# Java Message Service Concepts

This chapter provides an introduction to the Java Message Service (JMS) API, a Java API that allows applications to create, send, receive, and read messages using reliable, asynchronous, loosely coupled communication.

# It covers the following topics:

- Overview of the JMS API
- Basic JMS API Concepts
- The JMS API Programming Model
- Creating Robust JMS Applications
- Using the JMS API in Java EE Applications
- Further Information about JMS

# Overview of the JMS API

This overview of the JMS API answers the following questions.

- What Is Messaging?
- What Is the JMS API?
- When Can You Use the JMS API?
- How Does the JMS API Work with the Java EE Platform?

# What Is Messaging?

Messaging is a method of communication between software components or applications.

A messaging system is a peer-to-peer facility:

A messaging client can send messages to, and receive messages from, any other client.

Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is loosely coupled.

A component sends a message to a destination, and the recipient can retrieve the message from the destination.

However, the sender and the receiver do not have to be available at the same time in order to communicate.

In fact, the sender does not need to know anything about the receiver; nor does the receiver need to know anything about the sender.

The sender and the receiver need to know only which message format and which destination to use.

In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people.

# Messaging is used for communication between software applications or software components.

# What Is the JMS API?

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.

Designed by Sun and several partner companies, the JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications.

It also strives to maximize the portability of JMS applications across JMS providers in the same messaging domain.

# The JMS API enables communication that is not only loosely coupled but also

- **Asynchronous:** A JMS provider can deliver messages to a client as they arrive; a client does not have to request messages in order to receive them.

- **Reliable:** The JMS API can ensure that a message is delivered once and only once.

Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The JMS specification was first published in August 1998.

The latest version is Version 1.1, which was released in April 2002.

You can download a copy of the specification from the JMS web site:

http://www.oracle.com/technetwork/java/index-jsp-142945.html.

# When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as remote procedure call (RPC), under the following circumstances.

- The provider wants the components not to depend on information about other components' interfaces, so that components can be easily replaced.

- The provider wants the application to run whether or not all components are up and running simultaneously.

- The application business model allows a component to send information to another and to continue to operate without receiving an immediate response.

For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like these:

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so that the factory can make more cars.

- The factory component can send a message to the parts components so that the factory can assemble the parts it needs.

- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.

