters and is transparent to the user and the administrator. The job that handles propagation is submitted to the same instance as the owner of the queue table where the source topic resides. If at any time there is a failure at an instance and the queue table that stores the topic is migrated to a different instance, then the propagation job is also automatically migrated to the new instance. This minimizes the pinging between instances and thus offers better performance. Propagation has been designed to handle any number of concurrent schedules.

The number of `job_queue_processes` is limited to a maximum of 1000 and some of these can be used to handle jobs unrelated to propagation. Hence, propagation has built in support for multitasking and load balancing. The propagation algorithms are designed such that multiple schedules can be handled by a single snapshot (`job_queue`) process. The propagation load on a `job_queue` processes can be skewed based on the arrival rate of messages in the different source topics. If one process is overburdened with several active schedules while another is less loaded with many passive schedules, then propagation automatically redistributes the schedules among the processes such that they are loaded uniformly.

Exception Handling During Propagation

When a system error such as a network failure occurs, Oracle Database Advanced Queuing continues to attempt to propagate messages using an exponential back-off algorithm. In some situations that indicate application errors in queue-to-dblink propagations, Oracle Database Advanced Queuing marks messages as `UNDELIVERABLE` and logs a message in `alert.log`. Examples of such errors are when the remote queue does not exist or when there is a type mismatch between the source queue and the remote queue. The trace files in the `background_dump_dest` directory can provide additional information about the error.

When a new job queue process starts, it clears the mismatched type errors so the types can be reverified. If you have capped the number of job queue processes and propagation remains busy, then you might not want to wait for the job queue process to terminate and restart. Queue types can be reverified at any time using `DBMS_AQADM.VERIFY_QUEUE_TYPES`.

Note:

When a type mismatch is detected in queue-to-queue propagation, propagation stops and throws an error. In such situations you must query the `DBA_SCHEDULES` view to determine the last error that occurred during propagation to a particular destination. The message is not marked as `UNDELIVERABLE`.

Message Transformation with JMS AQ

A transformation can be defined to map messages of one format to another. Transformations are useful when applications that use different formats to represent the same information must be integrated. Transformations can be SQL expressions and PL/SQL functions. Message transformation is an Oracle Database Advanced Queuing extension to the standard JMS interface.

The transformations can be created using the `DBMS_TRANSFORM.create_transformation` procedure. Transformation can be specified for the following operations:

- Sending a message to a queue or topic
- Receiving a message from a queue or topic
- Creating a `TopicSubscriber`
- Creating a `RemoteSubscriber`. This enables propagation of messages between topics of different formats.

Note:

TxEventQ does not support message transformation.

JMS Streaming

AQ JMS supports streaming with enqueue and dequeue for TxEventQ through `AQjmsBytesMessage` and `AQjmsStreamMessage` for applications to send and receive large message data or payload.

JMS streaming reduces the memory requirement when dealing with large messages, by dividing the message payload into small chunks rather than sending or receiving a large contiguous array of bytes. As JMS standard does not have any streaming mechanism, AQ JMS will provide proprietary interfaces to expose AQ streaming enqueue and dequeue features. This allows users to easily use an existing java input output stream to send and receive message data or payload.

In order to allow the existing applications to work without any changes on upgrading database to RDBMS 12.2, the streaming APIs will be disabled by default.

The client application can enable JMS Streaming by using the system property `oracle.jms.useJmsStreaming` set to `true`.

Note:

JMS Streaming is supported only for thin drivers.

JMS Streaming with Enqueue

AQ JMS provides the new API `setInputStream(java.io.InputStream)` in `AQjmsBytesMessage` and `AQjmsStreamMessage`, to set an input stream for message data.

```
/**
 * @param inputStream - InputStream to read the message payload
 * @throws JMSException - if the JMS provided fails to read the payload due to
 *                        some internal error
 */
public void setInputStream(InputStream inputStream) throws JMSException
```

The following code snippet creates a message of type `AQjmsBytesMessage` and sets a `FileInputStream` for the message data.

```
Session session = connection.createSession(false, Session.CLIENT_ACKNOWLEDGE);
Destination destination = session.createQueue("queueName");
MessageProducer producer = session.createProducer(destination);

 AQjmsBytesMessage bytesMessage = (AQjmsBytesMessage)session.createBytesMessage();
 InputStream input = new FileInputStream("somefile.data");
 bytesMessage.setInputStream(input);
 producer.send(bytesMessage);
```

Note:

- The methods in `BytesMessage` and `StreamMessage` are based on the methods found in `java.io.DataInputStream` and `java.io.DataOutputStream`, and hence, meaningful conversion of various `read*()` and `write*()` methods is not possible with streaming. The following scenarios will result in an exception:
  - `bytesMessage.setInputStream(input);`

    `bytesMessage.writeInt(99);`
  - `bytesMessage.writeInt(99);`

    `bytesMessage.setInputStream(input);`
- As with normal enqueue operation, the enqueue with streaming is going to be a synchronous one and we will return the control to the client only after the enqueue is complete.
- Streaming will be used with enqueue only when these APIs are explicitly used by the client. AQ JMS will not use streaming with enqueue with the normal enqueue, irrespective of the size of the message data.

JMS Streaming with Dequeue

The dequeue operation with streaming is achieved in two steps. The server decides whether to stream the message body or not based on the size of the message body. The default threshold limit is 10 MB. So when the message body is greater than 10MB and streaming is enabled by the client using the system property oracle.jms.useJmsStreaming, server will use streaming with dequeue.

- This is the normal dequeue process where a client calls the `receive()` method.

```
Destination destination = session.createQueue ("queueName");
AQjmsConsumer consumer = (AQjmsConsumer)
session.createConsumer(destination);
Message message = consumer.receive(10000);
```

- When the client receives the message without the payload, client can figure out whether the streaming is used for dequeue by calling `isLargeBody()` on the received message.

```
/**
 * This method can be used by the client applications to check whether the message
 * contains large messaege body and hence requires streaming with dequeue.
 *
 * @return true when the message body is large and server decides to stream
 *         the payload with dequeue
 */
public boolean isLargeBody()
```

  A value of true returned by `isLargeBody()` indicates streaming with dequeue. When the dequeue uses streaming, AQ JMS will populate the length of the message body properly for `AQjmsStreamMessage` along with `AQjmsBytesMessage`. So the client application can call the `getBodyLength()` on the message to determine the size of the payload.

```
 public long getBodyLength()
```

Once client has the understanding about the streaming with dequeue, the message data can be fetched by using one of the following APIs on the received message.

The client application can use on the following APIs available in `AQjmsBytesMessage` and `AQjmsStreamMessage` to receive the message data.

```
 /**
     * Writes the message body to the OutputStream specified.
     *
     * @param outputStream - the OutputStream to which message body can be written
     * @return the OutputStream containing the message body.
     * @throws JMSException - if the JMS provided fails to receive the message body
     *                         due to some internal error
     */
    public OutputStream getBody(OutputStream outputStream) throws JMSException

    /**
     * Writes the message body to the OutputStream specified, with chunkSize bytes
     * written at a time.
     *
     * @param outputStream - the OutputStream to which message body can be written
     * @param chunkSize - the number of bytes to be written at a time, default value
     *                    8192 (ie. 8KB)
     * @return the OutputStream containing the message body.
     * @throws JMSException - if the JMS provided fails to receive the message body
     *                         due to some internal error
     */
    public OutputStream getBody(OutputStream outputStream, int chunkSize)throws 
JMSException


    /**
     * Writes the message body to the OutputStream specified. This method waits until
     * the message body is written completely to the OutputStream or the timeout expires.
     *
     * A timeout of zero never expires, and a timeout of negative value is ignored.
     *
     * @param outputStream - the OutputStream to which message body can be written
     * @param timeout - the timeout value (in milliseconds)
     * @return the OutputStream containing the message body.
     * @throws JMSException - if the JMS provided fails to receive the message body
     *                         due to some internal error
     */
    public OutputStream getBody(OutputStream outputStream, long timeout) throws 
JMSException


    /**
     * Writes the message body to the OutputStream specified, chunkSize bytes at a time.
     * This method waits until the message body is written completely to the OutputStream
     * or the timeout expires.
     *
     * A timeout of zero never expires, and a timeout of negative value is ignored.
     *
     * @param outputStream - the OutputStream to which message body can be written
     * @param chunkSize - the number of bytes to be written at a time,
     *                    default value 8192 (ie. 8KB)
     * @param timeout - the timeout value (in milliseconds)
     * @return the OutputStream containing the message body.
     * @throws JMSException - if the JMS provided fails to receive the message body
     *                         due to some internal error
     */
    public OutputStream getBody(OutputStream outputStream, int chunkSize, long timeout) 
throws JMSException
```

The following code snippet checks whether streaming is used with dequeue and the payload received will be written to a `FileOutputStream`.

```
if (message instanceof BytesMessage && (AQjmsBytesMessage)message.isLargeBody()){
    // optional : check the size of the payload and take appropriate action before
    // receiving the payload.
     (AQjmsBytesMessage) message.getBody(new FileOutputStream(new File("…")));
} else {
     // normal dequeue
}
```

In general, when both the steps are complete, the message is considered as consumed completely. The AQ server keeps a lock on the message after Step 1 which will be released only after Step 2.

Considering the possible issues with partially consumed messages by the message consumers, we have restricted the Streaming APIs for the session with acknowledgement modes `CLIENT_ACKNOWLEDGE` and `SESSION_TRANSACTED`.

So all the messages including partially consumed messages are considered fully consumed when:

- `message.acknowledge()` is called with `CLIENT_ACKNOWLEDGE` session.
- Session's `commit()` is called in a transacted session.

As in normal case, session `rollback()`, rolls back the messages received in that session.

The JMS Streaming is available with the following restrictions:

- Streaming is disabled by default, and can be enabled by the client application using the system property `oracle.jms.useJmsStreaming`
- Dequeue uses streaming when the size of the message data is more than the threshold value. The default threshold value is 10 MB.
- Streaming support is available with `AQjmsBytesMessage` and `AQjmsStreamMessage`
- Streaming support is available only for TxEventQ queues
- Streaming support is available only with thin drivers
- Streaming support is not available when the message producer uses the message delivery mode as `NON_PERSISTENT`
- Streaming is not supported with message listener. So when a MessageConsumer has a message listener set and if the message data crosses threshold limit, internally we will use the normal dequeue.
- Streaming support is available with Sessions using acknowledgement modes `CLIENT_ACKNOWLEDGE` and `SESSION_TRANSACTED`.

Java EE Compliance

Oracle JMS conforms to the Oracle Sun Microsystems JMS 1.1 standard. You can define the Java EE compliance mode for an Oracle Java Message Service (Oracle JMS) client at runtime. For compliance, set the Java property `oracle.jms.j2eeCompliant` to `TRUE` as a command line option. For noncompliance, do nothing. `FALSE` is the default value.

Features in Oracle Database Advanced Queuing that support Java EE compliance (and are also available in the noncompliant mode) include:

- Nontransactional sessions

- Durable subscribers
- Temporary queues and topics
- Nonpersistent delivery mode
- Multiple JMS messages types on a single JMS queue or topic (using Oracle Database Advanced Queuing queues of the `AQ$_JMS_MESSAGE` type)
- The `noLocal` option for durable subscribers
- TxEventQ has native JMS support and conform to Java EE compliance

See Also:

- *Java Message Service Specification*, version 1.1, March 18, 2002, Sun Microsystems, Inc.
- "JMS Message Headers" for information on how the Java property `oracle.jms.j2eeCompliant` affects JMSPriority and JMSExpiration
- "DurableSubscriber" for information on how the Java property `oracle.jms.j2eeCompliant` affects durable subscribers

Oracle Java Message Service Basic Operations

The following topics describe the basic operational Java Message Service (JMS) administrative interface to Oracle Database Advanced Queuing (AQ).

- EXECUTE Privilege on DBMS_AQIN
- Registering a ConnectionFactory
- Unregistering a Queue/Topic ConnectionFactory
- Getting a QueueConnectionFactory or TopicConnectionFactory
- Getting a Queue or Topic in LDAP
- Creating an AQ Queue Table
- Creating a Queue
- Getting an AQ Queue Table
- Granting and Revoking Privileges
- Managing Destinations
- Propagation Schedules

EXECUTE Privilege on DBMS_AQIN

Users should never directly call methods in the `DBMS_AQIN` package, but they do need the `EXECUTE` privilege on `DBMS_AQIN`. Use the following syntax to accomplish this:

```
GRANT EXECUTE ON DBMS_AQIN to user;
```

Registering a ConnectionFactory

You can register a ConnectionFactory four ways:

- Registering Through the Database Using JDBC Connection Parameters
- Registering Through the Database Using a JDBC URL
- Registering Through LDAP Using JDBC Connection Parameters
- Registering Through LDAP Using a JDBC URL

Registering Through the Database Using JDBC Connection Parameters

```
public static int registerConnectionFactory(java.sql.Connection connection,
                                            java.lang.String conn_name,
                                            java.lang.String hostname,
                                            java.lang.String oracle_sid,
                                            int portno,
                                            java.lang.String driver,
                                            java.lang.String type)
                                     throws JMSException
```

This method registers a `QueueConnectionFactory` or TopicConnectionFactory through the database to a Lightweight Directory Access Protocol (LDAP) server with JDBC connection parameters. This method is static and has the following parameters:

Parameter	Description
`connection`	JDBC connection used in registration
`conn_name`	Name of the connection to be registered
`hostname`	Name of the host running Oracle Database Advanced Queuing
`oracle_sid`	Oracle system identifier
`portno`	Port number
`driver`	JDBC driver type
`type`	Connection factory type (`QUEUE` or `TOPIC`)

The database connection passed to `registerConnectionFactory` must be granted `AQ_ADMINISTRATOR_ROLE`. After registration, you can look up the connection factory using Java Naming and Directory Interface (JNDI).

Example 6-1 Registering Through the Database Using JDBC Connection Parameters

```
String              url;
java.sql.connection  db_conn;

url = "jdbc:oracle:thin:@sun-123:1521:db1";
db_conn = DriverManager.getConnection(url, "scott", "tiger");
AQjmsFactory.registerConnectionFactory(
   db_conn, "queue_conn1", "sun-123", "db1", 1521, "thin", "queue");
```

Registering Through the Database Using a JDBC URL

```
public static int registerConnectionFactory(java.sql.Connection connection,
                                            java.lang.String conn_name,
                                            java.lang.String jdbc_url,
                                            java.util.Properties info,
                                            java.lang.String type)
                                     throws JMSException
```

This method registers a `QueueConnectionFactory` or TopicConnectionFactory through the database with a JDBC URL to LDAP. It is static and has the following parameters:

Parameter	Description
`connection`	JDBC connection used in registration
`conn_name`	Name of the connection to be registered
`jdbc_url`	URL to connect to
`info`	Properties information
`portno`	Port number
`type`	Connection factory type (`QUEUE` or `TOPIC`)

The database connection passed to `registerConnectionFactory` must be granted `AQ_ADMINISTRATOR_ROLE`. After registration, you can look up the connection factory using JNDI.

Example 6-2   Registering Through the Database Using a JDBC URL

```
String                        url;
java.sql.connection           db_conn;

url = "jdbc:oracle:thin:@sun-123:1521:db1";
db_conn = DriverManager.getConnection(url, "scott", "tiger");
AQjmsFactory.registerConnectionFactory(
   db_conn, "topic_conn1", url, null, "topic");
```

Registering Through LDAP Using JDBC Connection Parameters

```
public static int registerConnectionFactory(java.util.Hashtable env,
                                            java.lang.String conn_name,
                                            java.lang.String hostname,
                                            java.lang.String oracle_sid,
                                            int portno,
                                            java.lang.String driver,
                                            java.lang.String type)
                                     throws JMSException
```

This method registers a `QueueConnectionFactory` or TopicConnectionFactory through LDAP with JDBC connection parameters to LDAP. It is static and has the following parameters:

Parameter	Description
`env`	Environment of LDAP connection
`conn_name`	Name of the connection to be registered
`hostname`	Name of the host running Oracle Database Advanced Queuing
`oracle_sid`	Oracle system identifier
`portno`	Port number
`driver`	JDBC driver type
`type`	Connection factory type (`QUEUE` or `TOPIC`)

The hash table passed to `registerConnectionFactory()` must contain all the information to establish a valid connection to the LDAP server. Furthermore, the connection must have write access to the connection factory entries in the LDAP server (which requires the LDAP user to be either the database itself or be granted `GLOBAL_AQ_USER_ROLE`). After registration, look up the connection factory using JNDI.

Example 6-3   Registering Through LDAP Using JDBC Connection Parameters

```
Hashtable            env = new Hashtable(5, 0.75f);
/* the following statements set in hashtable env:
   * service provider package
   * the URL of the ldap server
   * the distinguished name of the database server
   * the authentication method (simple)
   * the LDAP username
   * the LDAP user password
*/
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://sun-456:389");
env.put("searchbase", "cn=db1,cn=Oraclecontext,cn=acme,cn=com");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=db1aqadmin,cn=acme,cn=com");
env.put(Context.SECURITY_CREDENTIALS, "welcome");

AQjmsFactory.registerConnectionFactory(env,
                                       "queue_conn1",
                                       "sun-123",
                                       "db1",
                                       1521,
                                       "thin",
                                       "queue");
```

Registering Through LDAP Using a JDBC URL

```
public static int registerConnectionFactory(java.util.Hashtable env,
                                            java.lang.String conn_name,
                                            java.lang.String jdbc_url,
                                            java.util.Properties info,
                                            java.lang.String type)
                                     throws JMSException
```

This method registers a `QueueConnectionFactory` or TopicConnectionFactory through LDAP with JDBC connection parameters to LDAP. It is static and has the following parameters:

Parameter	Description
`env`	Environment of LDAP connection
`conn_name`	Name of the connection to be registered
`jdbc_url`	URL to connect to
info	Properties information
`type`	Connection factory type (`QUEUE` or `TOPIC`)

The hash table passed to `registerConnectionFactory()` must contain all the information to establish a valid connection to the LDAP server. Furthermore, the connection must have write access to the connection factory entries in the LDAP server (which requires the LDAP user to be either the database itself or be granted `GLOBAL_AQ_USER_ROLE)`. After registration, look up the connection factory using JNDI.

Example 6-4   Registering Through LDAP Using a JDBC URL

```
String              url;
Hashtable           env = new Hashtable(5, 0.75f);

/* the following statements set in hashtable env:
   * service provider package
   * the URL of the ldap server
   * the distinguished name of the database server
   * the authentication method (simple)
   * the LDAP username
   * the LDAP user password
*/
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://sun-456:389");
env.put("searchbase", "cn=db1,cn=Oraclecontext,cn=acme,cn=com");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=db1aqadmin,cn=acme,cn=com");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
url = "jdbc:oracle:thin:@sun-123:1521:db1";
AQjmsFactory.registerConnectionFactory(env, "topic_conn1", url, null, "topic");
```

Unregistering a Queue/Topic ConnectionFactory

You can unregister a queue/topic `ConnectionFactory` in LDAP two ways:

- Unregistering Through the Database
- Unregistering Through LDAP

Unregistering Through the Database

```
public static int unregisterConnectionFactory(java.sql.Connection connection,
                                              java.lang.String conn_name)
                                       throws JMSException
```

This method unregisters a `QueueConnectionFactory` or `TopicConnectionFactory` in LDAP. It is static and has the following parameters:

Parameter	Description
`connection`	JDBC connection used in registration
`conn_name`	Name of the connection to be registered

The database connection passed to `unregisterConnectionFactory()` must be granted `AQ_ADMINISTRATOR_ROLE`.

Example 6-5   Unregistering Through the Database

```
String              url;
java.sql.connection db_conn;

url = "jdbc:oracle:thin:@sun-123:1521:db1";
db_conn = DriverManager.getConnection(url, "scott", "tiger");
AQjmsFactory.unregisterConnectionFactory(db_conn, "topic_conn1");
```

Unregistering Through LDAP

```
public static int unregisterConnectionFactory(java.util.Hashtable env,
                                              java.lang.String conn_name)
                                       throws JMSException
```

This method unregisters a `QueueConnectionFactory` or TopicConnectionFactory in LDAP. It is static and has the following parameters:

Parameter	Description
`env`	Environment of LDAP connection
`conn_name`	Name of the connection to be registered

The hash table passed to `unregisterConnectionFactory()` must contain all the information to establish a valid connection to the LDAP server. Furthermore, the connection must have write access to the connection factory entries in the LDAP server (which requires the LDAP user to be either the database itself or be granted `GLOBAL_AQ_USER_ROLE`).