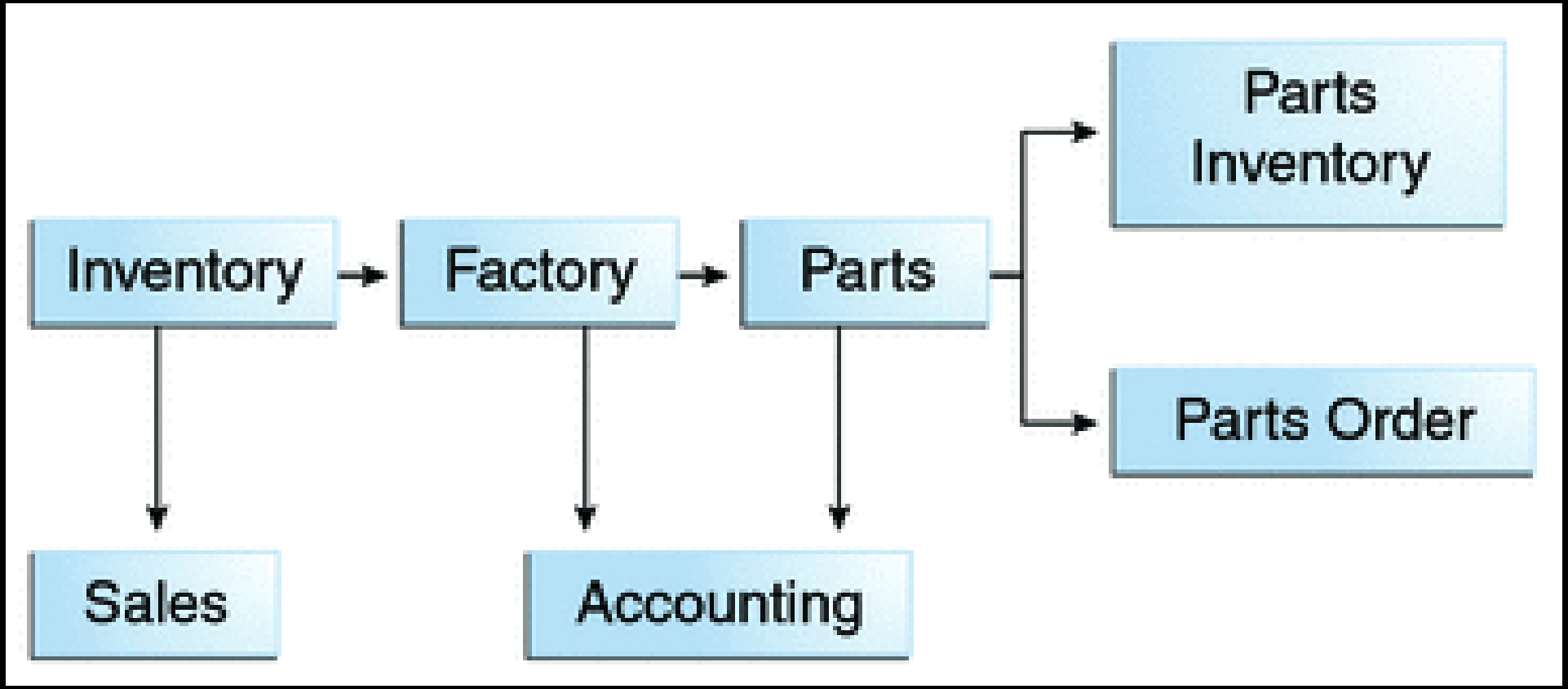
- Both the factory and the parts components can send messages to the accounting component to update their budget numbers.

- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources.

Figure 45-1 illustrates how this simple example might work.

Figure 45-1 Messaging in an Enterprise Application




**Manufacturing is only one example of how an enterprise can use the JMS API.**

Retail applications, financial services applications, health services applications, and many others can make use of messaging.

# How Does the JMS API Work with the Java EE Platform?

When the JMS API was introduced in 1998, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems, such as MQSeries from IBM.

Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise.

Beginning with the 1.3 release of the Java EE platform, the JMS API has been an integral part of the platform, and application developers can use messaging with Java EE components.

# The JMS API in the Java EE platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message.

  Application clients can in addition receive JMS messages asynchronously.

(Applets, however, are not required to support the JMS API.)

- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages.

A JMS provider can optionally implement concurrent processing of messages by message-driven beans.

- Message send and receive operations can participate in distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging.

A developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events.

The Java EE platform, moreover, enhances the JMS API by providing support for distributed transactions and allowing for the concurrent consumption of messages.

For more information, see the Enterprise JavaBeans specification, v3.1.

The JMS provider can be integrated with the application server using the Java EE Connector architecture.

You access the JMS provider through a resource adapter.

This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers.

For more information, see the Java EE Connector architecture specification, v1.6.

# Basic JMS API Concepts

This section introduces the most basic JMS API concepts, the ones you must know to get started writing simple application clients that use the JMS API.

The next section introduces the JMS API programming model.

Later sections cover more advanced concepts, including the ones you need to write applications that use message-driven beans.

# JMS API Architecture

A JMS application is composed of the following parts.

- A JMS provider is a messaging system that implements the JMS interfaces and provides administrative and control features.

An implementation of the Java EE platform includes a JMS provider.

- JMS clients are the programs or components, written in the Java programming language, that produce and consume messages.

Any Java EE application component can act as a JMS client.

- Messages are the objects that communicate information between JMS clients.