Example 6-6 Unregistering Through LDAP

```
Hashtable            env = new Hashtable(5, 0.75f);

/* the following statements set in hashtable env:
   * service provider package
   * the distinguished name of the database server
   * the authentication method (simple)
   * the LDAP username
   * the LDAP user password
*/
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://sun-456:389");
env.put("searchbase", "cn=db1,cn=Oraclecontext,cn=acme,cn=com");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=db1aqadmin,cn=acme,cn=com");
env.put(Context.SECURITY_CREDENTIALS, "welcome");
url = "jdbc:oracle:thin:@sun-123:1521:db1";
AQjmsFactory.unregisterConnectionFactory(env, "queue_conn1");
```

Getting a QueueConnectionFactory or TopicConnectionFactory

This section contains these topics:

- Getting a QueueConnectionFactory with JDBC URL
- Getting a QueueConnectionFactory with JDBC Connection Parameters
- Getting a TopicConnectionFactory with JDBC URL
- Getting a TopicConnectionFactory with JDBC Connection Parameters
- Getting a QueueConnectionFactory or TopicConnectionFactory in LDAP

Getting a QueueConnectionFactory with JDBC URL

```
public static javax.jms.QueueConnectionFactory getQueueConnectionFactory(
              java.lang.String jdbc_url,
              java.util.Properties info)
       throws JMSException
```

This method gets a `QueueConnectionFactory` with JDBC URL. It is static and has the following parameters:

Parameter	Description
`jdbc_url`	URL to connect to
info	Properties information

Example 6-7 Getting a QueueConnectionFactory with JDBC URL

```
String      url         = "jdbc:oracle:oci10:internal/oracle"
Properties  info        = new Properties();
QueueConnectionFactory   qc_fact;

info.put("internal_logon", "sysdba");
qc_fact = AQjmsFactory.getQueueConnectionFactory(url, info);
```

Getting a QueueConnectionFactory with JDBC Connection Parameters

```
public static javax.jms.QueueConnectionFactory getQueueConnectionFactory(
              java.lang.String hostname,
              java.lang.String oracle_sid,
              int portno,
              java.lang.String driver)
       throws JMSException
```

This method gets a `QueueConnectionFactory` with JDBC connection parameters. It is static and has the following parameters:

Parameter	Description
`hostname`	Name of the host running Oracle Database Advanced Queuing
`oracle_sid`	Oracle system identifier
`portno`	Port number
`driver`	JDBC driver type

Example 6-8 Getting a QueueConnectionFactory with JDBC Connection Parameters

```
String      host        = "dlsun";
String      ora_sid     = "rdbms10i"
String      driver      = "thin";
int         port        = 5521;
QueueConnectionFactory   qc_fact;

qc_fact = AQjmsFactory.getQueueConnectionFactory(host, ora_sid, port, driver);
```

Getting a TopicConnectionFactory with JDBC URL

```
public static javax.jms.QueueConnectionFactory getQueueConnectionFactory(
              java.lang.String jdbc_url,
              java.util.Properties info)
       throws JMSException
```

This method gets a `TopicConnectionFactory` with a JDBC URL. It is static and has the following parameters:

Parameter	Description
`jdbc_url`	URL to connect to
info	Properties information

Example 6-9 Getting a TopicConnectionFactory with JDBC URL

```
String      url          = "jdbc:oracle:oci10:internal/oracle"
Properties  info         = new Properties();
TopicConnectionFactory   tc_fact;

info.put("internal_logon", "sysdba");
tc_fact = AQjmsFactory.getTopicConnectionFactory(url, info);
```

Getting a TopicConnectionFactory with JDBC Connection Parameters

```
public static javax.jms.TopicConnectionFactory getTopicConnectionFactory(
              java.lang.String hostname,
              java.lang.String oracle_sid,
              int portno,
              java.lang.String driver)
       throws JMSException
```

This method gets a `TopicConnectionFactory` with JDBC connection parameters. It is static and has the following parameters:

Parameter	Description
`hostname`	Name of the host running Oracle Database Advanced Queuing
`oracle_sid`	Oracle system identifier
`portno`	Port number
`driver`	JDBC driver type

Example 6-10 Getting a TopicConnectionFactory with JDBC Connection Parameters

```
String      host         = "dlsun";
String      ora_sid      = "rdbms10i"
String      driver       = "thin";
int         port         = 5521;
TopicConnectionFactory   tc_fact;

tc_fact = AQjmsFactory.getTopicConnectionFactory(host, ora_sid, port, driver);
```

Getting a QueueConnectionFactory or TopicConnectionFactory in LDAP

This method gets a `QueueConnectionFactory` or `TopicConnectionFactory` from LDAP.

Example 6-11 Getting a QueueConnectionFactory or TopicConnectionFactory in LDAP

```
Hashtable              env = new Hashtable(5, 0.75f);
DirContext             ctx;
queueConnectionFactory qc_fact;

/* the following statements set in hashtable env:
   * service provider package
   * the URL of the ldap server
   * the distinguished name of the database server
   * the authentication method (simple)
   * the LDAP username
   * the LDAP user password
*/
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://sun-456:389");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=db1aquser1,cn=acme,cn=com");
env.put(Context.SECURITY_CREDENTIALS, "welcome");

ctx = new InitialDirContext(env);
ctx =
(DirContext)ctx.lookup("cn=OracleDBConnections,cn=db1,cn=Oraclecontext,cn=acme,cn=com");
qc_fact = (queueConnectionFactory)ctx.lookup("cn=queue_conn1");
```

Getting a Queue or Topic in LDAP

This method gets a queue or topic from LDAP.

Example 6-12 Getting a Queue or Topic in LDAP

```
Hashtable               env = new Hashtable(5, 0.75f);
DirContext              ctx;
topic                   topic_1;

/* the following statements set in hashtable env:
   * service provider package
   * the URL of the ldap server
   * the distinguished name of the database server
   * the authentication method (simple)
   * the LDAP username
   * the LDAP user password
*/
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://sun-456:389");
env.put(Context.SECURITY_AUTHENTICATION, "simple");
env.put(Context.SECURITY_PRINCIPAL, "cn=db1aquser1,cn=acme,cn=com");
env.put(Context.SECURITY_CREDENTIALS, "welcome");

ctx = new InitialDirContext(env);
ctx = (DirContext)ctx.lookup("cn=OracleDBQueues,cn=db1,cn=Oraclecontext,cn=acme,cn=com");
topic_1 = (topic)ctx.lookup("cn=topic_1");
```

Creating an AQ Queue Table

```
public oracle.AQ.AQQueueTable createQueueTable(
                 java.lang.String owner,
                 java.lang.String name,
                 oracle.AQ.AQQueueTableProperty property)
          throws JMSException
```

This method creates a queue table. It has the following parameters:

Parameter	Description
`owner`	Queue table owner (schema)
name	Queue table name
property	Queue table properties

If the queue table is used to hold queues, then the queue table must not be multiconsumer enabled (default). If the queue table is used to hold topics, then the queue table must be multiconsumer enabled.

CLOB, BLOB, and BFILE objects are valid attributes for an Oracle Database Advanced Queuing object type load. However, only CLOB and BLOB can be propagated using Oracle Database Advanced Queuing propagation in Oracle8*i* and after.

Note:

Currently TxEventQ queues can be created and dropped only through the `DBMS_AQADM` PL/SQL APIs.

Example 6-13 Creating a Queue Table

```
QueueSession              q_sess    = null;
AQQueueTable              q_table   = null;
AQQueueTableProperty      qt_prop   = null;

qt_prop = new AQQueueTableProperty("SYS.AQ$_JMS_BYTES_MESSAGE");
q_table = ((AQjmsSession)q_sess).createQueueTable(
    "boluser", "bol_ship_queue_table", qt_prop);
```

Creating a Queue

This section contains these topics:

- Creating a Point-to-Point Queue
- Creating a Publish/Subscribe Topic
- Creating a TxEventQ Queue for Point-to-Point Queue and Publish/Subscribe Topic

Creating a Point-to-Point Queue

```
public javax.jms.Queue createQueue(
             oracle.AQ.AQQueueTable q_table,
             java.lang.String queue_name,
             oracle.jms.AQjmsDestinationProperty dest_property)
      throws JMSException
```

This method creates a queue in a specified queue table. It has the following parameters:

Parameter	Description
`q_table`	Queue table in which the queue is to be created. The queue table must be single-consumer.
`queue_name`	Name of the queue to be created
`dest_property`	Queue properties

This method is specific to Oracle JMS. You cannot use standard Java `javax.jms.Session` objects with it. Instead, you must cast the standard type to the Oracle JMS concrete class `oracle.jms.AQjmsSession`.

Example 6-14 Creating a Point-to-Point Queue

```
QueueSession                q_sess;
AQQueueTable                q_table;
AqjmsDestinationProperty dest_prop;
Queue                       queue;

queue = ((AQjmsSession)q_sess).createQueue(q_table, "jms_q1", dest_prop);
```

Creating a Publish/Subscribe Topic

```
public javax.jms.Topic createTopic(
             oracle.AQ.AQQueueTable q_table,
             java.lang.String topic_name,
             oracle.jms.AQjmsDestinationProperty dest_property)
      throws JMSException
```

This method creates a topic in the publish/subscribe model. It has the following parameters:

Parameter	Description
`q_table`	Queue table in which the queue is to be created. The queue table must be multiconsumer.
`queue_name`	Name of the queue to be created
`dest_property`	Queue properties

This method is specific to Oracle JMS. You cannot use standard Java `javax.jms.Session` objects with it. Instead, you must cast the standard type to the Oracle JMS concrete class `oracle.jms.AQjmsSession`.

In Example 6-16, if an order cannot be filled because of insufficient inventory, then the transaction processing the order is terminated. The `bookedorders` topic is set up with `max_retries` = 4 and `retry_delay` = 12 hours.Thus, if an order is not filled up in two days, then it is moved to an exception queue.

Example 6-15 Creating a Publish/Subscribe Topic

```
TopicSession                t_sess;
AQQueueTable                q_table;
AqjmsDestinationProperty dest_prop;
Topic                       topic;

topic = ((AQjmsSessa)t_sess).createTopic(q_table, "jms_t1", dest_prop);
```

Example 6-16 Specifying Max Retries and Max Delays in Messages

```
public BolOrder process_booked_order(TopicSession jms_session)
  {
    Topic             topic;
    TopicSubscriber   tsubs;
    ObjectMessage     obj_message;
    BolCustomer       customer;
    BolOrder          booked_order = null;
    String            country;
    int               i = 0;

    try
    {
      /* get a handle to the OE_bookedorders_topic */
      topic = ((AQjmsSession)jms_session).getTopic("WS",
                                                   "WS_bookedorders_topic");

      /* Create local subscriber - to track messages for Western Region  */
      tsubs = jms_session.createDurableSubscriber(topic, "SUBS1",
                                       "Region = 'Western' ",
                                                   false);

       /* wait for a message to show up in the topic */
       obj_message = (ObjectMessage)tsubs.receive(10);

       booked_order = (BolOrder)obj_message.getObject();

       customer = booked_order.getCustomer();
       country    = customer.getCountry();

       if (country == "US")
       {
          jms_session.commit();
       }
       else
       {
          jms_session.rollback();
          booked_order = null;
       }
    }catch (JMSException ex)
    { System.out.println("Exception " + ex) ;}

     return booked_order;
   }
```

Creating a TxEventQ Queue for Point-to-Point Queue and Publish/Subscribe Topic

AQ JMS has defined a new APIs to create and drop TxEventQ queues. There is no alter queue API in JMS. The signatures are as follows:

```
    /**
     * Create a TxEventQ queue. It also internally creates the related queue
     * objects (table, indexes) based on this name.
     *
     * @param queueName name of the queue to be created, format is schema.queueName
     *        (where the schema. is optional
     * @param isMultipleConsumer flag to indicate whether the queue is a
     *        multi-consumer or single-consumer queue
     * @return javax.jms.Destination
     * @throws JMSException if the queue could not be created
     */
    public synchronized javax.jms.Destination createJMSTransactionalEventQueue(String
queueName,
            boolean isMultipleConsumer) throws JMSException {
        return createJMSTransactionalEventQueue(queueName, isMultipleConsumer, null, 0,
null);
    }


    /**
     * Create a TxEventQ queue. It also internally creates the related queue
     * objects (table, indexes) based on this name.
     *
     * @param queueName name of the queue to be created, format is schema.queueName
     *        (where the schema. is optional
     * @param isMultipleConsumer flag to indicate whether the queue is a
     *        multi-consumer or single-consumer queue
     * @param storageClause additional storage clause
     * @param maxRetries retry count before skip the message while dequeue
     * @param comment comment for the queue
     * @return javax.jms.Destination
     * @throws JMSException if the queue could not be created
*/
public Destination createJMSTransactionalEventQueue(java.lang.String queueName,
                                         boolean isMultipleConsumer,
                                         java.lang.String storageClause,
                                         int maxRetries,
                                         java.lang.String comment)
                                  throws JMSException
```

Getting an AQ Queue Table

```
public oracle.AQ.AQQueueTable getQueueTable(java.lang.String owner,
                                            java.lang.String name)
                                     throws JMSException
```

This method gets a queue table for an AQ queue. It has the following parameters:

Parameter	Description
`owner`	Queue table owner (schema)
name	Queue table name

If the caller that opened the connection is not the owner of the queue table, then the caller must have Oracle Database Advanced Queuing enqueue/dequeue privileges on queues/topics in the queue table. Otherwise the queue table is not returned.

Example 6-17   Getting a Queue Table

```
QueueSession                q_sess;
AQQueueTable                q_table;

q_table = ((AQjmsSession)q_sess).getQueueTable(
   "boluser", "bol_ship_queue_table");
```

Granting and Revoking Privileges

This section contains these topics:

- Granting Oracle Database Advanced Queuing System Privileges
- Revoking Oracle Database Advanced Queuing System Privileges
- Granting Publish/Subscribe Topic Privileges
- Revoking Publish/Subscribe Topic Privileges
- Granting Point-to-Point Queue Privileges
- Revoking Point-to-Point Queue Privileges

Granting Oracle Database Advanced Queuing System Privileges

```
public void grantSystemPrivilege(java.lang.String privilege,
                                 java.lang.String grantee,
                                 boolean admin_option)
                          throws JMSException
```

This method grants Oracle Database Advanced Queuing system privileges to a user or role.

Parameter	Description
`privilege`	`ENQUEUE_ANY`, `DEQUEUE_ANY` or `MANAGE_ANY`
`grantee`	Grantee (user, role, or `PUBLIC`)
`admin_option`	If this is set to true, then the grantee is allowed to use this procedure to grant the system privilege to other users or roles

Initially only `SYS` and `SYSTEM` can use this procedure successfully. Users granted the `ENQUEUE_ANY` privilege are allowed to enqueue messages to any queues in the database. Users granted the `DEQUEUE_ANY` privilege are allowed to dequeue messages from any queues in the database. Users granted the `MANAGE_ANY` privilege are allowed to run `DBMS_AQADM` calls on any schemas in the database.

Example 6-18 Granting Oracle Database Advanced Queuing System Privileges

```
TopicSession             t_sess;

((AQjmsSession)t_sess).grantSystemPrivilege("ENQUEUE_ANY", "scott", false);
```

Revoking Oracle Database Advanced Queuing System Privileges

```
public void revokeSystemPrivilege(java.lang.String privilege,
                                  java.lang.String grantee)
                           throws JMSException
```

This method revokes Oracle Database Advanced Queuing system privileges from a user or role. It has the following parameters:

Parameter	Description
`privilege`	`ENQUEUE_ANY`, `DEQUEUE_ANY` or `MANAGE_ANY`
`grantee`	Grantee (user, role, or `PUBLIC`)

Users granted the `ENQUEUE_ANY` privilege are allowed to enqueue messages to any queues in the database. Users granted the `DEQUEUE_ANY` privilege are allowed to dequeue messages from any queues in the database. Users granted the `MANAGE_ANY` privilege are allowed to run `DBMS_AQADM` calls on any schemas in the database.

Example 6-19 Revoking Oracle Database Advanced Queuing System Privileges

```
TopicSession             t_sess;

((AQjmsSession)t_sess).revokeSystemPrivilege("ENQUEUE_ANY", "scott");
```

Granting Publish/Subscribe Topic Privileges

```
public void grantTopicPrivilege(javax.jms.Session session,
                                java.lang.String privilege,
                                java.lang.String grantee,
                                boolean grant_option)
                         throws JMSException
```

This method grants a topic privilege in the publish/subscribe model. Initially only the queue table owner can use this procedure to grant privileges on the topic. It has the following parameters:

Parameter	Description
`session`	JMS session
`privilege`	`ENQUEUE`, `DEQUEUE`, or `ALL` (`ALL` means both.)
`grantee`	Grantee (user, role, or `PUBLIC`)
`grant_option`	If this is set to true, then the grantee is allowed to use this procedure to grant the system privilege to other users or roles

Example 6-20 Granting Publish/Subscribe Topic Privileges

```
TopicSession                  t_sess;
Topic                         topic;

((AQjmsDestination)topic).grantTopicPrivilege(
   t_sess, "ENQUEUE", "scott", false);
```

Revoking Publish/Subscribe Topic Privileges

```
public void revokeTopicPrivilege(javax.jms.Session session,
                                 java.lang.String privilege,
                                 java.lang.String grantee)
                          throws JMSException
```

This method revokes a topic privilege in the publish/subscribe model. It has the following parameters:

Parameter	Description
`session`	JMS session
`privilege`	`ENQUEUE`, `DEQUEUE`, or `ALL` (`ALL` means both.)
`grantee`	Revoked grantee (user, role, or `PUBLIC`)

Example 6-21 Revoking Publish/Subscribe Topic Privileges

```
TopicSession                  t_sess;
Topic                         topic;

((AQjmsDestination)topic).revokeTopicPrivilege(t_sess, "ENQUEUE", "scott");
```

Granting Point-to-Point Queue Privileges

```
public void grantQueuePrivilege(javax.jms.Session session,
                                java.lang.String privilege,
                                java.lang.String grantee,
                                boolean grant_option)
                         throws JMSException
```

This method grants a queue privilege in the point-to-point model. Initially only the queue table owner can use this procedure to grant privileges on the queue. It has the following parameters:

Parameter	Description
`session`	JMS session
`privilege`	`ENQUEUE`, `DEQUEUE`, or `ALL` (`ALL` means both.)

Parameter	Description
grantee	Grantee (user, role, or PUBLIC)
grant_option	If this is set to true, then the grantee is allowed to use this procedure to grant the system privilege to other users or roles

Example 6-22 Granting Point-to-Point Queue Privileges

```
QueueSession              q_sess;
Queue                     queue;

((AQjmsDestination)queue).grantQueuePrivilege(
   q_sess, "ENQUEUE", "scott", false);
```

Revoking Point-to-Point Queue Privileges

```
public void revokeQueuePrivilege(javax.jms.Session session,
                                 java.lang.String privilege,
                                 java.lang.String grantee)
                          throws JMSException
```

This method revokes queue privileges in the point-to-point model. Initially only the queue table owner can use this procedure to grant privileges on the queue. It has the following parameters:

Parameter	Description
session	JMS session
privilege	ENQUEUE, DEQUEUE, or ALL (ALL means both.)
grantee	Revoked grantee (user, role, or PUBLIC)

To revoke a privilege, the revoker must be the original grantor of the privilege. Privileges propagated through the GRANT option are revoked if the grantor privilege is also revoked.

Example 6-23 Revoking Point-to-Point Queue Privileges

```
QueueSession              q_sess;
Queue                     queue;

((AQjmsDestination)queue).revokeQueuePrivilege(q_sess, "ENQUEUE", "scott");
```

Managing Destinations

This section contains these topics:

- Starting a Destination
- Stopping a Destination
- Altering a Destination
- Dropping a Destination

Note:

Currently TEQs can be managed only through the DBMS_AQADM PL/SQL APIs.

Starting a Destination

```
public void start(javax.jms.Session session,
                  boolean enqueue,
                  boolean dequeue)
           throws JMSException
```

This method starts a destination. It has the following parameters:

Parameter	Description
`session`	JMS session
`enqueue`	If set to `TRUE`, then enqueue is enabled
dequeue	If set to `TRUE`, then dequeue is enabled

Example 6-24 Starting a Destination

```
TopicSession t_sess;
QueueSession q_sess;
Topic        topic;
Queue        queue;

(AQjmsDestination)topic.start(t_sess, true, true);
(AQjmsDestination)queue.start(q_sess, true, true);
```

Stopping a Destination

```
public void stop(javax.jms.Session session,
                 boolean enqueue,
                 boolean dequeue,
                 boolean wait)
          throws JMSException
```

This method stops a destination. It has the following parameters:

Parameter	Description
`session`	JMS session
`enqueue`	If set to `TRUE`, then enqueue is disabled
dequeue	If set to `TRUE`, then dequeue is disabled
`wait`	If set to true, then pending transactions on the queue/topic are allowed to complete before the destination is stopped

Example 6-25 Stopping a Destination

```
TopicSession t_sess;
Topic        topic;

((AQjmsDestination)topic).stop(t_sess, true, false);
```

Altering a Destination

```
public void alter(javax.jms.Session session,
                  oracle.jms.AQjmsDestinationProperty dest_property)
           throws JMSException
```

This method alters a destination. It has the following properties:

Parameter	Description
`session`	JMS session
dest_property	New properties of the queue or topic

Example 6-26 Altering a Destination

```
QueueSession q_sess;
Queue        queue;
TopicSession t_sess;
Topic        topic;
AQjmsDestionationProperty dest_prop1, dest_prop2;

((AQjmsDestination)queue).alter(dest_prop1);
((AQjmsDestination)topic).alter(dest_prop2);
```

Dropping a Destination

```
public void drop(javax.jms.Session session)
          throws JMSException
```

This method drops a destination. It has the following parameter:

Parameter	Description
`session`	JMS session

Example 6-27 Dropping a Destination

```
QueueSession q_sess;
Queue        queue;
TopicSession t_sess;
Topic        topic;

((AQjmsDestionation)queue).drop(q_sess);
((AQjmsDestionation)topic).drop(t_sess);
```

Propagation Schedules

This section contains these topics:

- Scheduling a Propagation
- Enabling a Propagation Schedule
- Altering a Propagation Schedule
- Disabling a Propagation Schedule
- Unscheduling a Propagation

Note:

TxEventQs are currently managed only through the `DBMS_AQADM` PL/SQL APIs and do not support propagation.

Scheduling a Propagation

```
public void schedulePropagation(javax.jms.Session session,
                                java.lang.String destination,
                                java.util.Date start_time,
                                java.lang.Double duration,
                                java.lang.String next_time,
                                java.lang.Double latency)
                         throws JMSException
```

This method schedules a propagation. It has the following parameters:

Parameter	Description
`session`	JMS session
`destination`	Database link of the remote database for which propagation is being scheduled. A null string means that propagation is scheduled for all subscribers in the database of the topic.
`start_time`	Time propagation starts
`duration`	Duration of propagation
`next_time`	Next time propagation starts
`latency`	Latency in seconds that can be tolerated. Latency is the difference between the time a message was enqueued and the time it was propagated.

If a message has multiple recipients at the same destination in either the same or different queues, then it is propagated to all of them at the same time.

Example 6-28 Scheduling a Propagation

```
TopicSession t_sess;
Topic        topic;

((AQjmsDestination)topic).schedulePropagation(
   t_sess, null, null, null, null, new Double(0));
```

Enabling a Propagation Schedule

```
public void enablePropagationSchedule(javax.jms.Session session,
                                      java.lang.String destination)
                               throws JMSException
```

This method enables a propagation schedule. It has the following parameters:

Parameter	Description
`session`	JMS session
`destination`	Database link of the destination database. A null string means that propagation is to the local database.

Example 6-29 Enabling a Propagation Schedule

```
TopicSession            t_sess;
Topic                   topic;

((AQjmsDestination)topic).enablePropagationSchedule(t_sess, "dbs1");
```

Altering a Propagation Schedule

```
public void alterPropagationSchedule(javax.jms.Session session,
                                     java.lang.String destination,
                                     java.lang.Double duration,
                                     java.lang.String next_time,
                                     java.lang.Double latency)
                              throws JMSException
```

This method alters a propagation schedule. It has the following parameters:

Parameter	Description
`session`	JMS session
`destination`	Database link of the remote database for which propagation is being scheduled. A null string means that propagation is scheduled for all subscribers in the database of the topic.
`duration`	Duration of propagation
`next_time`	Next time propagation starts
`latency`	Latency in seconds that can be tolerated. Latency is the difference between the time a message was enqueued and the time it was propagated.

Example 6-30   Altering a Propagation Schedule

```
TopicSession              t_sess;
Topic                     topic;

((AQjmsDestination)topic).alterPropagationSchedule(
   t_sess, null, 30, null, new Double(30));
```

Disabling a Propagation Schedule

```
public void disablePropagationSchedule(javax.jms.Session session,
                                       java.lang.String destination)
                                throws JMSException
```

This method disables a propagation schedule. It has the following parameters:

Parameter	Description
`session`	JMS session
`destination`	Database link of the destination database. A null string means that propagation is to the local database.

Example 6-31   Disabling a Propagation Schedule

```
TopicSession              t_sess;
Topic                     topic;

((AQjmsDestination)topic).disablePropagationSchedule(t_sess, "dbs1");
```

Unscheduling a Propagation

```
public void unschedulePropagation(javax.jms.Session session,
                                  java.lang.String destination)
                           throws JMSException
```

This method unschedules a previously scheduled propagation. It has the following parameters:

Parameter	Description
`session`	JMS session
`destination`	Database link of the destination database. A null string means that propagation is to the local database.

Example 6-32 Unscheduling a Propagation

```
TopicSession   t_sess;
Topic          topic;

((AQjmsDestination)topic).unschedulePropagation(t_sess, "dbs1");
```

Oracle Java Message Service Point-to-Point

The following topics describe the components of the Oracle Database Advanced Queuing (AQ) Java Message Service (JMS) operational interface that are specific to point-to-point operations. Components that are shared by point-to-point and publish/subscribe are described in Oracle Java Message Service Shared Interfaces.

- Creating a Connection with User Name/Password
- Creating a Connection with Default ConnectionFactory Parameters
- Creating a QueueConnection with User Name/Password
- Creating a QueueConnection with an Open JDBC Connection
- Creating a QueueConnection with Default ConnectionFactory Parameters
- Creating a QueueConnection with an Open OracleOCIConnectionPool
- Creating a Session
- Creating a QueueSession
- Creating a QueueSender
- Sending Messages Using a QueueSender with Default Send Options
- Sending Messages Using a QueueSender by Specifying Send Options
- Creating a QueueBrowser for Standard JMS Type Messages
- Creating a QueueBrowser for Standard JMS Type Messages_ Locking Messages
- Creating a QueueBrowser for Oracle Object Type Messages
- Creating a QueueBrowser for Oracle Object Type Messages_ Locking Messages
- Creating a QueueReceiver for Standard JMS Type Messages
- Creating a QueueReceiver for Oracle Object Type Messages

Creating a Connection with User Name/Password

```
public javax.jms.Connection createConnection(
              java.lang.String username,
              java.lang.String password)
       throws JMSException
```

This method creates a connection supporting both point-to-point and publish/subscribe operations with the specified user name and password. This method is new and supports JMS version 1.1 specifications. It has the following parameters:

Parameter	Description
`username`	Name of the user connecting to the database for queuing
`password`	Password for creating the connection to the server

Creating a Connection with Default ConnectionFactory Parameters

```
public javax.jms.Connection createConnection()
      throws JMSException
```

This method creates a connection supporting both point-to-point and publish/subscribe operations with default ConnectionFactory parameters. This method is new and supports JMS version 1.1 specifications. If the `ConnectionFactory` properties do not contain a default user name and password, then it throws a JMSException.

Creating a QueueConnection with User Name/Password

```
public javax.jms.QueueConnection createQueueConnection(
             java.lang.String username,
             java.lang.String password)
      throws JMSException
```

This method creates a queue connection with the specified user name and password. It has the following parameters:

Parameter	Description
`username`	Name of the user connecting to the database for queuing
`password`	Password for creating the connection to the server

Example 6-33   Creating a QueueConnection with User Name/Password

```
QueueConnectionFactory qc_fact = AQjmsFactory.getQueueConnectionFactory(
   "sun123", "oratest", 5521, "thin");
QueueConnection qc_conn = qc_fact.createQueueConnection("jmsuser", "jmsuser");
```

Creating a QueueConnection with an Open JDBC Connection

```
public static javax.jms.QueueConnection createQueueConnection(
   java.sql.Connection jdbc_connection)
   throws JMSException
```

This method creates a queue connection with an open JDBC connection. It is static and has the following parameter:

Parameter	Description
`jdbc_connection`	Valid open connection to the database

The method in Example 6-34 can be used if the user wants to use an existing JDBC connection (say from a connection pool) for JMS operations. In this case JMS does not open a new connection, but instead uses the supplied JDBC connection to create the JMS `QueueConnection` object.

The method in Example 6-35 is the only way to create a JMS `QueueConnection` when using JMS from a Java stored procedures inside the database (JDBC Server driver)

Example 6-34 Creating a QueueConnection with an Open JDBC Connection

```
Connection db_conn;     /* previously opened JDBC connection */
QueueConnection qc_conn = AQjmsQueueConnectionFactory.createQueueConnection(
           db_conn);
```

Example 6-35 Creating a QueueConnection from a Java Procedure Inside Database

```
OracleDriver ora = new OracleDriver();
QueueConnection qc_conn =
AQjmsQueueConnectionFactory.createQueueConnection(ora.defaultConnection());
```

Creating a QueueConnection with Default ConnectionFactory Parameters

```
public javax.jms.QueueConnection createQueueConnection()
      throws JMSException
```

This method creates a queue connection with default ConnectionFactory parameters. If the queue connection factory properties do not contain a default user name and password, then it throws a JMSException.

Creating a QueueConnection with an Open OracleOCIConnectionPool

```
public static javax.jms.QueueConnection createQueueConnection(
          oracle.jdbc.pool.OracleOCIConnectionPool cpool)
   throws JMSException
```

This method creates a queue connection with an open `OracleOCIConnectionPool`. It is static and has the following parameter:

Parameter	Description
`cpool`	Valid open OCI connection pool to the database

The method in Example 6-36 can be used if the user wants to use an existing `OracleOCIConnectionPool` instance for JMS operations. In this case JMS does not open an new `OracleOCIConnectionPool` instance, but instead uses the supplied `OracleOCIConnectionPool` instance to create the JMS QueueConnection object.

Example 6-36 Creating a QueueConnection with an Open OracleOCIConnectionPool

```
OracleOCIConnectionPool cpool; /* previously created OracleOCIConnectionPool */
QueueConnection qc_conn = AQjmsQueueConnectionFactory.createQueueConnection(cpool);
```

Creating a Session

```
public javax.jms.Session createSession(boolean transacted,
                                       int ack_mode)
                                throws JMSException
```

This method creates a `Session`, which supports both point-to-point and publish/subscribe operations. This method is new and supports JMS version 1.1 specifications. Transactional and nontransactional sessions are supported. It has the following parameters:

Parameter	Description
`transacted`	If set to true, then the session is transactional
`ack_mode`	Indicates whether the consumer or the client will acknowledge any messages it receives. It is ignored if the session is transactional. Legal values are `Session.AUTO_ACKNOWLEDGE`, `Session.CLIENT_ACKNOWLEDGE`, and `Session.DUPS_OK_ACKNOWLEDGE`.

Creating a QueueSession

```
public javax.jms.QueueSession createQueueSession(
   boolean transacted, int ack_mode)
        throws JMSException
```

This method creates a `QueueSession`. Transactional and nontransactional sessions are supported. It has the following parameters:

Parameter	Description
`transacted`	If set to true, then the session is transactional
`ack_mode`	Indicates whether the consumer or the client will acknowledge any messages it receives. It is ignored if the session is transactional. Legal values are `Session.AUTO_ACKNOWLEDGE`, `Session.CLIENT_ACKNOWLEDGE`, and `Session.DUPS_OK_ACKNOWLEDGE`.

Example 6-37 Creating a Transactional QueueSession

```
QueueConnection qc_conn;
QueueSession  q_sess = qc_conn.createQueueSession(true, 0);
```

Creating a QueueSender

```
public javax.jms.QueueSender createSender(javax.jms.Queue queue)
                                   throws JMSException
```

This method creates a `QueueSender`. If a sender is created without a default queue, then the destination queue must be specified on every send operation. It has the following parameter:

Parameter	Description
`queue`	Name of destination queue

Sending Messages Using a QueueSender with Default Send Options

```
public void send(javax.jms.Queue queue,
                 javax.jms.Message message)
          throws JMSException
```

This method sends a message using a `QueueSender` with default send options. This operation uses default values for message `priority` (`1`) and `timeToLive` (`infinite`). It has the following parameters:

Parameter	Description
`queue`	Queue to send this message to
`message`	Message to send

If the `QueueSender` has been created with a default queue, then the queue parameter may not necessarily be supplied in the `send()` call. If a queue is specified in the `send()` operation, then this value overrides the default queue of the `QueueSender`.

If the `QueueSender` has been created without a default queue, then the queue parameter must be specified in every `send()` call.

Example 6-38 Creating a Sender to Send Messages to Any Queue

```
/* Create a sender to send messages to any queue */
QueueSession  jms_sess;
QueueSender  sender1;
TextMessage  message;
sender1 = jms_sess.createSender(null);
sender1.send(queue, message);
```

Example 6-39 Creating a Sender to Send Messages to a Specific Queue

```
/* Create a sender to send messages to a specific queue */
QueueSession jms_sess;
QueueSender sender2;
Queue   billed_orders_que;
TextMessage  message;
sender2 = jms_sess.createSender(billed_orders_que);
sender2.send(queue, message);
```

Sending Messages Using a QueueSender by Specifying Send Options

```
public void send(javax.jms.Queue queue,
                 javax.jms.Message message,
                 int deliveryMode,
                 int priority,
                 long timeToLive)
          throws JMSException
```

This method sends messages using a `QueueSender` by specifying send options. It has the following parameters:

Parameter	Description
`queue`	Queue to send this message to
`message`	Message to send
`deliveryMode`	Delivery mode to use
`priority`	Priority for this message
`timeToLive`	Message lifetime in milliseconds (zero is unlimited)

If the `QueueSender` has been created with a default queue, then the queue parameter may not necessarily be supplied in the `send()` call. If a queue is specified in the `send()` operation, then this value overrides the default queue of the `QueueSender`.

If the `QueueSender` has been created without a default queue, then the queue parameter must be specified in every `send()` call.

Example 6-40 Sending Messages Using a QueueSender by Specifying Send Options 1

```
/* Create a sender to send messages to any queue */
/* Send a message to new_orders_que with priority 2 and timetoLive 100000
   milliseconds */
QueueSession  jms_sess;
QueueSender  sender1;
TextMessage mesg;
Queue   new_orders_que
sender1 = jms_sess.createSender(null);
sender1.send(new_orders_que, mesg, DeliveryMode.PERSISTENT, 2, 100000);
```

Example 6-41 Sending Messages Using a QueueSender by Specifying Send Options 2

```
/* Create a sender to send messages to a specific queue */
/* Send a message with priority 1 and timetoLive 400000 milliseconds */
QueueSession jms_sess;
QueueSender sender2;
Queue   billed_orders_que;
TextMessage mesg;
sender2 = jms_sess.createSender(billed_orders_que);
sender2.send(mesg, DeliveryMode.PERSISTENT, 1, 400000);
```

Creating a QueueBrowser for Standard JMS Type Messages

```
public javax.jms.QueueBrowser createBrowser(javax.jms.Queue queue,
                                            java.lang.String messageSelector)
                                     throws JMSException
```

This method creates a `QueueBrowser` for queues with text, stream, objects, bytes or MapMessage message bodies. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered

Use methods in `java.util.Enumeration` to go through list of messages.

See Also:

"MessageSelector"

Example 6-42 Creating a QueueBrowser Without a Selector

```
/* Create a browser without a selector */
QueueSession    jms_session;
QueueBrowser    browser;
Queue           queue;
browser = jms_session.createBrowser(queue);
```

Example 6-43 Creating a QueueBrowser With a Specified Selector

```
/* Create a browser for queues with a specified selector */
QueueSession    jms_session;
QueueBrowser    browser;
Queue           queue;
/* create a Browser to look at messages with correlationID = RUSH  */
browser = jms_session.createBrowser(queue, "JMSCorrelationID = 'RUSH'");
```

Creating a QueueBrowser for Standard JMS Type Messages, Locking Messages

```
public javax.jms.QueueBrowser createBrowser(javax.jms.Queue queue,
                                            java.lang.String messageSelector,
                                            boolean locked)
                                     throws JMSException
```

This method creates a `QueueBrowser` for queues with TextMessage, StreamMessage, ObjectMessage, BytesMessage, or MapMessage message bodies, locking messages while browsing. Locked messages cannot be removed by other consumers until the browsing session ends the transaction. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered
`locked`	If set to true, then messages are locked as they are browsed (similar to a SELECT for UPDATE)

Example 6-44 Creating a QueueBrowser Without a Selector, Locking Messages

```
/* Create a browser without a selector */
QueueSession    jms_session;
QueueBrowser    browser;
Queue           queue;
browser = jms_session.createBrowser(queue, null, true);
```

Example 6-45 Creating a QueueBrowser With a Specified Selector, Locking Messages

```
/* Create a browser for queues with a specified selector */
QueueSession    jms_session;
QueueBrowser    browser;
Queue           queue;
/* create a Browser to look at messages with
correlationID = RUSH in lock mode */
browser = jms_session.createBrowser(queue, "JMSCorrelationID = 'RUSH'", true);
```

Creating a QueueBrowser for Oracle Object Type Messages

```
public javax.jms.QueueBrowser createBrowser(javax.jms.Queue queue,
                                            java.lang.String messageSelector,
                                            java.lang.Object payload_factory)
                                     throws JMSException
```

This method creates a `QueueBrowser` for queues of Oracle object type messages. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered
payload_factory	`CustomDatumFactory` or `ORADataFactory` for the java class that maps to the Oracle ADT

The `CustomDatumFactory` for a particular java class that maps to the SQL object payload can be obtained using the `getFactory` static method.

Note:

`CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.

Assume the queue `test_queue` has payload of type `SCOTT.EMPLOYEE` and the java class that is generated by Jpublisher for this Oracle object type is called Employee. The Employee class implements the `CustomDatum` interface. The `CustomDatumFactory` for this class can be obtained by using the `Employee.getFactory()` method.

Note:

TEQs do not support Object Type messages

See Also:

"MessageSelector"

Example 6-46 Creating a QueueBrowser for ADTMessages

```
/* Create a browser for a Queue with AdtMessage messages of type EMPLOYEE*/
QueueSession jms_session
QueueBrowser browser;
Queue        test_queue;
browser = ((AQjmsSession)jms_session).createBrowser(test_queue,
                                                   "corrid='EXPRESS'",
                                                    Employee.getFactory());
```

Creating a QueueBrowser for Oracle Object Type Messages, Locking Messages

```
public javax.jms.QueueBrowser createBrowser(javax.jms.Queue queue,
                                            java.lang.String messageSelector,
                                            java.lang.Object payload_factory,
                                            boolean locked)
                                     throws JMSException
```

This method creates a `QueueBrowser` for queues of Oracle object type messages, locking messages while browsing. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered
payload_factory	`CustomDatumFactory` or `ORADataFactory` for the java class that maps to the Oracle ADT
`locked`	If set to true, then messages are locked as they are browsed (similar to a SELECT for UPDATE)

Note:

`CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.

Note:

TxEventQ queues do not support Object Type messages

Example 6-47 Creating a QueueBrowser for AdtMessages, Locking Messages

```
/* Create a browser for a Queue with AdtMessage messagess of type EMPLOYEE* in lock mode/
QueueSession jms_session
QueueBrowser browser;
Queue        test_queue;
browser = ((AQjmsSession)jms_session).createBrowser(test_queue,
                                                    null,
                                                    Employee.getFactory(),
                                                    true);
```

Creating a QueueReceiver for Standard JMS Type Messages

```
public javax.jms.QueueReceiver createReceiver(javax.jms.Queue queue,
                                              java.lang.String messageSelector)
                                       throws JMSException
```

This method creates a `QueueReceiver` for queues of standard JMS type messages. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered

See Also:

"MessageSelector"

Example 6-48 Creating a QueueReceiver Without a Selector

```
/* Create a receiver without a selector */
QueueSession    jms_session
QueueReceiver   receiver;
Queue           queue;
receiver = jms_session.createReceiver(queue);
```

Example 6-49 Creating a QueueReceiver With a Specified Selector

```
/* Create a receiver for queues with a specified selector */
QueueSession    jms_session;
QueueReceiver   receiver;
Queue           queue;
/* create Receiver to receive messages with correlationID starting with EXP  */
browser = jms_session.createReceiver(queue, "JMSCorrelationID LIKE 'EXP%'");
```

Creating a QueueReceiver for Oracle Object Type Messages

```
public javax.jms.QueueReceiver createReceiver(javax.jms.Queue queue,
                                              java.lang.String messageSelector,
                                              java.lang.Object payload_factory)
                                       throws JMSException
```

This method creates a `QueueReceiver` for queues of Oracle object type messages. It has the following parameters:

Parameter	Description
`queue`	Queue to access
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered
payload_factory	`CustomDatumFactory` or `ORADataFactory` for the java class that maps to the Oracle ADT

The `CustomDatumFactory` for a particular java class that maps to the SQL object type payload can be obtained using the `getFactory` static method.

Note:

`CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.

Assume the queue `test_queue` has payload of type `SCOTT.EMPLOYEE` and the java class that is generated by Jpublisher for this Oracle object type is called Employee. The Employee class implements the `CustomDatum` interface. The `ORADataFactory` for this class can be obtained by using the Employee.getFactory() method.

Note:

TxEventQ queues do not support Object Type messages

See Also:

"MessageSelector"

Example 6-50 Creating a QueueReceiver for AdtMessage Messages

```
/* Create a receiver for a Queue with AdtMessage messages of type EMPLOYEE*/
QueueSession jms_session
QueueReceiver receiver;
Queue        test_queue;
browser = ((AQjmsSession)jms_session).createReceiver(
                 test_queue,
                "JMSCorrelationID = 'MANAGER',
                 Employee.getFactory());
```

Oracle Java Message Service Publish/Subscribe

The following topics describe the components of the Oracle Database Advanced Queuing (AQ) Java Message Service (JMS) operational interface that are specific to publish/subscribe operations. Components that are shared by point-to-point and publish/subscribe are described in Oracle Java Message Service Shared Interfaces.