- Administered objects are preconfigured JMS objects created by an administrator for the use of clients.

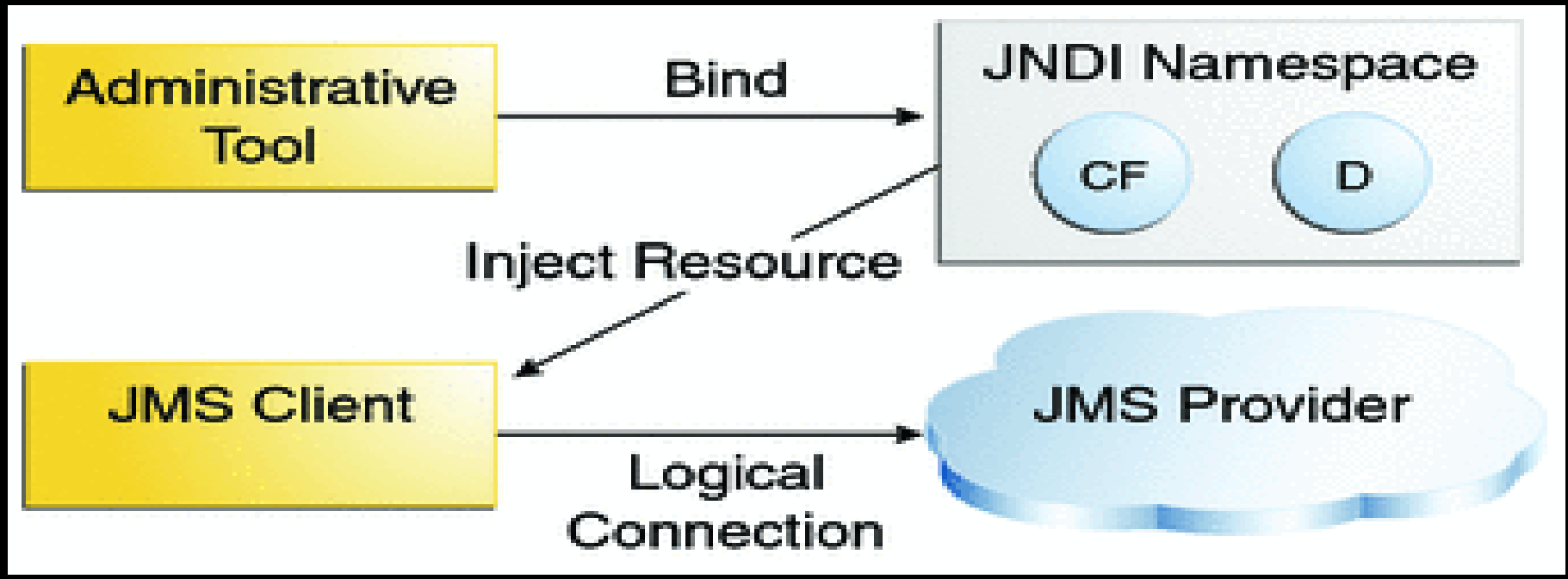
The two kinds of JMS administered objects are destinations and connection factories, which are described in JMS Administered Objects.

Figure 45-2 illustrates the way these parts interact.

Administrative tools allow you to bind destinations and connection factories into a JNDI namespace.

A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same objects through the JMS provider.

Figure 45-2 JMS API Architecture



# Messaging Domains

Before the JMS API existed, most messaging products supported either the point-to-point or the publish/subscribe approach to messaging.

The JMS specification provides a separate domain for each approach and defines compliance for each domain.

A stand-alone JMS provider can implement one or both domains. A Java EE provider must implement both domains.

In fact, most implementations of the JMS API support both the point-to-point and the publish/subscribe domains, and some JMS clients combine the use of both domains in a single application.

In this way, the JMS API has extended the power and flexibility of messaging products.

The JMS specification goes one step further:

It provides common interfaces that enable you to use the JMS API in a way that is not specific to either domain.