- Creating a Connection with User Name/Password
- Creating a Connection with Default ConnectionFactory Parameters
- Creating a TopicConnection with User Name/Password
- Creating a TopicConnection with Open JDBC Connection
- Creating a TopicConnection with an Open OracleOCIConnectionPool
- Creating a Session
- Creating a TopicSession
- Creating a TopicPublisher
- Publishing Messages with Minimal Specification
- Publishing Messages Specifying Topic
- Publishing Messages Specifying Delivery Mode_ Priority_ and TimeToLive
- Publishing Messages Specifying a Recipient List
- Creating a DurableSubscriber for a JMS Topic Without Selector
- Creating a DurableSubscriber for a JMS Topic with Selector
- Creating a DurableSubscriber for an Oracle Object Type Topic Without Selector
- Creating a DurableSubscriber for an Oracle Object Type Topic with Selector
- Specifying Transformations for Topic Subscribers

- Creating a Remote Subscriber for JMS Messages
- Creating a Remote Subscriber for Oracle Object Type Messages
- Specifying Transformations for Remote Subscribers
- Unsubscribing a Durable Subscription for a Local Subscriber
- Unsubscribing a Durable Subscription for a Remote Subscriber
- Creating a TopicReceiver for a Topic of Standard JMS Type Messages
- Creating a TopicReceiver for a Topic of Oracle Object Type Messages
- Creating a TopicBrowser for Standard JMS Messages
- Creating a TopicBrowser for Standard JMS Messages_ Locking Messages
- Creating a TopicBrowser for Oracle Object Type Messages
- Creating a TopicBrowser for Oracle Object Type Messages_ Locking Messages
- Browsing Messages Using a TopicBrowser

Creating a Connection with User Name/Password

```
public javax.jms.Connection createConnection(
             java.lang.String username,
             java.lang.String password)
      throws JMSException
```

This method creates a connection supporting both point-to-point and publish/subscribe operations with the specified user name and password. This method is new and supports JMS version 1.1 specifications. It has the following parameters:

Parameter	Description
`username`	Name of the user connecting to the database for queuing
`password`	Password for creating the connection to the server

Creating a Connection with Default ConnectionFactory Parameters

```
public javax.jms.Connection createConnection()
      throws JMSException
```

This method creates a connection supporting both point-to-point and publish/subscribe operations with default ConnectionFactory parameters. This method is new and supports JMS version 1.1 specifications. If the `ConnectionFactory` properties do not contain a default user name and password, then it throws a JMSException.

Creating a TopicConnection with User Name/Password

```
public javax.jms.TopicConnection createTopicConnection(
             java.lang.String username,
             java.lang.String password)
      throws JMSException
```

This method creates a `TopicConnection` with the specified user name and password. It has the following parameters:

Parameter	Description
`username`	Name of the user connecting to the database for queuing
`password`	Password for creating the connection to the server

Example 6-51 Creating a TopicConnection with User Name/Password

```
TopicConnectionFactory tc_fact = AQjmsFactory.getTopicConnectionFactory("sun123",
"oratest", 5521, "thin");
/* Create a TopicConnection using a username/password */
TopicConnection tc_conn = tc_fact.createTopicConnection("jmsuser", "jmsuser");
```

Creating a TopicConnection with Open JDBC Connection

```
public static javax.jms.TopicConnection createTopicConnection(
              java.sql.Connection jdbc_connection)
       throws JMSException
```

This method creates a `TopicConnection` with open JDBC connection. It has the following parameter:

Parameter	Description
`jdbc_connection`	Valid open connection to database

Example 6-52 Creating a TopicConnection with Open JDBC Connection

```
Connection db_conn;   /*previously opened JDBC connection */
TopicConnection tc_conn = 
AQjmsTopicConnectionFactory createTopicConnection(db_conn);
```

Example 6-53 Creating a TopicConnection with New JDBC Connection

```
OracleDriver ora = new OracleDriver();
TopicConnection tc_conn = 
AQjmsTopicConnectionFactory.createTopicConnection(ora.defaultConnection());
```

Creating a TopicConnection with an Open OracleOCIConnectionPool

```
public static javax.jms.TopicConnection createTopicConnection(
              oracle.jdbc.pool.OracleOCIConnectionPool cpool)
       throws JMSException
```

This method creates a `TopicConnection` with an open `OracleOCIConnectionPool`. It is static and has the following parameter:

Parameter	Description
`cpool`	Valid open OCI connection pool to the database

Example 6-54 Creating a TopicConnection with Open OracleOCIConnectionPool

```
OracleOCIConnectionPool cpool; /* previously created OracleOCIConnectionPool */
TopicConnection tc_conn = 
AQjmsTopicConnectionFactory.createTopicConnection(cpool);
```

Creating a Session

```
public javax.jms.Session createSession(boolean transacted,
                                                 int ack_mode)
                                          throws JMSException
```

This method creates a `Session` supporting both point-to-point and publish/subscribe operations. It is new and supports JMS version 1.1 specifications. It has the following parameters:

Parameter	Description
`transacted`	If set to true, then the session is transactional
`ack_mode`	Indicates whether the consumer or the client will acknowledge any messages it receives. It is ignored if the session is transactional. Legal values are `Session.AUTO_ACKNOWLEDGE`, `Session.CLIENT_ACKNOWLEDGE`, and `Session.DUPS_OK_ACKNOWLEDGE`.

Creating a TopicSession

```
public javax.jms.TopicSession createTopicSession(boolean transacted,
                                                 int ack_mode)
                                          throws JMSException
```

This method creates a `TopicSession`. It has the following parameters:

Parameter	Description
`transacted`	If set to true, then the session is transactional
`ack_mode`	Indicates whether the consumer or the client will acknowledge any messages it receives. It is ignored if the session is transactional. Legal values are `Session.AUTO_ACKNOWLEDGE`, `Session.CLIENT_ACKNOWLEDGE`, and `Session.DUPS_OK_ACKNOWLEDGE`.

Example 6-55 Creating a TopicSession

```
TopicConnection tc_conn;
TopicSession t_sess = tc_conn.createTopicSession(true,0);
```

Creating a TopicPublisher

```
public javax.jms.TopicPublisher createPublisher(javax.jms.Topic topic)
                                         throws JMSException
```

This method creates a `TopicPublisher`. It has the following parameter:

Parameter	Description
`topic`	Topic to publish to, or null if this is an unidentified producer

Publishing Messages with Minimal Specification

```
public void publish(javax.jms.Message message)
             throws JMSException
```

This method publishes a message with minimal specification. It has the following parameter:

Parameter	Description
`message`	Message to send

The `TopicPublisher` uses the default values for message `priority` (`1`) and `timeToLive` (`infinite`).

Example 6-56 Publishing Without Specifying Topic

```
/* Publish without specifying topic */
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicPublisher            publisher1;
Topic                     shipped_orders;
int                       myport = 5521;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME",
          "MYSID",
           myport,
          "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
/* create TopicSession */
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
/* get shipped orders topic */
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE",
          "Shipped_Orders_Topic");
publisher1 = jms_sess.createPublisher(shipped_orders);
/* create TextMessage */
TextMessage     jms_sess.createTextMessage();
/* publish without specifying the topic */
publisher1.publish(text_message);
```

Example 6-57 Publishing Specifying Correlation and Delay

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicPublisher            publisher1;
Topic                     shipped_orders;
int                       myport = 5521;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME",
          "MYSID",
           myport,
          "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE",
          "Shipped_Orders_Topic");
publisher1 = jms_sess.createPublisher(shipped_orders);
/* Create TextMessage */
TextMessage     jms_sess.createTextMessage();
/* Set correlation and delay */
/* Set correlation */
jms_sess.setJMSCorrelationID("FOO");
```

```
/* Set delay of 30 seconds */
jms_sess.setLongProperty("JMS_OracleDelay", 30);
/* Publish  */
publisher1.publish(text_message);
```

Publishing Messages Specifying Topic

```
public void publish(javax.jms.Topic topic, javax.jms.Message message)
             throws JMSException
```

This method publishes a message specifying the topic. It has the following parameters:

Parameter	Description
`topic`	Topic to publish to
`message`	Message to send

If the `TopicPublisher` has been created with a default topic, then the `topic` parameter may not be specified in the `publish()` call. If a topic is specified, then that value overrides the default in the `TopicPublisher`. If the `TopicPublisher` has been created without a default topic, then the topic must be specified with the `publish()` call.

Example 6-58   Publishing Specifying Topic

```
/* Publish specifying topic */
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicPublisher            publisher1;
Topic                     shipped_orders;
int                       myport = 5521;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          'MYHOSTNAME', 'MYSID', myport, 'oci8');
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
/* create TopicPublisher */
publisher1 = jms_sess.createPublisher(null);
/* get topic object */
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          'WS', 'Shipped_Orders_Topic');
/* create text message */
TextMessage     jms_sess.createTextMessage();
/* publish specifying the topic */
publisher1.publish(shipped_orders, text_message);
```

Publishing Messages Specifying Delivery Mode, Priority, and TimeToLive

```
public void publish(javax.jms.Topic topic,
                    javax.jms.Message message,
                    oracle.jms.AQjmsAgent[] recipient_list,
                    int deliveryMode,
                    int priority,
                    long timeToLive)
             throws JMSException
```

This method publishes a message specifying delivery mode, priority and `TimeToLive`. It has the following parameters:

Parameter	Description
`topic`	Topic to which to publish the message (overrides the default topic of the `MessageProducer`)
`message`	Message to publish
`recipient_list`	List of recipients to which the message is published. Recipients are of type `AQjmsAgent`.
`deliveryMode`	`PERSISTENT` or `NON_PERSISTENT` (only `PERSISTENT` is supported in this release)
`priority`	Priority for this message
`timeToLive`	Message lifetime in milliseconds (zero is unlimited)

Example 6-59 Publishing Specifying Priority and TimeToLive

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicPublisher            publisher1;
Topic                     shipped_orders;
int                       myport = 5521;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
publisher1 = jms_sess.createPublisher(shipped_orders);
/* Create TextMessage */
TextMessage     jms_sess.createTextMessage();
/* Publish  message with priority 1 and time to live 200 seconds */
publisher1.publish(text_message, DeliveryMode.PERSISTENT, 1, 200000);
```

Publishing Messages Specifying a Recipient List

```
public void publish(javax.jms.Message message,
                    oracle.jms.AQjmsAgent[] recipient_list)
             throws JMSException
```

This method publishes a message specifying a recipient list overriding topic subscribers. It has the following parameters:

Parameter	Description
`message`	Message to publish
`recipient_list`	List of recipients to which the message is published. Recipients are of type `AQjmsAgent`.

Example 6-60 Publishing Specifying a Recipient List Overriding Topic Subscribers

```
/* Publish specifying priority and timeToLive */
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicPublisher            publisher1;
Topic                     shipped_orders;
```

```
int                          myport = 5521;
AQjmsAgent[]                 recipList;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
publisher1 = jms_sess.createPublisher(shipped_orders);
/* create TextMessage */
TextMessage     jms_sess.createTextMessage();
/* create two receivers */
recipList = new AQjmsAgent[2];
recipList[0] = new AQjmsAgent(
          "ES", "ES.shipped_orders_topic", AQAgent.DEFAULT_AGENT_PROTOCOL);
recipList[1] = new AQjmsAgent(
          "WS", "WS.shipped_orders_topic", AQAgent.DEFAULT_AGENT_PROTOCOL);
/* publish  message specifying a recipient list */
publisher1.publish(text_message, recipList);
```

Creating a DurableSubscriber for a JMS Topic Without Selector

```
public javax.jms.TopicSubscriber createDurableSubscriber(
             javax.jms.Topic topic,
             java.lang.String subs_name)
      throws JMSException
```

This method creates a DurableSubscriber for a JMS topic without selector. It has the following parameters:

Parameter	Description
`topic`	Non-temporary topic to subscribe to
`subs_name`	Name used to identify this subscription

Exclusive Access to Topics

`CreateDurableSubscriber()` and `Unsubscribe()` both require exclusive access to their target topics. If there are pending JMS `send()`, `publish()`, or `receive()` operations on the same topic when these calls are applied, then exception ORA - 4020 is raised. There are two solutions to the problem:

- Limit calls to `createDurableSubscriber()` and `Unsubscribe()` to the setup or cleanup phase when there are no other JMS operations pending on the topic. That makes sure that the required resources are not held by other JMS operational calls.
- Call `TopicSession.commit` before calling `createDurableSubscriber()` or `Unsubscribe()`.

Example 6-61 Creating a Durable Subscriber for a JMS Topic Without Selector

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       myport = 5521;
AQjmsAgent[]              recipList;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME",
          "MYSID",
           myport,
          "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE",
          "Shipped_Orders_Topic");
/* create a durable subscriber on the shipped_orders topic*/
subscriber1 = jms_sess.createDurableSubscriber(
          shipped_orders,
         'WesternShipping');
```

Creating a DurableSubscriber for a JMS Topic with Selector

```
public javax.jms.TopicSubscriber createDurableSubscriber(
             javax.jms.Topic topic,
             java.lang.String subs_name,
             java.lang.String messageSelector,
             boolean noLocal)
      throws JMSException
```

This method creates a durable subscriber for a JMS topic with selector. It has the following parameters:

Parameter	Description
`topic`	Non-temporary topic to subscribe to
`subs_name`	Name used to identify this subscription
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`noLocal`	If set to true, then it inhibits the delivery of messages published by its own connection

A client can change an existing durable subscription by creating a durable TopicSubscriber with the same name and a different `messageSelector`. An unsubscribe call is needed to end the subscription to the topic.

See Also:

- "Exclusive Access to Topics"
- "MessageSelector"

Example 6-62   Creating a Durable Subscriber for a JMS Topic With Selector

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       myport = 5521;
AQjmsAgent[]              recipList;
```

```
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
/* create a subscriber */
/* with condition on JMSPriority and user property 'Region' */
subscriber1 = jms_sess.createDurableSubscriber(
          shipped_orders, 'WesternShipping',
         "JMSPriority > 2 and Region like 'Western%'", false);
```

Creating a DurableSubscriber for an Oracle Object Type Topic Without Selector

```
public javax.jms.TopicSubscriber createDurableSubscriber(
             javax.jms.Topic topic,
             java.lang.String subs_name,
             java.lang.Object payload_factory)
      throws JMSException
```

This method creates a durable subscriber for an Oracle object type topic without selector. It has the following parameters:

Parameter	Description
`topic`	Non-temporary topic to subscribe to
`subs_name`	Name used to identify this subscription
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support object type messages.

See Also:

"Exclusive Access to Topics"

Example 6-63 Creating a Durable Subscriber for an Oracle Object Type Topic Without Selector

```
/* Subscribe to an ADT queue */
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
```

```
TopicSubscriber          subscriber1;
Topic                    shipped_orders;
int                      my[port = 5521;
AQjmsAgent[]             recipList;
/* the java mapping of the oracle object type created by J Publisher */
ADTMessage               message;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
/* create a subscriber, specifying the correct CustomDatumFactory */
subscriber1 = jms_sess.createDurableSubscriber(
          shipped_orders, 'WesternShipping', AQjmsAgent.getFactory());
```

Creating a DurableSubscriber for an Oracle Object Type Topic with Selector

```
public javax.jms.TopicSubscriber createDurableSubscriber(
             javax.jms.Topic topic,
             java.lang.String subs_name,
             java.lang.String messageSelector,
             boolean noLocal,
             java.lang.Object payload_factory)
      throws JMSException
```

This method creates a durable subscriber for an Oracle object type topic with selector. It has the following parameters:

Parameter	Description
topic	Non-temporary topic to subscribe to
subs_name	Name used to identify this subscription
messageSelector	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
noLocal	If set to true, then it inhibits the delivery of messages published by its own connection
payload_factory	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support object yype messages.

See Also:

"Exclusive Access to Topics"

Example 6-64 Creating a Durable Subscriber for an Oracle Object Type Topic With Selector

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       myport = 5521;
AQjmsAgent[]              recipList;
/* the java mapping of the oracle object type created by J Publisher */
ADTMessage                message;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
/* create a subscriber, specifying correct CustomDatumFactory and selector */
subscriber1 = jms_sess.createDurableSubscriber(
          shipped_orders, "WesternShipping",
         "priority > 1 and tab.user_data.region like 'WESTERN %'", false,
          ADTMessage.getFactory());
```

Specifying Transformations for Topic Subscribers

A transformation can be supplied when sending/publishing a message to a queue/topic. The transformation is applied before putting the message into the queue/topic.

The application can specify a transformation using the `setTransformation` interface in the `AQjmsQueueSender` and `AQjmsTopicPublisher` interfaces.

A transformation can also be specified when creating topic subscribers using the `CreateDurableSubscriber()` call. The transformation is applied to the retrieved message before returning it to the subscriber. If the subscriber specified in the `CreateDurableSubscriber()` call already exists, then its transformation is set to the specified transformation.

Example 6-65 Sending Messages to a Destination Using a Transformation

Suppose that the orders that are processed by the order entry application should be published to `WS_bookedorders_topic`. The transformation OE2WS (defined in the previous section) is supplied so that the messages are inserted into the topic in the correct format.

```
public void ship_bookedorders(
   TopicSession jms_session,
       AQjmsADTMessage adt_message)
{
    TopicPublisher  publisher;
    Topic           topic;

    try
    {
      /* get a handle to the WS_bookedorders_topic */
          topic = ((AQjmsSession)jms_session).getTopic("WS", "WS_bookedorders_topic");
      publisher = jms_session.createPublisher(topic);

      /* set the transformation in the publisher */
      ((AQjmsTopicPublisher)publisher).setTransformation("OE2WS");
      publisher.publish(topic, adt_message);
```

```
        }
        catch (JMSException ex)
        {
           System.out.println("Exception :" ex);
    }
}
```

Example 6-66 Specifying Transformations for Topic Subscribers

The Western Shipping application subscribes to the OE_bookedorders_topic with the transformation `OE2WS`. This transformation is applied to the messages and the returned message is of Oracle object type `WS.WS_orders`.

Suppose that the WSOrder java class has been generated by Jpublisher to map to the Oracle object `WS.WS_order`:

```
public AQjmsAdtMessage retrieve_bookedorders(TopicSession jms_session)
{
    TopicSubscriber     subscriber;
    Topic               topic;
    AQjmsAdtMessage     msg = null;

    try
    {
      /* get a handle to the OE_bookedorders_topic */
      topic = ((AQjmsSession)jms_session).getTopic("OE", "OE_bookedorders_topic");

      /* create a subscriber with the transformation OE2WS */
      subs = ((AQjmsSession)jms_session).createDurableSubscriber(
         topic, 'WShip', null, false, WSOrder.getFactory(), "OE2WS");
      msg = subscriber.receive(10);
    }
    catch (JMSException ex)
    {
        System.out.println("Exception :" ex);
    }
    return (AQjmsAdtMessage)msg;
}
```

Creating a Remote Subscriber for JMS Messages

```
public void createRemoteSubscriber(javax.jms.Topic topic,
                                   oracle.jms.AQjmsAgent remote_subscriber,
                                   java.lang.String messageSelector)
                            throws JMSException
```

This method creates a remote subscriber for topics of JMS messages. It has the following parameters:

Parameter	Description
`topic`	Topic to subscribe to
`remote_subscriber`	`AQjmsAgent` that refers to the remote subscriber
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.

Oracle Database Advanced Queuing allows topics to have remote subscribers, for example, subscribers at other topics in the same or different database. In order to use remote subscribers, you must set up propagation between the local and remote topic.

Remote subscribers can be a specific consumer at the remote topic or all subscribers at the remote topic. A remote subscriber is defined using the `AQjmsAgent` structure. An `AQjmsAgent` consists of a name and address. The name refers to the `consumer_name` at the remote topic. The address refers to the remote topic. Its syntax is `schema.topic_name[@dblink]`.

To publish messages to a particular consumer at the remote topic, the `subscription_name` of the recipient at the remote topic must be specified in the name field of `AQjmsAgent`, and the remote topic must be specified in the address field. To publish messages to all subscribers of the remote topic, the name field of `AQjmsAgent` must be set to null.

Note:

TxEventQ queues do not support remote subscribers.

See Also:

"MessageSelector"

Example 6-67 Creating a Remote Subscriber for Topics of JMS Messages

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       my[port = 5521;
AQjmsAgent                remoteAgent;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
remoteAgent = new AQjmsAgent("WesternRegion", "WS.shipped_orders_topic", null);
/* create a remote subscriber (selector is null )*/
subscriber1 = ((AQjmsSession)jms_sess).createRemoteSubscriber(
          shipped_orders, remoteAgent, null);
```

Creating a Remote Subscriber for Oracle Object Type Messages

```
public void createRemoteSubscriber(javax.jms.Topic topic,
                                   oracle.jms.AQjmsAgent remote_subscriber,
                                   java.lang.String messageSelector,
                                   java.lang.Object payload_factory)
                            throws JMSException
```

This method creates a remote subscriber for topics of Oracle object type messages. It has the following parameters:

Parameter	Description
`topic`	Topic to subscribe to
`remote_subscriber`	`AQjmsAgent` that refers to the remote subscriber
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support remote subscribers or object type messages.

Oracle Database Advanced Queuing allows topics to have remote subscribers, for example, subscribers at other topics in the same or different database. In order to use remote subscribers, you must set up propagation between the local and remote topic.

Remote subscribers can be a specific consumer at the remote topic or all subscribers at the remote topic. A remote subscriber is defined using the `AQjmsAgent` structure. An `AQjmsAgent` consists of a name and address. The name refers to the `consumer_name` at the remote topic. The address refers to the remote topic. Its syntax is `schema.topic_name[@dblink]`.

To publish messages to a particular consumer at the remote topic, the `subscription_name` of the recipient at the remote topic must be specified in the name field of `AQjmsAgent`, and the remote topic must be specified in the address field. To publish messages to all subscribers of the remote topic, the name field of `AQjmsAgent` must be set to null.

Note:

AQ does not support the use of multiple dblink to the same destination. As a workaround, use a single database link for each destination.

See Also:

"MessageSelector"

Example 6-68 Creating a Remote Subscriber for Topics of Oracle Object Type Messages

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
```

```
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       my[port = 5521;
AQjmsAgent                remoteAgent;
ADTMessage                message;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
/* create TopicSession */
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
/* get the Shipped order topic */
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
/* create a remote agent */
remoteAgent = new AQjmsAgent("WesternRegion", "WS.shipped_orders_topic", null);
/* create a remote subscriber  with null selector*/
subscriber1 = ((AQjmsSession)jms_sess).createRemoteSubscriber(
          shipped_orders, remoteAgent, null, message.getFactory);
```

Specifying Transformations for Remote Subscribers

Oracle Database Advanced Queuing allows a remote subscriber, that is a subscriber at another database, to subscribe to a topic.

Transformations can be specified when creating remote subscribers using the `createRemoteSubscriber()` call. This enables propagation of messages between topics of different formats. When a message published at a topic meets the criterion of a remote subscriber, Oracle Database Advanced Queuing automatically propagates the message to the queue/topic at the remote database specified for the remote subscriber. If a transformation is also specified, then Oracle Database Advanced Queuing applies the transformation to the message before propagating it to the queue/topic at the remote database.

Note:

TxEventQ queues do not support remote subscribers.

Example 6-69 Specifying Transformations for Remote Subscribers

A remote subscriber is created at the OE.OE_bookedorders_topic so that messages are automatically propagated to the WS.WS_bookedorders_topic. The transformation OE2WS is specified when creating the remote subscriber so that the messages reaching the WS_bookedorders_topic have the correct format.

Suppose that the WSOrder java class has been generated by Jpublisher to map to the Oracle object `WS.WS_order`

```
public void create_remote_sub(TopicSession jms_session)
{
   AQjmsAgent          subscriber;
   Topic               topic;

   try
   {
     /* get a handle to the OE_bookedorders_topic */
     topic = ((AQjmsSession)jms_session).getTopic("OE", "OE_bookedorders_topic");
     subscriber = new AQjmsAgent("WShip", "WS.WS_bookedorders_topic");
```

```
      ((AQjmsSession )jms_session).createRemoteSubscriber(
         topic, subscriber, null, WSOrder.getFactory(),"OE2WS");
    }
    catch (JMSException ex)
    {
      System.out.println("Exception :" ex);
    }
}
```

Unsubscribing a Durable Subscription for a Local Subscriber

```
public void unsubscribe(javax.jms.Topic topic,
                        java.lang.String subs_name)
                 throws JMSException
```

This method unsubscribes a durable subscription for a local subscriber. It has the following parameters:

Parameter	Description
`topic`	Non-temporary topic to unsubscribe
`subs_name`	Name used to identify this subscription

See Also:

"Exclusive Access to Topics"

Example 6-70 Unsubscribing a Durable Subscription for a Local Subscriber

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              jms_sess;
TopicSubscriber           subscriber1;
Topic                     shipped_orders;
int                       myport = 5521;
AQjmsAgent[]              recipList;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
/* unsusbcribe "WesternShipping" from shipped_orders */
jms_sess.unsubscribe(shipped_orders, "WesternShipping");
```

Unsubscribing a Durable Subscription for a Remote Subscriber

```
public void unsubscribe(javax.jms.Topic topic,
                        oracle.jms.AQjmsAgent remote_subscriber)
                 throws JMSException
```

This method unsubscribes a durable subscription for a remote subscriber. It has the following parameters:

Parameter	Description
`topic`	Non-temporary topic to unsubscribe
`remote_subscriber`	`AQjmsAgent` that refers to the remote subscriber. The address field of the `AQjmsAgent` cannot be null.

Note:

TEQ queues do not support remote subscribers.

See Also:

"Exclusive Access to Topics"

Example 6-71   Unsubscribing a Durable Subscription for a Remote Subscriber

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;
AQjmsAgent                remoteAgent;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "OE", "Shipped_Orders_Topic");
remoteAgent = new AQjmsAgent("WS", "WS.Shipped_Orders_Topic", null);
/* unsubscribe the remote agent from shipped_orders */
((AQjmsSession)jms_sess).unsubscribe(shipped_orders, remoteAgent);
```

Creating a TopicReceiver for a Topic of Standard JMS Type Messages

```
public oracle.jms.AQjmsTopicReceiver createTopicReceiver(
              javax.jms.Topic topic,
              java.lang.String receiver_name,
              java.lang.String messageSelector)
       throws JMSException
```

This method creates a `TopicReceiver` for a topic of standard JMS type messages. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`receiver_name`	Name of message receiver

Parameter	Description
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.

Oracle Database Advanced Queuing allows messages to be sent to specified recipients. These receivers may or may not be subscribers of the topic. If the receiver is not a subscriber to the topic, then it receives only those messages that are explicitly addressed to it. This method must be used order to create a `TopicReceiver` object for consumers that are not durable subscribers.

See Also:

"MessageSelector"

Example 6-72 Creating a TopicReceiver for Standard JMS Type Messages

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = ull;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;
TopicReceiver             receiver;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "WS", "Shipped_Orders_Topic");
receiver = ((AQjmsSession)jms_sess).createTopicReceiver(
          shipped_orders, "WesternRegion", null);
```

Creating a TopicReceiver for a Topic of Oracle Object Type Messages

```
public oracle.jms.AQjmsTopicReceiver createTopicReceiver(
              javax.jms.Topic topic,
              java.lang.String receiver_name,
              java.lang.String messageSelector,
              java.lang.Object payload_factory)
       throws JMSException
```

This method creates a `TopicReceiver` for a topic of Oracle object type messages with selector. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`receiver_name`	Name of message receiver
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.

Parameter	Description
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support object type messages.

Oracle Database Advanced Queuing allows messages to be sent to all subscribers of a topic or to specified recipients. These receivers may or may not be subscribers of the topic. If the receiver is not a subscriber to the topic, then it receives only those messages that are explicitly addressed to it. This method must be used order to create a `TopicReceiver` object for consumers that are not durable subscribers.

See Also:

"MessageSelector"

Example 6-73 Creating a TopicReceiver for Oracle Object Type Messages

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;
TopicReceiver             receiver;
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
          "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
          "WS", "Shipped_Orders_Topic");
receiver = ((AQjmsSession)jms_sess).createTopicReceiver(
          shipped_orders, "WesternRegion", null);
```

Creating a TopicBrowser for Standard JMS Messages

```
public oracle.jms.TopicBrowser createBrowser(javax.jms.Topic topic,
                                             java.lang.String cons_name,
                                             java.lang.String messageSelector)
                                      throws JMSException
```

This method creates a `TopicBrowser` for topics with `TextMessage`, `StreamMessage`, `ObjectMessage`, `BytesMessage`, or `MapMessage` message bodies. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`cons_name`	Name of the durable subscriber or consumer
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

See Also:

"MessageSelector"

Example 6-74 Creating a TopicBrowser Without a Selector

```
/* Create a browser without a selector */
TopicSession    jms_session;
TopicBrowser    browser;
Topic           topic;
browser = ((AQjmsSession) jms_session).createBrowser(topic, "SUBS1");
```

Example 6-75 Creating a TopicBrowser With a Specified Selector

```
/* Create a browser for topics with a specified selector */
TopicSession    jms_session;
TopicBrowser    browser;
Topic           topic;
/* create a Browser to look at messages with correlationID = RUSH  */
browser = ((AQjmsSession) jms_session).createBrowser(
          topic, "SUBS1", "JMSCorrelationID = 'RUSH'");
```

Creating a TopicBrowser for Standard JMS Messages, Locking Messages

```
public oracle.jms.TopicBrowser createBrowser(javax.jms.Topic topic,
                                             java.lang.String cons_name,
                                             java.lang.String messageSelector,
                                             boolean locked)
                                      throws JMSException
```

This method creates a `TopicBrowser` for topics with text, stream, objects, bytes or map messages, locking messages while browsing. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`cons_name`	Name of the durable subscriber or consumer
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`locked`	If set to true, then messages are locked as they are browsed (similar to a SELECT for UPDATE)

Example 6-76 Creating a TopicBrowser Without a Selector, Locking Messages While Browsing

```
/* Create a browser without a selector */
TopicSession    jms_session;
TopicBrowser    browser;
Topic           topic;
browser = ((AQjmsSession) jms_session).createBrowser(
          topic, "SUBS1", true);
```

Example 6-77 Creating a TopicBrowser With a Specified Selector, Locking Messages

```
/* Create a browser for topics with a specified selector */
TopicSession    jms_session;
TopicBrowser    browser;
Topic           topic;
/* create a Browser to look at messages with correlationID = RUSH in
lock mode */
browser = ((AQjmsSession) jms_session).createBrowser(
          topic, "SUBS1", "JMSCorrelationID = 'RUSH'", true);
```

Creating a TopicBrowser for Oracle Object Type Messages

```
public oracle.jms.TopicBrowser createBrowser(javax.jms.Topic topic,
                                             java.lang.String cons_name,
                                             java.lang.String messageSelector,
                                             java.lang.Object payload_factory)
                                      throws JMSException
```

This method creates a `TopicBrowser` for topics of Oracle object type messages. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`cons_name`	Name of the durable subscriber or consumer
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support object type messages.

The `CustomDatumFactory` for a particular Java class that maps to the SQL object type payload can be obtained using the `getFactory` static method. Assume the topic `test_topic` has payload of type `SCOTT.EMPLOYEE` and the Java class that is generated by Jpublisher for this Oracle object type is called `Employee`. The Employee class implements the `CustomDatum` interface. The `CustomDatumFactory` for this class can be obtained by using the `Employee.getFactory()` method.

See Also:

"MessageSelector"

Example 6-78 Creating a TopicBrowser for AdtMessage Messages

```
/* Create a browser for a Topic with AdtMessage messages of type EMPLOYEE*/
TopicSession jms_session
TopicBrowser browser;
Topic        test_topic;
browser = ((AQjmsSession) jms_session).createBrowser(
          test_topic, "SUBS1", Employee.getFactory());
```

Creating a TopicBrowser for Oracle Object Type Messages, Locking Messages

```
public oracle.jms.TopicBrowser createBrowser(javax.jms.Topic topic,
                                             java.lang.String cons_name,
                                             java.lang.String messageSelector,
                                             java.lang.Object payload_factory,
                                             boolean locked)
                                      throws JMSException
```

This method creates a `TopicBrowser` for topics of Oracle object type messages, locking messages while browsing. It has the following parameters:

Parameter	Description
`topic`	Topic to access
`cons_name`	Name of the durable subscriber or consumer
`messageSelector`	Only messages with properties matching the `messageSelector` expression are delivered. A value of null or an empty string indicates that there is no `messageSelector` for the message consumer.
`payload_factory`	`CustomDatumFactory` or `ORADataFactory` for the Java class that maps to the Oracle ADT
`locked`	If set to true, then messages are locked as they are browsed (similar to a SELECT for UPDATE)

Note:

- `CustomDatum` support will be deprecated in a future release. Use `ORADataFactory` payload factories instead.
- TxEventQ queues do not support object type messages.

See Also:

"MessageSelector"

Example 6-79 Creating a TopicBrowser for AdtMessage Messages, Locking Messages

```
/* Create a browser for a Topic with AdtMessage messages of type EMPLOYEE* in
lock mode/
TopicSession jms_session
TopicBrowser browser;
Topic        test_topic;
browser = ((AQjmsSession) jms_session).createBrowser(
          test_topic, "SUBS1", Employee.getFactory(), true);
```

Browsing Messages Using a TopicBrowser

```
public void purgeSeen()
               throws JMSException
```

This method browses messages using a `TopicBrowser`. Use methods in `java.util.Enumeration` to go through the list of messages. Use the method `purgeSeen` in `TopicBrowser` to purge messages that have been seen during the current browse.

Example 6-80 Creating a TopicBrowser with a Specified Selector

```
/* Create a browser for topics with a specified selector */
public void browse_rush_orders(TopicSession jms_session)
TopicBrowser    browser;
Topic           topic;
ObjectMessage   obj_message
BolOrder        new_order;
Enumeration     messages;
/* get a handle to the new_orders topic */
topic = ((AQjmsSession) jms_session).getTopic("OE", "OE_bookedorders_topic");
/* create a Browser to look at RUSH orders */
browser = ((AQjmsSession) jms_session).createBrowser(
          topic, "SUBS1", "JMSCorrelationID = 'RUSH'");
/* Browse through the messages */
for (messages = browser.elements() ; message.hasMoreElements() ;)
{obj_message = (ObjectMessage)message.nextElement();}
/* Purge messages seen during this browse */
browser.purgeSeen()
```

Oracle Java Message Service Shared Interfaces

The following topics describe the Java Message Service (JMS) operational interface (shared interfaces) to Oracle Database Advanced Queuing (AQ).

- Oracle Database Advanced Queuing JMS Operational Interface: Shared Interfaces
- Specifying JMS Message Properties
- Setting Default TimeToLive for All Messages Sent by a MessageProducer
- Setting Default Priority for All Messages Sent by a MessageProducer
- Creating an AQjms Agent
- Receiving a Message Synchronously

- Specifying the Navigation Mode for Receiving Messages
- Receiving a Message Asynchronously
- Getting Message ID
- Getting JMS Message Properties
- Closing and Shutting Down
- Troubleshooting

Oracle Database Advanced Queuing JMS Operational Interface: Shared Interfaces

The following topics discuss Oracle Database Advanced Queuing shared interfaces for JMS operations.

- Starting a JMS Connection
- Getting a JMS Connection
- Committing All Operations in a Session
- Rolling Back All Operations in a Session
- Getting the JDBC Connection from a Session
- Getting the OracleOCIConnectionPool from a JMS Connection
- Creating a BytesMessage
- Creating a MapMessage
- Creating a StreamMessage
- Creating an ObjectMessage
- Creating a TextMessage
- Creating a JMS Message
- Creating an AdtMessage
- Setting a JMS Correlation Identifier

Starting a JMS Connection

```
public void start()
          throws JMSException
```

`AQjmsConnection.start()` starts a JMS connection for receiving messages.

Getting a JMS Connection

```
public oracle.jms.AQjmsConnection getJmsConnection()
                                            throws JMSException
```

`AQjmsSession.getJmsConnection()` gets a JMS connection from a session.

Committing All Operations in a Session

```
public void commit()
           throws JMSException
```

`AQjmsSession.commit()` commits all JMS and SQL operations performed in a session.

Rolling Back All Operations in a Session

```
public void rollback()
              throws JMSException
```

`AQjmsSession.rollback()` terminates all JMS and SQL operations performed in a session.

Getting the JDBC Connection from a Session

```
public java.sql.Connection getDBConnection()
                                    throws JMSException
```

`AQjmsSession.getDBConnection()` gets the underlying JDBC connection from a JMS session. The JDBC connection can be used to perform SQL operations as part of the same transaction in which the JMS operations are accomplished.

Example 6-81   Getting Underlying JDBC Connection from JMS Session

```
java.sql.Connection db_conn;
QueueSession     jms_sess;
db_conn = ((AQjmsSession)jms_sess).getDBConnection();
```

Getting the OracleOCIConnectionPool from a JMS Connection

```
public oracle.jdbc.pool.OracleOCIConnectionPool getOCIConnectionPool()
```

`AQjmsConnection.getOCIConnectionPool()` gets the underlying `OracleOCIConnectionPool` from a JMS connection. The settings of the `OracleOCIConnectionPool` instance can be tuned by the user depending on the connection usage, for example, the number of sessions the user wants to create using the given connection. The user should not, however, close the `OracleOCIConnectionPool` instance being used by the JMS connection.

Example 6-82   Getting Underlying OracleOCIConnectionPool from JMS Connection

```
oracle.jdbc.pool.OracleOCIConnectionPool cpool;
QueueConnection jms_conn;
cpool = ((AQjmsConnection)jms_conn).getOCIConnectionPool();
```

Creating a BytesMessage

```
public javax.jms.BytesMessage createBytesMessage()
                                          throws JMSException
```

`AQjmsSession.createBytesMessage()` creates a bytes message. It can be used only if the queue table that contains the destination queue/topic was created with the `SYS.AQ$_JMS_BYTES_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

Creating a MapMessage

```
public javax.jms.MapMessage createMapMessage()
                                      throws JMSException
```

`AQjmsSession.createMapMessage()` creates a map message. It can be used only if the queue table that contains the destination queue/topic was created with the `SYS.AQ$_JMS_MAP_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

Creating a StreamMessage

```
public javax.jms.StreamMessage createStreamMessage()
                                           throws JMSException
```

`AQjmsSession.createStreamMessage()` creates a stream message. It can be used only if the queue table that contains the destination queue/topic was created with the `SYS.AQ$_JMS_STREAM_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

Creating an ObjectMessage

```
public javax.jms.ObjectMessage createObjectMessage(java.io.Serializable object)
                                           throws JMSException
```

`AQjmsSession.createObjectMessage()` creates an object message. It can be used only if the queue table that contains the destination queue/topic was created with the `SYS.AQ$_JMS_OBJECT_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

Creating a TextMessage

```
public javax.jms.TextMessage createTextMessage()
                                       throws JMSException
```

`AQjmsSession.createTextMessage()` creates a text message. It can be used only if the queue table that contains the destination queue/topic was created with the `SYS.AQ$_JMS_TEXT_MESSAGE` or `AQ$_JMS_MESSAGE` payload types.

Creating a JMS Message

```
public javax.jms.Message createMessage()
                                 throws JMSException
```

`AQjmsSession.createMessage()` creates a JMS message. You can use the `AQ$_JMS_MESSAGE` construct message to construct messages of different types. The message type must be one of the following:

- `DBMS_AQ.JMS_TEXT_MESSAGE`
- `DBMS_AQ.JMS_OBJECT_MESSAGE`
- `DBMS_AQ.JMS_MAP_MESSAGE`
- `DBMS_AQ.JMS_BYTES_MESSAGE`
- `DBMS_AQ.JMS_STREAM_MESSAGE`

You can also use this ADT to create a header-only JMS message.

Creating an AdtMessage

```
public oracle.jms.AdtMessage createAdtMessage()
                                       throws JMSException
```

`AQjmsSession.createAdtMessage()` creates an `AdtMessage`. It can be used only if the queue table that contains the queue/topic was created with an Oracle ADT payload type. An `AdtMessage` must be populated with an object that implements the `CustomDatum` interface. This object must be the Java mapping of the SQL ADT defined as the payload for the queue/topic. Java classes corresponding to SQL ADT types can be generated using the Jpublisher tool.

Setting a JMS Correlation Identifier

```
public void setJMSCorrelationID(java.lang.String correlationID)
                         throws JMSException
```

`AQjmsMessage.setJMSCorrelationID()` specifies the message correlation identifier.

Specifying JMS Message Properties

Property names starting with JMS are provider-specific. User-defined properties cannot start with JMS.

The following provider properties can be set by clients using text, stream, object, bytes or map messages:

- `JMSXAppID (string)`
- `JMSXGroupID (string)`
- `JMSXGroupSeq (int)`
- `JMS_OracleExcpQ (string)`

  This message property specifies the exception queue.
- `JMS_OracleDelay (int)`

  This message property specifies the message delay in seconds.

The following properties can be set on `AdtMessage`

- `JMS_OracleExcpQ (String)`

  This message property specifies the exception queue as "*schema*.queue_name"
- `JMS_OracleDelay (int)`

  This message property specifies the message delay in seconds.

This section contains these topics:

- Setting a Boolean Message Property
- Setting a String Message Property
- Setting an Integer Message Property
- Setting a Double Message Property
- Setting a Float Message Property
- Setting a Byte Message Property
- Setting a Long Message Property
- Setting a Short Message Property
- Getting an Object Message Property

Setting a Boolean Message Property

```
public void setBooleanProperty(java.lang.String name,
                               boolean value)
                        throws JMSException
```

`AQjmsMessage.setBooleanProperty()` specifies a message property as Boolean. It has the following parameters:

Parameter	Description
`name`	Name of the Boolean property
`value`	Boolean property value to set in the message

Setting a String Message Property

```
public void setStringProperty(java.lang.String name,
                              java.lang.String value)
                       throws JMSException
```

`AQjmsMessage.setStringProperty()` specifies a message property as string. It has the following parameters:

Parameter	Description
`name`	Name of the string property
`value`	String property value to set in the message

Setting an Integer Message Property

```
public void setIntProperty(java.lang.String name,
                           int value)
                    throws JMSException
```

`AQjmsMessage.setIntProperty()` specifies a message property as integer. It has the following parameters:

Parameter	Description
`name`	Name of the integer property
`value`	Integer property value to set in the message

Setting a Double Message Property

```
public void setDoubleProperty(java.lang.String name,
                              double value)
                       throws JMSException
```

`AQjmsMessage.setDoubleProperty()` specifies a message property as double. It has the following parameters:

Parameter	Description
`name`	Name of the double property
`value`	Double property value to set in the message

Setting a Float Message Property

```
public void setFloatProperty(java.lang.String name,
                             float value)
                      throws JMSException
```

`AQjmsMessage.setFloatProperty()` specifies a message property as float. It has the following parameters:

Parameter	Description
`name`	Name of the float property
`value`	Float property value to set in the message

Setting a Byte Message Property

```
public void setByteProperty(java.lang.String name,
                            byte value)
                     throws JMSException
```

`AQjmsMessage.setByteProperty()` specifies a message property as byte. It has the following parameters:

Parameter	Description
`name`	Name of the byte property
`value`	Byte property value to set in the message

Setting a Long Message Property

```
public void setLongProperty(java.lang.String name,
                            long value)
                     throws JMSException
```

`AQjmsMessage.setLongProperty()` specifies a message property as long. It has the following parameters:

Parameter	Description
`name`	Name of the long property
`value`	Long property value to set in the message

Setting a Short Message Property

```
public void setShortProperty(java.lang.String name,
                             short value)
                      throws JMSException
```

`AQjmsMessage.setShortProperty()` specifies a message property as short. It has the following parameters:

Parameter	Description
`name`	Name of the short property
`value`	Short property value to set in the message

Setting an Object Message Property