The following subsections describe the two messaging domains and then describe the use of the common interfaces.

# *Point-to-Point Messaging Domain*

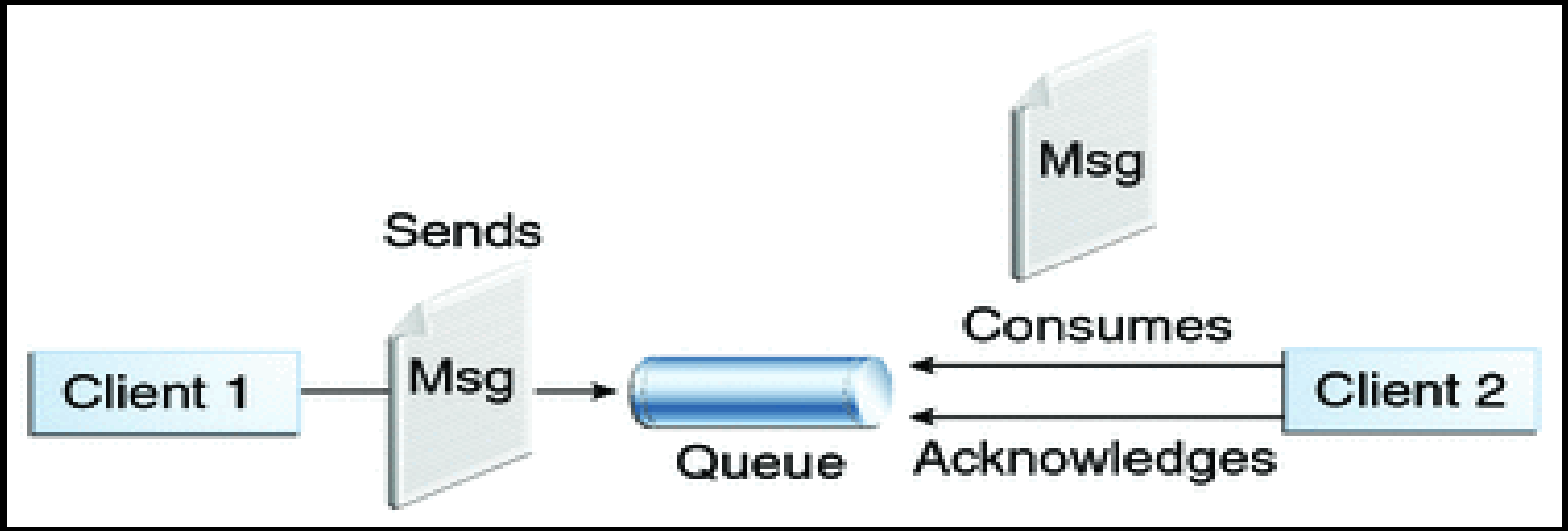
A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers.

Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages.

Queues retain all messages sent to them until the messages are consumed or until the messages expire.

PTP messaging has the following characteristics and is illustrated in Figure 45-3.

Figure 45-3 Point-to-Point Messaging




- Each message has only one consumer.
- A sender and a receiver of a message have no timing dependencies.

The receiver can fetch the message whether or not it was running when the client sent the message.

. The receiver acknowledges the successful processing of a message.

Use PTP messaging when every message you send must be processed successfully by one consumer.

# *Publish/Subscribe Messaging Domain*

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic, which functions somewhat like a bulletin board.

Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy.

The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers.

Topics retain messages only as long as it takes to distribute them to current subscribers.

Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.

- Publishers and subscribers have a timing dependency.

  A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.

The JMS API relaxes this timing dependency to some extent by allowing subscribers to create durable subscriptions, which receive messages sent while the subscribers are not active.

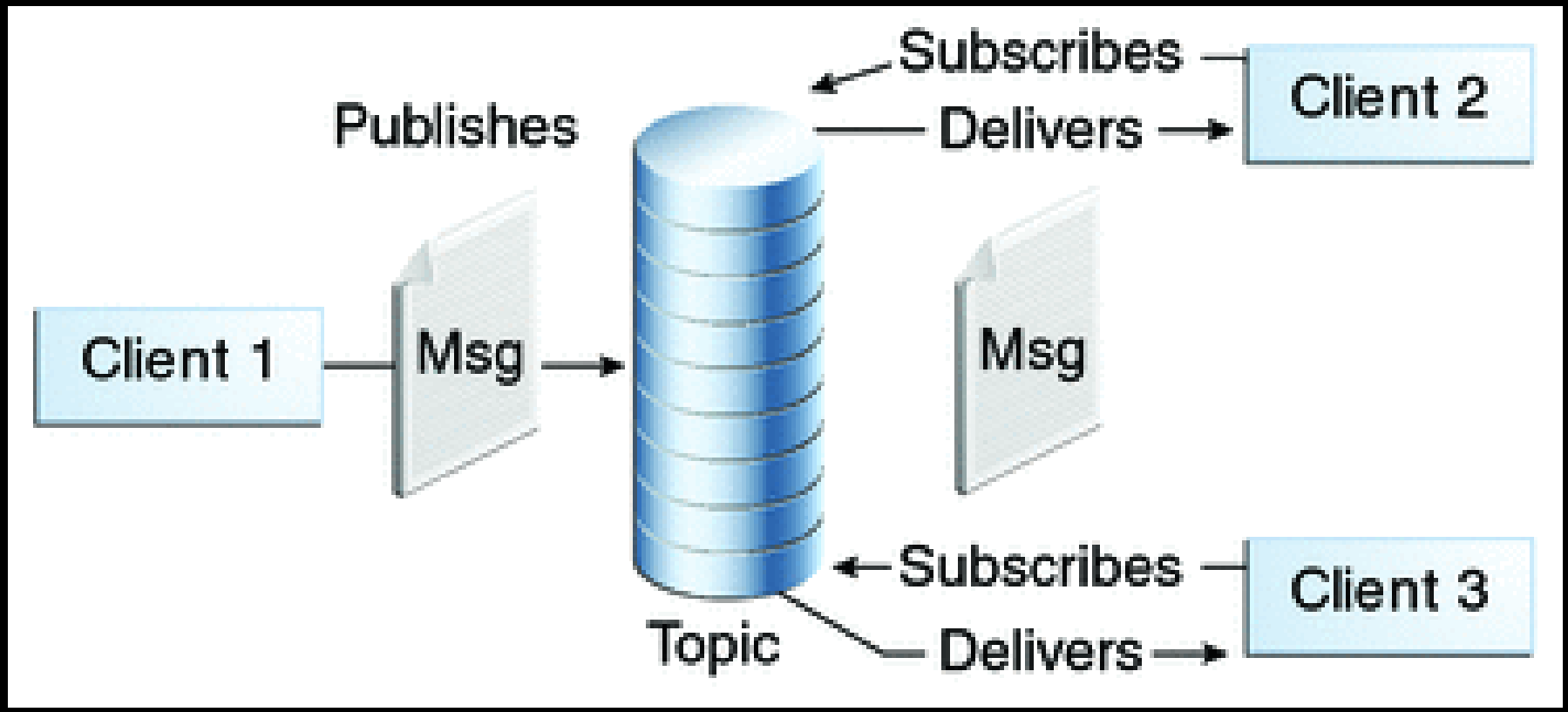
Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

For more information about durable subscriptions, see Creating Durable Subscriptions.

Use pub/sub messaging when each message can be processed by zero, one, or many consumers.

Figure 45-4 illustrates pub/sub messaging.

Figure 45-4 Publish/Subscribe Messaging



# *Programming with the Common Interfaces*

**Version 1.1 of the JMS API allows you to use the same code to send and receive messages under either the PTP or the pub/sub domain.**

**The destinations that you use remain domain-specific, and the behavior of the application will depend in part on whether you are using a queue or a topic.**

However, the code itself can be common to both domains, making your applications flexible and reusable.