```
public void setObjectProperty(java.lang.String name,
                              java.lang.Object value)
                       throws JMSException
```

`AQjmsMessage.setObjectProperty()` specifies a message property as object. Only objectified primitive values are supported: Boolean, byte, short, integer, long, float, double and string. It has the following parameters:

Parameter	Description
`name`	Name of the Java object property
`value`	Java object property value to set in the message

Setting Default TimeToLive for All Messages Sent by a MessageProducer

```
public void setTimeToLive(long timeToLive)
                   throws JMSException
```

This method sets the default `TimeToLive` for all messages sent by a `MessageProducer`. It is calculated after message delay has taken effect. This method has the following parameter:

Parameter	Description
`timeToLive`	Message time to live in milliseconds (zero is unlimited)

Example 6-83 Setting Default TimeToLive for All Messages Sent by a MessageProducer

```
/* Set default timeToLive value to 100000 milliseconds for all messages sent by the
QueueSender*/
QueueSender sender;
sender.setTimeToLive(100000);
```

Setting Default Priority for All Messages Sent by a MessageProducer

```
public void setPriority(int priority)
                 throws JMSException
```

This method sets the default `Priority` for all messages sent by a `MessageProducer`. It has the following parameter:

Parameter	Description
`priority`	Message priority for this message producer. The default is 4.

Priority values can be any integer. A smaller number indicates higher priority. If a priority value is explicitly specified during a `send()` operation, then it overrides the default value set by this method.

Example 6-84 Setting Default Priority Value for All Messages Sent by QueueSender

```
/* Set default priority value to 2 for all messages sent by the QueueSender*/
QueueSender sender;
sender.setPriority(2);
```

Example 6-85 Setting Default Priority Value for All Messages Sent by TopicPublisher

```
/* Set default priority value to 2 for all messages sent by the TopicPublisher*/
TopicPublisher publisher;
publisher.setPriority(1);
```

Creating an AQjms Agent

```
public void createAQAgent(java.lang.String agent_name,
                          boolean enable_http,
                    throws JMSException
```

This method creates an `AQjmsAgent`. It has the following parameters:

Parameter	Description
`agent_name`	Name of the AQ agent
`enable_http`	If set to true, then this agent is allowed to access AQ through HTTP

Receiving a Message Synchronously

You can receive a message synchronously by specifying Timeout or without waiting. You can also receive a message using a transformation:

- Using a Message Consumer by Specifying Timeout
- Using a Message Consumer Without Waiting
- Receiving Messages from a Destination Using a Transformation

Using a Message Consumer by Specifying Timeout

```
public javax.jms.Message receive(long timeout)
                          throws JMSException
```

This method receives a message using a message consumer by specifying timeout.

Parameter	Description
`timeout`	Timeout value in milliseconds

Example 6-86 Using a Message Consumer by Specifying Timeout

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;
```

```
/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
   "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
   "WS", "Shipped_Orders_Topic");

/* create a subscriber, specifying the correct CustomDatumFactory  and
selector */
subscriber1 = jms_sess.createDurableSubscriber(
   shipped_orders, 'WesternShipping',
   " priority > 1 and tab.user_data.region like 'WESTERN %'",
   false, AQjmsAgent.getFactory());
/* receive, blocking for 30 seconds if there were no messages */
Message = subscriber.receive(30000);
```

Example 6-87 JMS: Blocking Until a Message Arrives

```
public BolOrder get_new_order1(QueueSession jms_session)
 {
   Queue            queue;
   QueueReceiver    qrec;
   ObjectMessage    obj_message;
   BolCustomer      customer;
   BolOrder         new_order = null;
   String           state;

   try
   {
   /* get a handle to the new_orders queue */
    queue = ((AQjmsSession) jms_session).getQueue("OE", "OE_neworders_que");
    qrec = jms_session.createReceiver(queue);

    /* wait for a message to show up in the queue */
    obj_message = (ObjectMessage)qrec.receive();
    new_order = (BolOrder)obj_message.getObject();
    customer = new_order.getCustomer();
    state    = customer.getState();
    System.out.println("Order:  for customer " + customer.getName());
   }
  catch (JMSException ex)
   {
     System.out.println("Exception: " + ex);
   }
   return new_order;
 }
```

Using a Message Consumer Without Waiting

```
public javax.jms.Message receiveNoWait()
                                throws JMSException
```

This method receives a message using a message consumer without waiting.

Example 6-88 JMS: Nonblocking Messages

```
public BolOrder poll_new_order3(QueueSession jms_session)
 {
   Queue            queue;
   QueueReceiver    qrec;
```

```
  ObjectMessage    obj_message;
  BolCustomer      customer;
  BolOrder         new_order = null;
  String           state;

  try
  {
   /* get a handle to the new_orders queue */
   queue = ((AQjmsSession) jms_session).getQueue("OE", "OE_neworders_que");
   qrec = jms_session.createReceiver(queue);

   /* check for a message to show in the queue */
   obj_message = (ObjectMessage)qrec.receiveNoWait();
   new_order = (BolOrder)obj_message.getObject();
   customer = new_order.getCustomer();
   state    = customer.getState();

   System.out.println("Order:  for customer " + customer.getName());
  }
  catch (JMSException ex)
  {
    System.out.println("Exception: " + ex);
  }
  return new_order;
}
```

Receiving Messages from a Destination Using a Transformation

A transformation can be applied when receiving a message from a queue or topic. The transformation is applied to the message before returning it to JMS application.

The transformation can be specified using the `setTransformation()` interface of the `AQjmsQueueReceiver`, `AQjmsTopicSubscriber` or `AQjmsTopicReceiver`.

Example 6-89 JMS: Receiving Messages from a Destination Using a Transformation

Assume that the Western Shipping application retrieves messages from the OE_bookedorders_topic. It specifies the transformation `OE2WS` to retrieve the message as the Oracle object type `WS_order`. Assume that the WSOrder Java class has been generated by Jpublisher to map to the Oracle object `WS.WS_order`:

```
public AQjmsAdtMessage retrieve_bookedorders(TopicSession jms_session)
  AQjmsTopicReceiver  receiver;
  Topic               topic;
  Message             msg = null;

  try
  {
    /* get a handle to the OE_bookedorders_topic */
    topic = ((AQjmsSession)jms_session).getTopic("OE", "OE_bookedorders_topic");

    /* Create a receiver for WShip */
    receiver = ((AQjmsSession)jms_session).createTopicReceiver(
       topic, "WShip, null, WSOrder.getFactory());

    /* set the transformation in the publisher */
    receiver.setTransformation("OE2WS");
    msg = receiver.receive(10);
  }
  catch (JMSException ex)
  {
     System.out.println("Exception :", ex);
  }
       return (AQjmsAdtMessage)msg;
}
```

Specifying the Navigation Mode for Receiving Messages

```
public void setNavigationMode(int mode)
                       throws JMSException
```

This method specifies the navigation mode for receiving messages. It has the following parameter:

Parameter	Description
`mode`	New value of the navigation mode

Example 6-90 Specifying Navigation Mode for Receiving Messages

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;

/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
   "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic("WS", "Shipped_Orders_Topic");

/* create a subscriber, specifying the correct CustomDatumFactory and selector */
subscriber1 = jms_sess.createDurableSubscriber(
   shipped_orders, 'WesternShipping',
   "priority > 1 and tab.user_data.region like 'WESTERN %'", false,
   AQjmsAgent.getFactory());
subscriber1.setNavigationMode(AQjmsConstants.NAVIGATION_FIRST_MESSAGE);

/* get message for the subscriber, returning immediately if there was nomessage */
Message = subscriber.receive();
```

Receiving a Message Asynchronously

You can receive a message asynchronously two ways:

- Specifying a Message Listener at the Message Consumer
- Specifying a Message Listener at the Session

Specifying a Message Listener at the Message Consumer

```
public void setMessageListener(javax.jms.MessageListener myListener)
                        throws JMSException
```

This method specifies a message listener at the message consumer. It has the following parameter:

Parameter	Description
`myListener`	Sets the consumer message listener

Example 6-91 Specifying Message Listener at Message Consumer

```
TopicConnectionFactory    tc_fact   = null;
TopicConnection           t_conn    = null;
TopicSession              t_sess    = null;
TopicSession              jms_sess;
Topic                     shipped_orders;
int                       myport = 5521;
MessageListener           mLis = null;

/* create connection and session */
tc_fact = AQjmsFactory.getTopicConnectionFactory(
   "MYHOSTNAME", "MYSID", myport, "oci8");
t_conn = tc_fact.createTopicConnection("jmstopic", "jmstopic");
jms_sess = t_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
shipped_orders = ((AQjmsSession )jms_sess).getTopic(
    "WS", "Shipped_Orders_Topic");

/* create a subscriber, specifying the correct CustomDatumFactory and selector */
subscriber1 = jms_sess.createDurableSubscriber(
   shipped_orders, 'WesternShipping',
   "priority > 1 and tab.user_data.region like 'WESTERN %'",
   false, AQjmsAgent.getFactory());
mLis = new myListener(jms_sess, "foo");

/* get message for the subscriber, returning immediately if there was nomessage */
subscriber.setMessageListener(mLis);
The definition of the myListener class
import oracle.AQ.*;
import oracle.jms.*;
import javax.jms.*;
import java.lang.*;
import java.util.*;
public class myListener implements MessageListener
{
   TopicSession   mySess;
   String         myName;
   /* constructor */
   myListener(TopicSession t_sess, String t_name)
   {
      mySess = t_sess;
      myName = t_name;
   }
   public onMessage(Message m)
   {
      System.out.println("Retrieved message with correlation: " ||
m.getJMSCorrelationID());
      try{
        /* commit the dequeue */
        mySession.commit();
      } catch (java.sql.SQLException e)
      {System.out.println("SQL Exception on commit"); }
   }
}
```

Specifying a Message Listener at the Session

```
public void setMessageListener(javax.jms.MessageListener listener)
                        throws JMSException
```

This method specifies a message listener at the session.

Parameter	Description
`listener`	Message listener to associate with this session

Getting Message ID

This section contains these topics:

- Getting the Correlation Identifier
- Getting the Message Identifier

Getting the Correlation Identifier

```
public java.lang.String getJMSCorrelationID()
          throws JMSException
```

`AQjmsMessage.getJMSCorrelationID()` gets the correlation identifier of a message.

Getting the Message Identifier

```
public byte[] getJMSCorrelationIDAsBytes()
          throws JMSException
```

`AQjmsMessage.getJMSMessageID()` gets the message identifier of a message as bytes or a string.

Getting JMS Message Properties

This section contains these topics:

- Getting a Boolean Message Property
- Getting a String Message Property
- Getting an Integer Message Property
- Getting a Double Message Property
- Getting a Float Message Property
- Getting a Byte Message Property
- Getting a Long Message Property
- Getting a Short Message Property
- Getting an Object Message Property

Getting a Boolean Message Property

```
public boolean getBooleanProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getBooleanProperty()` gets a message property as Boolean. It has the following parameter:

Parameter	Description
`name`	Name of the Boolean property

Getting a String Message Property

```
public java.lang.String getStringProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getStringProperty()` gets a message property as string. It has the following parameter:

Parameter	Description
`name`	Name of the string property

Getting an Integer Message Property

```
public int getIntProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getIntProperty()` gets a message property as integer. It has the following parameter:

Parameter	Description
`name`	Name of the integer property

Getting a Double Message Property

```
public double getDoubleProperty(java.lang.String name)
                         throws JMSException
```

`AQjmsMessage.getDoubleProperty()` gets a message property as double. It has the following parameter:

Parameter	Description
`name`	Name of the double property

Getting a Float Message Property

```
public float getFloatProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getFloatProperty()` gets a message property as float. It has the following parameter:

Parameter	Description
`name`	Name of the float property

Getting a Byte Message Property

```
public byte getByteProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getByteProperty()` gets a message property as byte. It has the following parameter:

Parameter	Description
`name`	Name of the byte property

Getting a Long Message Property

```
public long getLongProperty(java.lang.String name)
          throws JMSException
```

`AQjmsMessage.getLongProperty()` gets a message property as long. It has the following parameter:

Parameter	Description
`name`	Name of the long property

Getting a Short Message Property

```
public short getShortProperty(java.lang.String name)
                       throws JMSException
```

AQjmsMessage.getShortProperty() gets a message property as short. It has the following parameter:

Parameter	Description
`name`	Name of the short property

Getting an Object Message Property

```
public java.lang.Object getObjectProperty(java.lang.String name)
                                   throws JMSException
```

`AQjmsMessage.getObjectProperty()` gets a message property as object. It has the following parameter:

Parameter	Description
`name`	Name of the object property

Example 6-92 Getting Message Property as an Object

```
TextMessage message;
message.getObjectProperty("empid", new Integer(1000);
```

Closing and Shutting Down

This section contains these topics:

- Closing a MessageProducer
- Closing a Message Consumer
- Stopping a JMS Connection
- Closing a JMS Session
- Closing a JMS Connection

Closing a MessageProducer

```
public void close()
           throws JMSException
```

`AQjmsProducer.close()` closes a `MessageProducer`.

Closing a Message Consumer

```
public void close()
           throws JMSException
```

`AQjmsConsumer.close()` closes a message consumer.

Stopping a JMS Connection

```
public void stop()
          throws JMSException
```

`AQjmsConnection.stop()` stops a JMS connection.

Closing a JMS Session

```
public void close()
           throws JMSException
```

`AQjmsSession.close()` closes a JMS session.

Closing a JMS Connection

```
public void close()
          throws JMSException
```

`AQjmsConnection.close()` closes a JMS connection and releases all resources allocated on behalf of the connection. Because the JMS provider typically allocates significant resources outside the JVM on behalf of a connection, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

Troubleshooting

This section contains these topics:

- Getting a JMS Error Code
- Getting a JMS Error Number
- Getting an Exception Linked to a JMS Exception
- Printing the Stack Trace for a JMS Exception
- Setting an Exception Listener
- Getting an Exception Listener

Getting a JMS Error Code

```
public java.lang.String getErrorCode()
```

`AQjmsException.getErrorCode()` gets the error code for a JMS exception.

Getting a JMS Error Number

```
public int getErrorNumber()
```

`AQjmsException.getErrorNumber()` gets the error number for a JMS exception.

Note:

This method will be deprecated in a future release. Use `getErrorCode()` instead.

Getting an Exception Linked to a JMS Exception

```
public java.lang.String getLinkString()
```

`AQjmsException.getLinkString()` gets the exception linked to a JMS exception. In general, this contains the SQL exception raised by the database.

Printing the Stack Trace for a JMS Exception

```
public void printStackTrace(java.io.PrintStream s)
```

`AQjmsException.printStackTrace()` prints the stack trace for a JMS exception.

Setting an Exception Listener

```
public void setExceptionListener(javax.jms.ExceptionListener listener)
                          throws JMSException
```

`AQjmsConnection.setExceptionListener()` specifies an exception listener for a connection. It has the following parameter:

Parameter	Description
`listener`	Exception listener

If an exception listener has been registered, then it is informed of any serious problem detected for a connection. This is accomplished by calling the listener `onException()` method, passing it a JMS exception describing the problem. This allows a JMS client to be notified of a problem asynchronously. Some connections only consume messages, so they have no other way to learn the connection has failed.

Example 6-93 Specifying Exception Listener for Connection

```
//register an exception listener
Connection jms_connection;
jms_connection.setExceptionListener(
    new ExceptionListener() {
        public void onException (JMSException jmsException) {
            System.out.println("JMS-EXCEPTION: " + jmsException.toString());
        }
    };
  );
```

Getting an Exception Listener

```
public javax.jms.ExceptionListener getExceptionListener()
                                                  throws JMSException
```

`AQjmsConnection.getExceptionListener()` gets the exception listener for the connection.

Example 6-94 demonstrates how to use `ExceptionListener` with `MessageListener`. Ensure that the following conditions are met:

- The user `jmsuser` with password `jmsuser` is created in the database, with appropriate privileges.
- The queue `demoQueue` is created and started.

This example demonstrates how to make the MessageListener asynchronously receive the messages, where the exception listener recreates the JMS objects in case there is a connection restart.

Example 6-94 Using ExceptionListener with MessageListener

```
import java.util.Enumeration;
import java.util.Properties;

import javax.jms.Connection;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.QueueBrowser;
import javax.jms.Session;
import javax.jms.TextMessage;

import oracle.jms.AQjmsConnectionFactory;
import oracle.jms.AQjmsFactory;
import oracle.jms.AQjmsSession;
```

```
public class JMSDemo {

  static String queueName = "demoQueue";

  static String queueOwner = "jmsuser";

  static String queueOwnerPassword = "jmsuser";

  static Connection connection = null;

  static int numberOfMessages = 25000;

  static int messageCount = 0;

  static String jdbcURL = "";

  public static void main(String args[]) {
    try {
      jdbcURL = System.getProperty("JDBC_URL");

      if (jdbcURL == null)
        System.out
            .println("The system property JDBC_URL has not been set, " +
                    "usage:java -DJDBC_URL=xxx filename ");
      else {
        JMSDemo demo = new JMSDemo();
        demo.performJmsOperations();
      }
    } catch (Exception exception) {
      System.out.println("Exception : " + exception);
      exception.printStackTrace();
    } finally {
      try {
        if (connection != null)
          connection.close();
      } catch (Exception exc) {
        exc.printStackTrace();
      }
    }
    System.out.println("\nEnd of Demo aqjmsdemo11.");
  }

  public void performJmsOperations() {
    try {
      connection = getConnection(jdbcURL);
      Session session = connection.createSession(false,
          Session.AUTO_ACKNOWLEDGE);
      Queue queue = session.createQueue(queueName);

      // remove the messages from the Queue
      drainQueue(queueName, queueOwner, jdbcURL, true);

      // set the exception listener on the Connection
      connection.setExceptionListener(new DemoExceptionListener());

      MessageProducer producer = session.createProducer(queue);
      TextMessage textMessage = null;

      System.out.println("Sending " + numberOfMessages + " messages to queue "
          + queueName);
      for (int i = 0; i < numberOfMessages; i++) {
        textMessage = session.createTextMessage();
        textMessage.setText("Sample message text");
        producer.send(textMessage);
      }

      MessageConsumer consumer = session.createConsumer(queue);
      System.out.println("Setting the message listener ...");
      consumer.setMessageListener(new DemoMessageListener());
      connection.start();

      // Introduce a long wait to allow the listener to receive all the messages
      while (messageCount < numberOfMessages) {
        try {
          Thread.sleep(5000);
        } catch (InterruptedException interruptedException) {
        }
      }
    } catch (JMSException jmsException) {
      jmsException.printStackTrace();
    }
  }

  // Sample message listener
  static class DemoMessageListener implements javax.jms.MessageListener {

    public void onMessage(Message message) {
      try {
        System.out.println("Message listener received message with JMSMessageID "
                + message.getJMSMessageID());
        messageCount++;
      } catch (JMSException jmsException) {
        System.out.println("JMSException " + jmsException.getMessage());
      }
    }
  }

  // sample exception listener
  static class DemoExceptionListener implements javax.jms.ExceptionListener {

    public void onException(JMSException jmsException) {
      try {
        // As a first step close the connection
        if (connection != null)
          connection.close();
      } catch (JMSException exception) {}

      try {
        System.out.println("Re-create the necessary JMS objects ...");
        connection = getConnection(jdbcURL);
        connection.start();
        Session session = connection.createSession(false,
            Session.AUTO_ACKNOWLEDGE);
        Queue queue = session.createQueue(queueName);
        MessageConsumer consumer = session.createConsumer(queue);
        consumer.setMessageListener(new DemoMessageListener());
      } catch (JMSException newJmsException) {
        newJmsException.printStackTrace();
      }
    }
  }

  // Utility method to get a connection
  static Connection getConnection(String jdbcUrl) throws JMSException {
    Properties prop = new Properties();
    prop.put("user", queueOwner);
    prop.put("password", queueOwnerPassword);

    AQjmsConnectionFactory fact = (AQjmsConnectionFactory) AQjmsFactory
        .getConnectionFactory(jdbcUrl, prop);
    Connection conn = fact.createConnection();
    return conn;
  }

  // Utility method to remove the messages from the queue
  static void drainQueue(String queueName, String queueOwner, String jdbcUrl,
      boolean debugInfo) {
    Connection connection = null;
    Session session = null;
    long timeout = 10000;
    int count = 0;
    Message message = null;
    try {
      connection = getConnection(jdbcUrl);
      connection.start();
      session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      Queue queue = ((AQjmsSession) session).getQueue(queueOwner, queueName);

      MessageConsumer messageConsumer = session.createConsumer(queue);
      QueueBrowser browser = session.createBrowser(queue);
      Enumeration enumeration = browser.getEnumeration();

      if (enumeration.hasMoreElements()) {
        while ((message = messageConsumer.receive(timeout)) != null) {
          if (debugInfo) {
            count++;
          }
        }
      }
      messageConsumer.close();
      if (debugInfo) {
        System.out.println("Removed " + count + " messages from the queue : "
            + queueName);
      }
    } catch (JMSException jmsException) {
      jmsException.printStackTrace();
    } finally {
      try {
        if (session != null)
          session.close();

        if (connection != null)
          connection.close();
      } catch (Exception exception) {

      }
    }
  }

}
```

Example 6-95 Getting the Exception Listener for the Connection

```
//Get the exception listener
Connection jms_connection;
ExceptionListener el = jms_connection.getExceptionListener();
```

Oracle Java Message Service Types Examples

The following examples illustrate how to use Oracle JMS Types to dequeue and enqueue Oracle Database Advanced Queuing (AQ) messages.

- How to Setup the Oracle Database Advanced Queuing JMS Type Examples
- JMS BytesMessage Examples
- JMS StreamMessage Examples
- JMS MapMessage Examples
- More Oracle Database Advanced Queuing JMS Examples

How to Set Up the Oracle Database Advanced Queuing JMS Type Examples

To run Example 6-98 through Example 6-103 follow these steps:

1. Copy and save Example 6-96 as `setup.sql`.
2. Run `setup.sql` as follows:

   ```
   sqlplus /NOLOG @setup.sql
   ```

3. Log in to SQL*Plus as `jmsuser/jmsuser`.
4. Run the corresponding pair of SQL scripts for each type of message.

   For JMS `BytesMessage`, for example, run Example 6-98 and Example 6-99.

5. Ensure that your database parameter `java_pool-size` is large enough. For example, you can use `java_pool_size`=20M.

Example 6-96 Setting Up Environment for Running JMS Types Examples

```
connect sys;
enter password: password

Rem
Rem Create the JMS user: jmsuser
Rem

DROP USER jmsuser CASCADE;
CREATE USER jmsuser IDENTIFIED BY jmsuser;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;


set echo offset verify offconnect sysDROP USER jmsuser CASCADE;ACCEPT password CHAR
PROMPT 'Enter the password for JMSUSER: ' HIDECREATE USER jmsuser IDENTIFIED BY
&password;GRANT DBA, AQ_ADMINISTRATOR_ROLE, AQ_USER_ROLE to jmsuser;GRANT EXECUTE ON
DBMS_AQADM TO jmsuser;GRANT EXECUTE ON DBMS_AQ TO jmsuser;GRANT EXECUTE ON DBMS_LOB TO
jmsuser;GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;connect jmsuser/&password
```

```
Rem
Rem Creating five AQ queue tables and five queues for five payloads:
Rem SYS.AQ$_JMS_TEXT_MESSAGE
Rem SYS.AQ$_JMS_BYTES_MESSAGE
Rem SYS.AQ$_JMS_STREAM_MESSAG
Rem SYS.AQ$_JMS_MAP_MESSAGE
Rem SYS.AQ$_JMS_MESSAGE
Rem

EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (Queue_table => 'jmsuser.jms_qtt_text',
    Queue_payload_type => 'SYS.AQ$_JMS_TEXT_MESSAGE', compatible => '8.1.0');
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (Queue_table => 'jmsuser.jms_qtt_bytes',
    Queue_payload_type => 'SYS.AQ$_JMS_BYTES_MESSAGE', compatible => '8.1.0');
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (Queue_table => 'jmsuser.jms_qtt_stream',
    Queue_payload_type => 'SYS.AQ$_JMS_STREAM_MESSAGE', compatible => '8.1.0');
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (Queue_table => 'jmsuser.jms_qtt_map',
    Queue_payload_type => 'SYS.AQ$_JMS_MAP_MESSAGE', compatible => '8.1.0');
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (Queue_table => 'jmsuser.jms_qtt_general',
    Queue_payload_type => 'SYS.AQ$_JMS_MESSAGE', compatible => '8.1.0');
EXECUTE DBMS_AQADM.CREATE_QUEUE (Queue_name => 'jmsuser.jms_text_que',
    Queue_table => 'jmsuser.jms_qtt_text');
EXECUTE DBMS_AQADM.CREATE_QUEUE (Queue_name => 'jmsuser.jms_bytes_que',
    Queue_table => 'jmsuser.jms_qtt_bytes');
EXECUTE DBMS_AQADM.CREATE_QUEUE (Queue_name => 'jmsuser.jms_stream_que',
    Queue_table => 'jmsuser.jms_qtt_stream');
EXECUTE DBMS_AQADM.CREATE_QUEUE (Queue_name => 'jmsuser.jms_map_que',
    Queue_table => 'jmsuser.jms_qtt_map');
EXECUTE DBMS_AQADM.CREATE_QUEUE (Queue_name => 'jmsuser.jms_general_que',
    Queue_table => 'jmsuser.jms_qtt_general');

Rem
Rem Starting the queues and enable both enqueue and dequeue
Rem
EXECUTE DBMS_AQADM.START_QUEUE (Queue_name => 'jmsuser.jms_text_que');
EXECUTE DBMS_AQADM.START_QUEUE (Queue_name => 'jmsuser.jms_bytes_que');
EXECUTE DBMS_AQADM.START_QUEUE (Queue_name => 'jmsuser.jms_stream_que');
EXECUTE DBMS_AQADM.START_QUEUE (Queue_name => 'jmsuser.jms_map_que');
EXECUTE DBMS_AQADM.START_QUEUE (Queue_name => 'jmsuser.jms_general_que');

Rem The supporting utility used in the example to help display results in SQLPLUS
enviroment

Rem
Rem Display a RAW data in SQLPLUS
Rem
create or replace procedure display_raw(rdata raw)
IS
    pos                 pls_integer;
    length              pls_integer;
BEGIN
    pos := 1;
    length := UTL_RAW.LENGTH(rdata);

    WHILE pos <= length LOOP
      IF pos+20 > length+1 THEN
        dbms_output.put_line(UTL_RAW.SUBSTR(rdata, pos, length-pos+1));
      ELSE
        dbms_output.put_line(UTL_RAW.SUBSTR(rdata, pos, 20));
      END IF;
      pos := pos+20;
    END LOOP;
END display_raw;
/

show errors;

Rem
Rem Display a BLOB data in SQLPLUS
Rem
create or replace procedure display_blob(bdata blob)
IS
    pos                 pls_integer;
    length              pls_integer;
BEGIN
    length :=  dbms_lob.getlength(bdata);
    pos := 1;
    WHILE pos <= length LOOP
      display_raw(DBMS_LOB.SUBSTR(bdata, 2000, pos));
      pos := pos+2000;
    END LOOP;
END display_blob;
/

show errors;

Rem
Rem Display a VARCHAR data in SQLPLUS
Rem
create or replace procedure display_varchar(vdata varchar)
IS
    pos                 pls_integer;
    text_len            pls_integer;
BEGIN
    text_len :=  length(vdata);
    pos := 1;

    WHILE pos <= text_len LOOP
      IF pos+20 > text_len+1 THEN
        dbms_output.put_line(SUBSTR(vdata, pos, text_len-pos+1));
      ELSE
        dbms_output.put_line(SUBSTR(vdata, pos, 20));
      END IF;
      pos := pos+20;
    END LOOP;

END display_varchar;
/

show errors;

Rem
Rem Display a CLOB data in SQLPLUS
Rem
create or replace procedure display_clob(cdata clob)
IS
    pos                 pls_integer;
    length              pls_integer;
BEGIN
    length :=  dbms_lob.getlength(cdata);
    pos := 1;
    WHILE pos <= length LOOP
      display_varchar(DBMS_LOB.SUBSTR(cdata, 2000, pos));
      pos := pos+2000;
    END LOOP;
END display_clob;
/

show errors;

Rem
Rem Display a SYS.AQ$_JMS_EXCEPTION data in SQLPLUS
Rem
Rem When application receives an ORA-24197 error, It means the JAVA stored
Rem procedures has thrown some exceptions that could not be catergorized. The
Rem user can use GET_EXCEPTION procedure of SYS.AQ$_JMS_BYTES_MESSAGE,
Rem SYS.AQ$_JMS_STREAM_MESSAG or SYS.AQ$_JMS_MAP_MESSAGE
Rem to retrieve a SYS.AQ$_JMS_EXCEPTION object which contains more detailed
Rem information on this JAVA exception including the exception name, JAVA error
Rem message and stack trace.
Rem
Rem This utility function is to help display the SYS.AQ$_JMS_EXCEPTION object in
Rem SQLPLUS
Rem
create or replace procedure display_exp(exp SYS.AQ$_JMS_EXCEPTION)
IS
    pos1                pls_integer;
    pos2                pls_integer;
    text_data           varchar(2000);
BEGIN
    dbms_output.put_line('exception:'||exp.exp_name);
    dbms_output.put_line('err_msg:'||exp.err_msg);
    dbms_output.put_line('stack:'||length(exp.stack));
    pos1 := 1;
    LOOP
      pos2 := INSTR(exp.stack, chr(10), pos1);
      IF pos2 = 0 THEN
        pos2 := length(exp.stack)+1;
      END IF;

      dbms_output.put_line(SUBSTR(exp.stack, pos1, pos2-pos1));

      IF pos2 > length(exp.stack) THEN
        EXIT;
      END IF;

      pos1 := pos2+1;
    END LOOP;

END display_exp;
/

show errors;

EXIT;
```

Example 6-97 Setting Up the Examples

Example 6-96 performs the necessary setup for the JMS types examples. Copy and save it as `setup.sql`.

JMS BytesMessage Examples

This section includes examples that illustrate enqueuing and dequeuing of a JMS `BytesMessage`.

Example 6-98 shows how to use JMS type member functions with `DBMS_AQ` functions to populate and enqueue a JMS `BytesMessage` represented as `sys.aq$_jms_bytes_message` type in the database. This message later can be dequeued by a JAVA Oracle Java Message Service (Oracle JMS) client.

Example 6-99 illustrates how to use JMS type member functions with `DBMS_AQ` functions to dequeue and retrieve data from a JMS `BytesMessage` represented as `sys.aq$_jms_bytes_message` type in the database. This message might be enqueued by an Oracle JMS client.

Example 6-98 Populating and Enqueuing a BytesMessage

```
set echo offset verify offconnect sysDROP USER jmsuser CASCADE;ACCEPT password CHAR
PROMPT 'Enter the password for JMSUSER: ' HIDECREATE USER jmsuser IDENTIFIED BY
&password;GRANT DBA, AQ_ADMINISTRATOR_ROLE, AQ_USER_ROLE to jmsuser;GRANT EXECUTE ON
DBMS_AQADM TO jmsuser;GRANT EXECUTE ON DBMS_AQ TO jmsuser;GRANT EXECUTE ON DBMS_LOB TO
jmsuser;GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;connect jmsuser/&password

SET ECHO ON
set serveroutput on

DECLARE

    id                 pls_integer;
    agent              sys.aq$_agent := sys.aq$_agent(' ', null, 0);
    message            sys.aq$_jms_bytes_message;
    enqueue_options    dbms_aq.enqueue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
BEGIN

    -- Consturct a empty BytesMessage object
    message := sys.aq$_jms_bytes_message.construct;

    -- Shows how to set the JMS header
    message.set_replyto(agent);
    message.set_type('tkaqpet1');
    message.set_userid('jmsuser');
    message.set_appid('plsql_enq');
    message.set_groupid('st');
    message.set_groupseq(1);

    -- Shows how to set JMS user properties
    message.set_string_property('color', 'RED');
    message.set_int_property('year', 1999);
    message.set_float_property('price', 16999.99);
    message.set_long_property('mileage', 300000);
    message.set_boolean_property('import', True);
    message.set_byte_property('password', -127);

    -- Shows how to populate the message payload of aq$_jms_bytes_message
    -- Passing -1 reserve a new slot within the message store of
sys.aq$_jms_bytes_message.
    -- The maximum number of sys.aq$_jms_bytes_message type of messges to be operated at
    -- the same time within a session is 20. Calling clean_body function with parameter
-1
    -- might result a ORA-24199 error if the messages currently operated is already 20.
    -- The user is responsible to call clean or clean_all function to clean up message
store.
    id := message.clear_body(-1);

    -- Write data into the BytesMessage paylaod. These functions are analogy of JMS JAVA
api's.
    -- See the document for detail.

    -- Write a byte to the BytesMessage payload
    message.write_byte(id, 10);

    -- Write a RAW data as byte array to the BytesMessage payload
    message.write_bytes(id, UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')));

    -- Write a portion of the RAW data as byte array to BytesMessage payload
    -- Note the offset follows JAVA convention, starting from 0
    message.write_bytes(id, UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')), 0, 16);

    -- Write a char to the BytesMessage payload
    message.write_char(id, 'A');

    -- Write a double to the BytesMessage payload
    message.write_double(id, 9999.99);

    -- Write a float to the BytesMessage payload
    message.write_float(id, 99.99);

    -- Write a int to the BytesMessage payload
    message.write_int(id, 12345);

    -- Write a long to the BytesMessage payload
    message.write_long(id, 1234567);

    -- Write a short to the BytesMessage payload
    message.write_short(id, 123);

    -- Write a String to the BytesMessage payload,
    -- the String is encoded in UTF8 in the message payload
    message.write_utf(id, 'Hello World!');

    -- Flush the data from JAVA stored procedure (JServ) to PL/SQL side
    -- Without doing this, the PL/SQL message is still empty.
    message.flush(id);

    -- Use either clean_all or clean to clean up the message store when the user
    -- do not plan to do paylaod population on this message anymore
    sys.aq$_jms_bytes_message.clean_all();
    --message.clean(id);

    -- Enqueue this message into AQ queue using DBMS_AQ package
    dbms_aq.enqueue(queue_name => 'jmsuser.jms_bytes_que',
                    enqueue_options => enqueue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);
    EXCEPTION
    WHEN java_exp THEN
      dbms_output.put_line('exception information:');
      display_exp(sys.aq$_jms_stream_message.get_exception());

END;
/

commit;
```

Example 6-99 Dequeuing and Retrieving JMS BytesMessage Data

```
set echo off
set verify off

DROP USER jmsuser CASCADE;

ACCEPT password CHAR PROMPT 'Enter the password for JMSUSER: ' HIDE

CREATE USER jmsuser IDENTIFIED BY &password;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;
connect jmsuser/&password
set echo on
set serveroutput on size 20000

DECLARE

    id                 pls_integer;
    blob_data          blob;
    clob_data          clob;
    blob_len           pls_integer;
    message            sys.aq$_jms_bytes_message;
    agent              sys.aq$_agent;
    dequeue_options    dbms_aq.dequeue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);
    gdata              sys.aq$_jms_value;

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
BEGIN
    DBMS_OUTPUT.ENABLE (20000);

    -- Dequeue this message from AQ queue using DBMS_AQ package
    dbms_aq.dequeue(queue_name => 'jmsuser.jms_bytes_que',
                    dequeue_options => dequeue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);

    -- Retrieve the header
    agent := message.get_replyto;

    dbms_output.put_line('Type: ' || message.get_type ||
                         ' UserId: ' || message.get_userid ||
                         ' AppId: ' || message.get_appid ||
                         ' GroupId: ' || message.get_groupid ||
                         ' GroupSeq: ' || message.get_groupseq);
```

```
        -- Retrieve the user properties
        dbms_output.put_line('price: ' || message.get_float_property('price'));
        dbms_output.put_line('color: ' || message.get_string_property('color'));
        IF message.get_boolean_property('import') = TRUE THEN
           dbms_output.put_line('import: Yes' );
        ELSIF message.get_boolean_property('import') = FALSE THEN
           dbms_output.put_line('import: No' );
        END IF;
        dbms_output.put_line('year: ' || message.get_int_property('year'));
        dbms_output.put_line('mileage: ' || message.get_long_property('mileage'));
        dbms_output.put_line('password: ' || message.get_byte_property('password'));

    -- Shows how to retrieve the message payload of aq$_jms_bytes_message

    -- Prepare call, send the content in the PL/SQL aq$_jms_bytes_message object to
       -- Java stored procedure(Jserv) in the form of a byte array.
       -- Passing -1 reserves a new slot in msg store of sys.aq$_jms_bytes_message.
       -- Max number of sys.aq$_jms_bytes_message type of messges to be operated at
       -- the same time in a session is 20. Call clean_body fn. with parameter -1
       -- might result in ORA-24199 error if messages operated on are already 20.
       -- You must call clean or clean_all function to clean up message store.
        id := message.prepare(-1);

    -- Read data from BytesMessage paylaod. These fns. are analogy of JMS Java
    -- API's. See the JMS Types chapter for detail.
        dbms_output.put_line('Payload:');

        -- read a byte from the BytesMessage payload
        dbms_output.put_line('read_byte:' || message.read_byte(id));

        -- read a byte array into a blob object from the BytesMessage payload
        dbms_output.put_line('read_bytes:');
        blob_len := message.read_bytes(id, blob_data, 272);
        display_blob(blob_data);

        -- read a char from the BytesMessage payload
        dbms_output.put_line('read_char:'|| message.read_char(id));

        -- read a double from the BytesMessage payload
        dbms_output.put_line('read_double:'|| message.read_double(id));

        -- read a float from the BytesMessage payload
        dbms_output.put_line('read_float:'|| message.read_float(id));

        -- read a int from the BytesMessage payload
        dbms_output.put_line('read_int:'|| message.read_int(id));

        -- read a long from the BytesMessage payload
        dbms_output.put_line('read_long:'|| message.read_long(id));

        -- read a short from the BytesMessage payload
        dbms_output.put_line('read_short:'|| message.read_short(id));

        -- read a String from the BytesMessage payload.
        -- the String is in UTF8 encoding in the message payload
        dbms_output.put_line('read_utf:');
        message.read_utf(id, clob_data);
        display_clob(clob_data);

        -- Use either clean_all or clean to clean up the message store when the user
        -- do not plan to do paylaod retrieving on this message anymore
        message.clean(id);
        -- sys.aq$_jms_bytes_message.clean_all();

        EXCEPTION
        WHEN java_exp THEN
          dbms_output.put_line('exception information:');
          display_exp(sys.aq$_jms_bytes_message.get_exception());

    END;
    /

    commit;
```

JMS StreamMessage Examples

This section includes examples that illustrate enqueuing and dequeuing of a JMS `StreamMessage`.

Example 6-100 shows how to use JMS type member functions with `DBMS_AQ` functions to populate and enqueue a JMS `StreamMessage` represented as `sys.aq$_jms_stream_message` type in the database. This message later can be dequeued by an Oracle JMS client.

Example 6-101 shows how to use JMS type member functions with `DBMS_AQ` functions to dequeue and retrieve data from a JMS `StreamMessage` represented as `sys.aq$_jms_stream_message` type in the database. This message might be enqueued by an Oracle JMS client.

Example 6-100 Populating and Enqueuing a JMS StreamMessage

```
set echo off
set verify off


DROP USER jmsuser CASCADE;

ACCEPT password CHAR PROMPT 'Enter the password for JMSUSER: ' HIDE

CREATE USER jmsuser IDENTIFIED BY &password;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;
connect jmsuser/&password
SET ECHO ON
set serveroutput on

DECLARE

    id                 pls_integer;
    agent              sys.aq$_agent := sys.aq$_agent(' ', null, 0);
    message            sys.aq$_jms_stream_message;
    enqueue_options    dbms_aq.enqueue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
BEGIN

    -- Consturct a empty StreamMessage object
    message := sys.aq$_jms_stream_message.construct;
    -- Shows how to set the JMS header
    message.set_replyto(agent);
    message.set_type('tkaqpet1');
    message.set_userid('jmsuser');
    message.set_appid('plsql_enq');
    message.set_groupid('st');
    message.set_groupseq(1);

    -- Shows how to set JMS user properties
    message.set_string_property('color', 'RED');
    message.set_int_property('year', 1999);
    message.set_float_property('price', 16999.99);
    message.set_long_property('mileage', 300000);
    message.set_boolean_property('import', True);
    message.set_byte_property('password', -127);

    -- Shows how to populate the message payload of aq$_jms_stream_message

    -- Passing -1 reserve a new slot within the message store of 
sys.aq$_jms_stream_message.
    -- The maximum number of sys.aq$_jms_stream_message type of messges to be operated at
    -- the same time within a session is 20. Calling clean_body function with parameter 
-1 
    -- might result a ORA-24199 error if the messages currently operated is already 20.
    -- The user is responsible to call clean or clean_all function to clean up message 
store.
    id := message.clear_body(-1);

    -- Write data into the message paylaod. These functions are analogy of JMS JAVA 
api's.
    -- See the document for detail.

    -- Write a byte to the StreamMessage payload
    message.write_byte(id, 10);

    -- Write a RAW data as byte array to the StreamMessage payload
    message.write_bytes(id, UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')));

    -- Write a portion of the RAW data as byte array to the StreamMessage payload
    -- Note the offset follows JAVA convention, starting from 0
    message.write_bytes(id, UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')), 0, 16);

    -- Write a char to the StreamMessage payload
    message.write_char(id, 'A');

    -- Write a double to the StreamMessage payload
    message.write_double(id, 9999.99);

    -- Write a float to the StreamMessage payload
    message.write_float(id, 99.99);

    -- Write a int to the StreamMessage payload
    message.write_int(id, 12345);

    -- Write a long to the StreamMessage payload
    message.write_long(id, 1234567);

    -- Write a short to the StreamMessage payload
    message.write_short(id, 123);

    -- Write a String to the StreamMessage payload
    message.write_string(id, 'Hello World!');

    -- Flush the data from JAVA stored procedure (JServ) to PL/SQL side
    -- Without doing this, the PL/SQL message is still empty.
    message.flush(id);

    -- Use either clean_all or clean to clean up the message store when the user
    -- do not plan to do paylaod population on this message anymore
    sys.aq$_jms_stream_message.clean_all();
    --message.clean(id);

    -- Enqueue this message into AQ queue using DBMS_AQ package
    dbms_aq.enqueue(queue_name => 'jmsuser.jms_stream_que',
                    enqueue_options => enqueue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);


    EXCEPTION
    WHEN java_exp THEN
      dbms_output.put_line('exception information:');
      display_exp(sys.aq$_jms_stream_message.get_exception());

END;
/

commit;
```