This tutorial describes and illustrates these common interfaces.

# Message Consumption

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message.

However, the JMS specification uses this term in a more precise sense.

# Messages can be consumed in either of two ways:

- **Synchronously:** A subscriber or a receiver explicitly fetches the message from the destination by calling the `receive` method.

  The `receive` method can block until a message arrives or can time out if a message does not arrive within a specified time limit.

- **Asynchronously: A client can register a message listener with a consumer.**

  **A message listener is similar to an event listener.**

  **Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage` method, which acts on the contents of the message.**

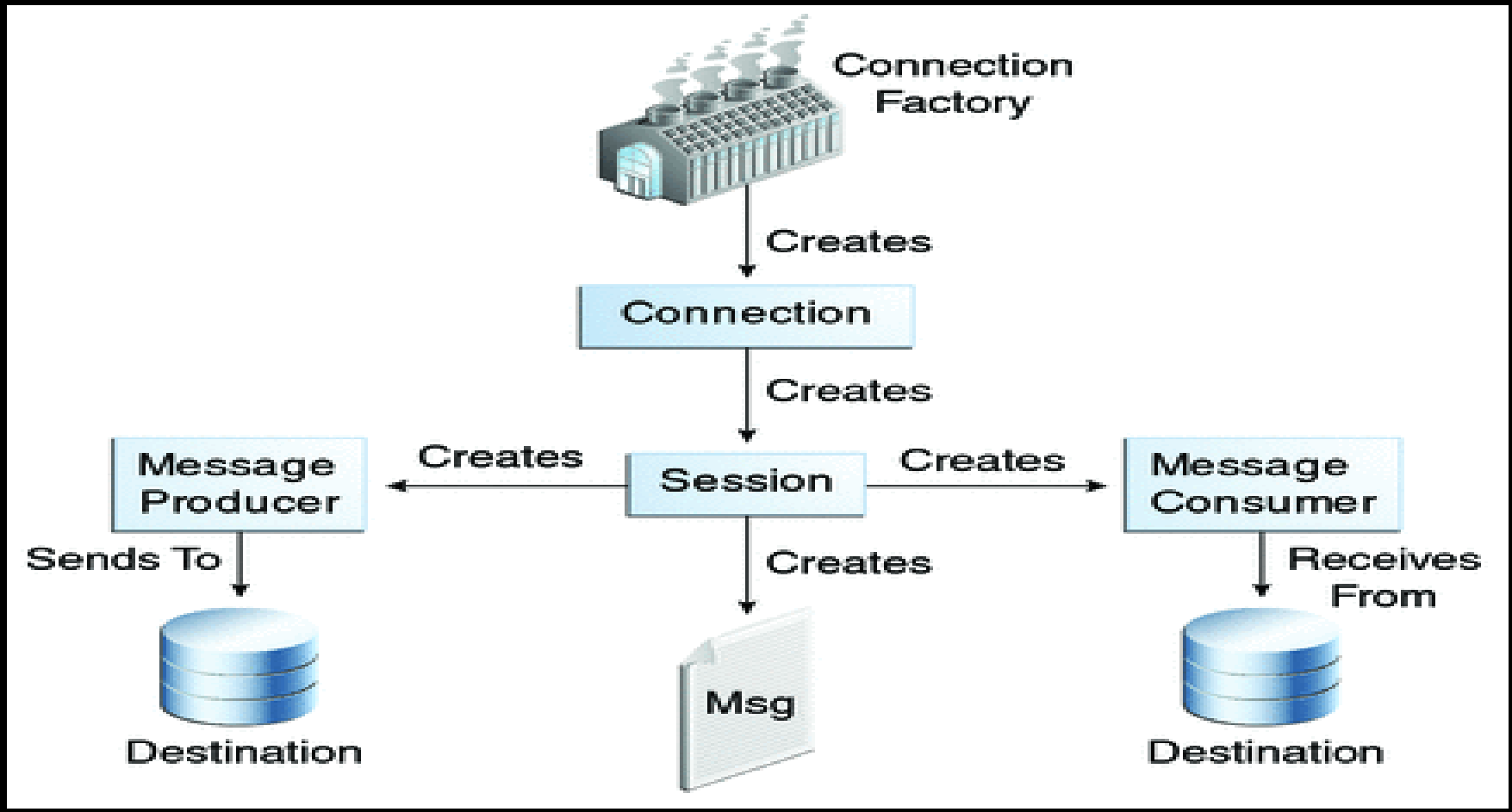
# The JMS API Programming Model

The basic building blocks of a JMS application consist of

- Administered objects: connection factories and destinations
- Connections
- Sessions
- Message producers
- Message consumers
- Messages

# Figure 45-5 shows how all these objects fit together in a JMS client application.

Figure 45-5 The JMS API Programming Model



This section describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects.

The last subsection briefly describes JMS API exception handling.

Examples that show how to combine all these objects in applications appear in later sections.

For more details, see the JMS API documentation, which is part of the Java EE API documentation.

# JMS Administered Objects

Two parts of a JMS application, destinations and connection factories, are best maintained administratively rather than programmatically.

The technology underlying these objects is likely to be very different from one implementation of the JMS API to another.

Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API.

Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

With GlassFish Server, you can use the `asadmin create-jms-resource` command or the Administration Console to create JMS administered objects in the form of connector resources.

You can also specify the resources in a file named `glassfish-resources.xml` that you can bundle with an application.

NetBeans IDE provides a wizard that allows you to create JMS resources for GlassFish Server.

See To Create JMS Resources Using NetBeans IDE for details.

# *JMS Connection Factories*

A **connection factory** is the **object** a client **uses** to create a connection to a provider.

A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator.

Each connection factory is an instance of the `ConnectionFactory`, `QueueConnectionFactory`, or `TopicConnectionFactory` interface.

To learn how to create connection factories, see To Create JMS Resources Using NetBeans IDE.

At the beginning of a JMS client program, you usually inject a connection factory resource into a `ConnectionFactory` object.

For example, the following code fragment specifies a resource whose JNDI name is `jms/ConnectionFactory` and assigns it to a `ConnectionFactory` object:

```
@Resource(
lookup = "jms/ConnectionFactory")
private static ConnectionFactory
connectionFactory;
```

In a Java EE application, JMS administered objects are normally placed in the `jms` naming subcontext.

# *JMS Destinations*

A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes.

In the PTP messaging domain, destinations are called queues.

In the pub/sub messaging domain, destinations are called topics.

A JMS application can use multiple queues or topics (or both).

To learn how to create destination resources, see To Create JMS Resources Using NetBeans IDE.

To create a destination using the GlassFish Server, you create a JMS destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination.

You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource.

Unlike connection factories, destinations are specific to one domain or the other.

To create an application that allows you to use the same code for both topics and queues, you assign the destination to a `Destination` object.

The following code specifies two resources, a queue and a topic.

The resource names are mapped to destination resources created in the JNDI namespace.

```
@Resource(lookup = "jms/Queue")
private static Queue queue;
@Resource(lookup = "jms/Topic")
private static Topic topic;
```

With the common interfaces, you can mix or match connection factories and destinations.

That is, in addition to using the `ConnectionFactory` interface, you can inject a `QueueConnectionFactory` resource and use it with a `Topic`, and you can inject a `TopicConnectionFactory` resource and use it with a `Queue`.

The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

# JMS Connections

A connection encapsulates a virtual connection with a JMS provider.

A connection could represent an open TCP/IP socket between a client and a provider service daemon.

You use a connection to create one or more sessions.