Example 6-101   Dequeuing and Retrieving Data From a JMS StreamMessage

```
set echo off
set verify off


DROP USER jmsuser CASCADE;

ACCEPT password CHAR PROMPT 'Enter the password for JMSUSER: ' HIDE

CREATE USER jmsuser IDENTIFIED BY &password;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;
connect jmsuser/&password
set echo on
set serveroutput on

DECLARE

    id                 pls_integer;
    blob_data          blob;
    clob_data          clob;
    message            sys.aq$_jms_stream_message;
    agent              sys.aq$_agent;
    dequeue_options    dbms_aq.dequeue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);
    gdata              sys.aq$_jms_value;

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
```

```
BEGIN
    DBMS_OUTPUT.ENABLE (20000);

    -- Dequeue this message from AQ queue using DBMS_AQ package
    dbms_aq.dequeue(queue_name => 'jmsuser.jms_stream_que',
                    dequeue_options => dequeue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);

    -- Retrieve the header
    agent := message.get_replyto;

    dbms_output.put_line('Type: ' || message.get_type ||
                         ' UserId: ' || message.get_userid ||
                         ' AppId: ' || message.get_appid ||
                         ' GroupId: ' || message.get_groupid ||
                         ' GroupSeq: ' || message.get_groupseq);

    -- Retrieve the user properties
    dbms_output.put_line('price: ' || message.get_float_property('price'));
    dbms_output.put_line('color: ' || message.get_string_property('color'));
    IF message.get_boolean_property('import') = TRUE THEN
       dbms_output.put_line('import: Yes' );
    ELSIF message.get_boolean_property('import') = FALSE THEN
       dbms_output.put_line('import: No' );
    END IF;
    dbms_output.put_line('year: ' || message.get_int_property('year'));
    dbms_output.put_line('mileage: ' || message.get_long_property('mileage'));
    dbms_output.put_line('password: ' || message.get_byte_property('password'));

    -- Shows how to retrieve the message payload of aq$_jms_stream_message

    -- The prepare call send the content in the PL/SQL aq$_jms_stream_message object to
    -- JAVA stored procedure(Jserv) in the form of byte array.
    -- Passing -1 reserve a new slot within the message store of
sys.aq$_jms_stream_message.
    -- The maximum number of sys.aq$_jms_stream_message type of messges to be operated at
    -- the same time within a session is 20. Calling clean_body function with parameter
-1
    -- might result a ORA-24199 error if the messages currently operated is already 20.
    -- The user is responsible to call clean or clean_all function to clean up message
store.
    id := message.prepare(-1);


    -- Assume the users know the types of data in the StreamMessage payload.
    -- The user can use the specific read function corresponding with the data type.
    -- These functions are analogy of JMS JAVA api's. See the document for detail.
    dbms_output.put_line('Retrieve payload by Type:');

    -- Read a byte from the StreamMessage payload
    dbms_output.put_line('read_byte:' || message.read_byte(id));

    -- Read a byte array into a blob object from the StreamMessage payload
    dbms_output.put_line('read_bytes:');
    message.read_bytes(id, blob_data);
    display_blob(blob_data);

    -- Read another byte array into a blob object from the StreamMessage payload
    dbms_output.put_line('read_bytes:');
    message.read_bytes(id, blob_data);
    display_blob(blob_data);

    -- Read a char from the StreamMessage payload
    dbms_output.put_line('read_char:'|| message.read_char(id));

    -- Read a double from the StreamMessage payload
    dbms_output.put_line('read_double:'|| message.read_double(id));

    -- Read a float from the StreamMessage payload
    dbms_output.put_line('read_float:'|| message.read_float(id));

    -- Read a int from the StreamMessage payload
    dbms_output.put_line('read_int:'|| message.read_int(id));

    -- Read a long from the StreamMessage payload
    dbms_output.put_line('read_long:'|| message.read_long(id));

    -- Read a short from the StreamMessage payload
    dbms_output.put_line('read_short:'|| message.read_short(id));

    -- Read a String into a clob data from the StreamMessage payload
    dbms_output.put_line('read_string:');
    message.read_string(id, clob_data);
    display_clob(clob_data);


    -- Assume the users do not know the types of data in the StreamMessage payload.
    -- The user can use read_object method to read the data into a sys.aq$_jms_value
object
    -- These functions are analogy of JMS JAVA api's. See the document for detail.

    -- Reset the stream pointer to the begining of the message so that we can read
throught
    -- the message payload again.
    message.reset(id);

    LOOP
      message.read_object(id, gdata);
      IF gdata IS NULL THEN
        EXIT;
      END IF;

      CASE gdata.type
        WHEN sys.dbms_jms_plsql.DATA_TYPE_BYTE      THEN
                 dbms_output.put_line('read_object/byte:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_SHORT     THEN
                 dbms_output.put_line('read_object/short:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_INTEGER   THEN
                 dbms_output.put_line('read_object/int:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_LONG      THEN
                 dbms_output.put_line('read_object/long:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_FLOAT     THEN
                 dbms_output.put_line('read_object/float:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_DOUBLE    THEN
                 dbms_output.put_line('read_object/double:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_BOOLEAN   THEN
                 dbms_output.put_line('read_object/boolean:' || gdata.num_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_CHARACTER THEN
                 dbms_output.put_line('read_object/char:' || gdata.char_val);
        WHEN sys.dbms_jms_plsql.DATA_TYPE_STRING    THEN
                 dbms_output.put_line('read_object/string:');
                 display_clob(gdata.text_val);
```

```
            WHEN sys.dbms_jms_plsql.DATA_TYPE_BYTES     THEN
                     dbms_output.put_line('read_object/bytes:');
                     display_blob(gdata.bytes_val);
            ELSE dbms_output.put_line('No such data type');
          END CASE;

        END LOOP;

        -- Use either clean_all or clean to clean up the message store when the user
        -- do not plan to do paylaod retrieving on this message anymore
        message.clean(id);
        -- sys.aq$_jms_stream_message.clean_all();

        EXCEPTION
        WHEN java_exp THEN
          dbms_output.put_line('exception information:');
          display_exp(sys.aq$_jms_stream_message.get_exception());

    END;
    /

    commit;
```

JMS MapMessage Examples

This section includes examples that illustrate enqueuing and dequeuing of a JMS `MapMessage`.

Example 6-102 shows how to use JMS type member functions with `DBMS_AQ` functions to populate and enqueue a JMS `MapMessage` represented as `sys.aq$_jms_map_message` type in the database. This message later can be dequeued by an Oracle JMS client.

Example 6-103 illustrates how to use JMS type member functions with `DBMS_AQ` functions to dequeue and retrieve data from a JMS `MapMessage` represented as `sys.aq$_jms_map_message` type in the database. This message can be enqueued by an Oracle JMS client.

Example 6-102   Populating and Enqueuing a JMS MapMessage

```
set echo off
set verify off


DROP USER jmsuser CASCADE;

ACCEPT password CHAR PROMPT 'Enter the password for JMSUSER: ' HIDE

CREATE USER jmsuser IDENTIFIED BY &password;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;
connect jmsuser/&password

SET ECHO ON
set serveroutput on

DECLARE

    id                 pls_integer;
    agent              sys.aq$_agent := sys.aq$_agent(' ', null, 0);
    message            sys.aq$_jms_map_message;
    enqueue_options    dbms_aq.enqueue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
BEGIN

    -- Consturct a empty map message object
    message := sys.aq$_jms_map_message.construct;

    -- Shows how to set the JMS header
    message.set_replyto(agent);
    message.set_type('tkaqpet1');
    message.set_userid('jmsuser');
    message.set_appid('plsql_enq');
    message.set_groupid('st');
    message.set_groupseq(1);

    -- Shows how to set JMS user properties
    message.set_string_property('color', 'RED');
    message.set_int_property('year', 1999);
    message.set_float_property('price', 16999.99);
    message.set_long_property('mileage', 300000);
    message.set_boolean_property('import', True);
    message.set_byte_property('password', -127);

    -- Shows how to populate the message payload of aq$_jms_map_message

    -- Passing -1 reserve a new slot within the message store of sys.aq$_jms_map_message.
    -- The maximum number of sys.aq$_jms_map_message type of messges to be operated at
    -- the same time within a session is 20. Calling clean_body function with parameter
-1
    -- might result a ORA-24199 error if the messages currently operated is already 20.
    -- The user is responsible to call clean or clean_all function to clean up message
store.
    id := message.clear_body(-1);

    -- Write data into the message paylaod. These functions are analogy of JMS JAVA
api's.
    -- See the document for detail.

    -- Set a byte entry in map message payload
    message.set_byte(id, 'BYTE', 10);

    -- Set a byte array entry using RAW data in map message payload
    message.set_bytes(id, 'BYTES', UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')));

    -- Set a byte array entry using only a portion of the RAW data in map message payload
    -- Note the offset follows JAVA convention, starting from 0
    message.set_bytes(id, 'BYTES_PART', UTL_RAW.XRANGE(HEXTORAW('00'), HEXTORAW('FF')),
0, 16);

    -- Set a char entry in map message payload
    message.set_char(id, 'CHAR', 'A');

    -- Set a double entry in map message payload
    message.set_double(id, 'DOUBLE', 9999.99);

    -- Set a float entry in map message payload
    message.set_float(id, 'FLOAT', 99.99);

    -- Set a int entry in map message payload
    message.set_int(id, 'INT', 12345);

    -- Set a long entry in map message payload
    message.set_long(id, 'LONG', 1234567);

    -- Set a short entry in map message payload
    message.set_short(id, 'SHORT', 123);

    -- Set a String entry in map message payload
    message.set_string(id, 'STRING', 'Hello World!');

    -- Flush the data from JAVA stored procedure (JServ) to PL/SQL side
    -- Without doing this, the PL/SQL message is still empty.
    message.flush(id);

    -- Use either clean_all or clean to clean up the message store when the user
    -- do not plan to do paylaod population on this message anymore
    sys.aq$_jms_map_message.clean_all();
    --message.clean(id);

    -- Enqueue this message into AQ queue using DBMS_AQ package
    dbms_aq.enqueue(queue_name => 'jmsuser.jms_map_que',
                    enqueue_options => enqueue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);

END;
/

commit;
```

Example 6-103 Dequeuing and Retrieving Data From a JMS MapMessage

```
set echo off
set verify off


DROP USER jmsuser CASCADE;

ACCEPT password CHAR PROMPT 'Enter the password for JMSUSER: ' HIDE

CREATE USER jmsuser IDENTIFIED BY &password;
GRANT EXECUTE ON DBMS_AQADM TO jmsuser;
GRANT EXECUTE ON DBMS_AQ TO jmsuser;
GRANT EXECUTE ON DBMS_LOB TO jmsuser;
GRANT EXECUTE ON DBMS_JMS_PLSQL TO jmsuser;
connect jmsuser/&password

set echo on
set serveroutput on

DECLARE

    id                 pls_integer;
    blob_data          blob;
    clob_data          clob;
    message            sys.aq$_jms_map_message;
    agent              sys.aq$_agent;
    dequeue_options    dbms_aq.dequeue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid              raw(16);
    name_arr           sys.aq$_jms_namearray;
    gdata              sys.aq$_jms_value;

    java_exp           exception;
    pragma EXCEPTION_INIT(java_exp, -24197);
BEGIN
    DBMS_OUTPUT.ENABLE (20000);

    -- Dequeue this message from AQ queue using DBMS_AQ package
    dbms_aq.dequeue(queue_name => 'jmsuser.jms_map_que',
                    dequeue_options => dequeue_options,
                    message_properties => message_properties,
                    payload => message,
                    msgid => msgid);

    -- Retrieve the header
    agent := message.get_replyto;

    dbms_output.put_line('Type: ' || message.get_type ||
                         ' UserId: ' || message.get_userid ||
                         ' AppId: ' || message.get_appid ||
                         ' GroupId: ' || message.get_groupid ||
                         ' GroupSeq: ' || message.get_groupseq);

    -- Retrieve the user properties
    dbms_output.put_line('price: ' || message.get_float_property('price'));
    dbms_output.put_line('color: ' || message.get_string_property('color'));
    IF message.get_boolean_property('import') = TRUE THEN
       dbms_output.put_line('import: Yes' );
    ELSIF message.get_boolean_property('import') = FALSE THEN
       dbms_output.put_line('import: No' );
    END IF;
    dbms_output.put_line('year: ' || message.get_int_property('year'));
    dbms_output.put_line('mileage: ' || message.get_long_property('mileage'));
    dbms_output.put_line('password: ' || message.get_byte_property('password'));


    -- Shows how to retrieve the message payload of aq$_jms_map_message

    -- 'Prepare' sends the content in the PL/SQL aq$_jms_map_message object to
    -- Java stored procedure(Jserv) in the form of byte array.
    -- Passing -1 reserve a new slot within the message store of
    -- sys.aq$_jms_map_message. The maximum number of sys.aq$_jms_map_message
    -- type of messges to be operated at the same time within a session is 20.
    -- Calling clean_body function with parameter -1
    -- might result a ORA-24199 error if the messages currently operated is
    -- already 20. The user is responsible to call clean or clean_all function
    -- to clean up message store.
    id := message.prepare(-1);

    -- Assume the users know the names and types in the map message payload.
    -- The user can use names to get the corresponsing values.
    -- These functions are analogous to JMS Java API's. See JMS Types chapter
    -- for detail.
    dbms_output.put_line('Retrieve payload by Name:');

    -- Get a byte entry from the map message payload
    dbms_output.put_line('get_byte:' || message.get_byte(id, 'BYTE'));

    -- Get a byte array entry from the map message payload
    dbms_output.put_line('get_bytes:');
    message.get_bytes(id, 'BYTES', blob_data);
```

```
display_blob(blob_data);

-- Get another byte array entry from the map message payload
dbms_output.put_line('get_bytes:');
message.get_bytes(id, 'BYTES_PART', blob_data);
display_blob(blob_data);

-- Get a char entry from the map message payload
dbms_output.put_line('get_char:'|| message.get_char(id, 'CHAR'));

-- get a double entry from the map message payload
dbms_output.put_line('get_double:'|| message.get_double(id, 'DOUBLE'));

-- Get a float entry from the map message payload
dbms_output.put_line('get_float:'|| message.get_float(id, 'FLOAT'));

-- Get a int entry from the map message payload
dbms_output.put_line('get_int:'|| message.get_int(id, 'INT'));

-- Get a long entry from the map message payload
dbms_output.put_line('get_long:'|| message.get_long(id, 'LONG'));

-- Get a short entry from the map message payload
dbms_output.put_line('get_short:'|| message.get_short(id, 'SHORT'));

-- Get a String entry from the map message payload
dbms_output.put_line('get_string:');
message.get_string(id, 'STRING', clob_data);
display_clob(clob_data);

-- Assume users do not know names and types in map message payload.
-- User can first retrieve the name array containing all names in the
-- payload and iterate through the name list and get the corresponding
-- value. These functions are analogous to JMS Java API's.
-- See JMS Type chapter for detail.
dbms_output.put_line('Retrieve payload by iteration:');

-- Get the name array from the map message payload
name_arr := message.get_names(id);

-- Iterate through the name array to retrieve the value for each of the name.
FOR i IN name_arr.FIRST..name_arr.LAST LOOP

-- Test if a name exist in the map message payload
-- (It is not necessary in this case, just a demostration on how to use it)
  IF message.item_exists(id, name_arr(i)) THEN
    dbms_output.put_line('item exists:'||name_arr(i));

-- Because we do not know the type of entry, we must use sys.aq$_jms_value
-- type object for the data returned
   message.get_object(id, name_arr(i), gdata);
   IF gdata IS NOT NULL THEN
    CASE gdata.type
    WHEN   sys.dbms_jms_plsql.DATA_TYPE_BYTE
      THEN dbms_output.put_line('get_object/byte:' || gdata.num_val);
    WHEN   sys.dbms_jms_plsql.DATA_TYPE_SHORT
      THEN dbms_output.put_line('get_object/short:' || gdata.num_val);
    WHEN   sys.dbms_jms_plsql.DATA_TYPE_INTEGER
      THEN dbms_output.put_line('get_object/int:' || gdata.num_val);
    WHEN   sys.dbms_jms_plsql.DATA_TYPE_LONG
      THEN dbms_output.put_line('get_object/long:' || gdata.num_val);
    WHEN   sys.dbms_jms_plsql.DATA_TYPE_FLOAT
```

```
              THEN dbms_output.put_line('get_object/float:' || gdata.num_val);
            WHEN   sys.dbms_jms_plsql.DATA_TYPE_DOUBLE
              THEN dbms_output.put_line('get_object/double:' || gdata.num_val);
            WHEN   sys.dbms_jms_plsql.DATA_TYPE_BOOLEAN
              THEN dbms_output.put_line('get_object/boolean:' || gdata.num_val);
            WHEN   sys.dbms_jms_plsql.DATA_TYPE_CHARACTER
              THEN dbms_output.put_line('get_object/char:' || gdata.char_val);
            WHEN   sys.dbms_jms_plsql.DATA_TYPE_STRING
              THEN dbms_output.put_line('get_object/string:');
                   display_clob(gdata.text_val);
            WHEN sys.dbms_jms_plsql.DATA_TYPE_BYTES
            THEN
                 dbms_output.put_line('get_object/bytes:');
                 display_blob(gdata.bytes_val);
              ELSE dbms_output.put_line('No such data type');
              END CASE;
            END IF;
          ELSE
            dbms_output.put_line('item not exists:'||name_arr(i));
          END IF;

        END LOOP;


        -- Use either clean_all or clean to clean up the message store when the user
        -- do not plan to do paylaod population on this message anymore
        message.clean(id);
        -- sys.aq$_jms_map_message.clean_all();

        EXCEPTION
        WHEN java_exp THEN
          dbms_output.put_line('exception information:');
          display_exp(sys.aq$_jms_stream_message.get_exception());

    END;
    /

    commit;
```

More Oracle Database Advanced Queuing JMS Examples

The sample program in Example 6-104 enqueues a large `TextMessage` (along with JMS user properties) in an Oracle Database Advanced Queuing queue created through the Oracle JMS administrative interfaces to hold JMS `TEXT` messages. Both the `TextMessage` and `BytesMessage` enqueued in this example can be dequeued using Oracle JMS clients.

The sample program in Example 6-105 enqueues a large `BytesMessage`.

Example 6-104   Enqueuing a Large TextMessage

```
DECLARE

    text        varchar2(32767);
    agent       sys.aq$_agent   := sys.aq$_agent(' ', null, 0);
    message     sys.aq$_jms_text_message;

    enqueue_options    dbms_aq.enqueue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid               raw(16);

BEGIN
```

```
    message := sys.aq$_jms_text_message.construct;

    message.set_replyto(agent);
    message.set_type('tkaqpet2');
    message.set_userid('jmsuser');
    message.set_appid('plsql_enq');
    message.set_groupid('st');
    message.set_groupseq(1);

    message.set_boolean_property('import', True);
    message.set_string_property('color', 'RED');
    message.set_short_property('year', 1999);
    message.set_long_property('mileage', 300000);
    message.set_double_property('price', 16999.99);
    message.set_byte_property('password', 127);

    FOR i IN 1..500 LOOP
        text := CONCAT (text, '1234567890');
    END LOOP;

    message.set_text(text);

    dbms_aq.enqueue(queue_name => 'jmsuser.jms_text_t1',
                       enqueue_options => enqueue_options,
                       message_properties => message_properties,
                       payload => message,
                       msgid => msgid);

END;
```

Example 6-105 Enqueuing a Large BytesMessage

```
DECLARE

    text        VARCHAR2(32767);
    bytes       RAW(32767);
    agent       sys.aq$_agent   := sys.aq$_agent(' ', null, 0);
    message     sys.aq$_jms_bytes_message;
    body        BLOB;
    position    INT;

    enqueue_options    dbms_aq.enqueue_options_t;
    message_properties dbms_aq.message_properties_t;
    msgid raw(16);

BEGIN

    message := sys.aq$_jms_bytes_message.construct;

    message.set_replyto(agent);
    message.set_type('tkaqper4');
    message.set_userid('jmsuser');
    message.set_appid('plsql_enq_raw');
    message.set_groupid('st');
    message.set_groupseq(1);

    message.set_boolean_property('import', True);
    message.set_string_property('color', 'RED');
    message.set_short_property('year', 1999);
    message.set_long_property('mileage', 300000);
    message.set_double_property('price', 16999.99);
```

```
--  prepare a huge payload into a blob

    FOR i IN 1..1000 LOOP
        text := CONCAT (text, '0123456789ABCDEF');
    END LOOP;

    bytes := HEXTORAW(text);

    dbms_lob.createtemporary(lob_loc => body, cache => TRUE);
    dbms_lob.open (body, DBMS_LOB.LOB_READWRITE);
    position := 1 ;
    FOR i IN 1..10 LOOP
        dbms_lob.write ( lob_loc => body,
                amount => FLOOR((LENGTH(bytes)+1)/2),
                offset => position,
                buffer => bytes);
        position := position + FLOOR((LENGTH(bytes)+1)/2) ;
    END LOOP;

-- end of the preparation

    message.set_bytes(body);
    dbms_aq.enqueue(queue_name => 'jmsuser.jms_bytes_t1',
                       enqueue_options => enqueue_options,
                       message_properties => message_properties,
                       payload => message,
                       msgid => msgid);

    dbms_lob.freetemporary(lob_loc => body);
END;
```