Connections implement the `Connection` interface.

When you have a `ConnectionFactory` object, you can use it to create a `Connection`:

```
Connection connection =
connectionFactory.
createConnection();
```

Before an application completes, you must close any connections that you have created.

Failure to close a connection can cause resources not to be released by the JMS provider.

# Closing a connection also closes its sessions and their message producers and message consumers.

```
connection.close();
```

Before your application can consume messages, you must call the connection's `start` method; for details, see JMS Message Consumers.

If you want to stop message delivery temporarily without closing the connection, you call the `stop` method.

# JMS Sessions

A session is a single-threaded context for producing and consuming messages.

# You use sessions to create the following:

- Message producers
- Message consumers
- Messages
- Queue browsers
- Temporary queues and topics
  (see Creating Temporary Destinations)

Sessions serialize the execution of message listeners; for details, see JMS Message Listeners.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

For details, see Using JMS API Local Transactions.

Sessions implement the `Session` interface.

After you create a Connection object, you use it to create a Session:

```
Session session =
connection.createSession
(false, Session.AUTO_ACKNOWLEDGE);
```

The first argument means that the session is not transacted;

the second means that the session automatically acknowledges messages when they have been received successfully.

(For more information, see Controlling Message Acknowledgment.)

To create a transacted session, use the following code:

```
Session session =
connection.createSession(true, 0);
```

Here, the first argument means that the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions.

For more information on transactions, see Using JMS API Local Transactions.

For information about the way JMS transactions work in Java EE applications, see Using the JMS API in Java EE Applications.

# JMS Message Producers

A message producer is an object that is created by a session and used for sending messages to a destination.

It implements the `MessageProducer` interface.

You use a `Session` to create a `MessageProducer` for a destination.

# The following examples show that you can create a producer for a `Destination` object, a `Queue` object, or a `Topic` object:

```
MessageProducer producer =
session.createProducer(dest);
MessageProducer producer =
session.createProducer(queue);
MessageProducer producer =
session.createProducer(topic);
```

You can create an unidentified producer by specifying `null` as the argument to `createProducer`.

With an unidentified producer, you do not specify a destination until you send a message.

After you have created a message producer, you can use it to send messages by using the `send` method:

```
producer.send(message);
```

You must first create the messages; see JMS Messages.

If you created an unidentified producer, use an overloaded `send` method that specifies the destination as the first parameter.

# For example:

```
MessageProducer anon_prod =
session.createProducer(null);
anon_prod.send(dest, message);
```

# JMS Message Consumers

A message consumer is an object that is created by a session and used for receiving messages sent to a destination.

It implements the `MessageConsumer` interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider.

The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you could use a `Session` to create a `MessageConsumer` for a `Destination` object, a `Queue` object, or a `Topic` object:

```
MessageConsumer consumer =
session.createConsumer(dest);
MessageConsumer consumer =
session.createConsumer(queue);
MessageConsumer consumer =
session.createConsumer(topic);
```

You use the `Session.createDurableSubscriber` method to create a durable topic subscriber.

This method is valid only if you are using a topic.

For details, see Creating Durable Subscriptions.

After you have created a message consumer, it becomes active, and you can use it to receive messages.

You can use the `close` method for a `MessageConsumer` to make the message consumer inactive.

Message delivery does not begin until you start the connection you created by calling its `start` method.

(Remember always to call the `start` method; forgetting to start the connection is one of the most common JMS programming errors.)

You use the `receive` method to consume a message synchronously.

You can use this method at any time after you call the `start` method:

```
connection.start();
Message m = consumer.receive();
connection.start();
Message m = consumer.receive(1000);
// time out after a second
```

To consume a message asynchronously, you use a message listener, described in the next section.

# *JMS Message Listeners*

A message listener is an object that acts as an asynchronous event handler for messages.

This object implements the `MessageListener` interface, which contains one method, `onMessage`.

In the `onMessage` method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the `setMessageListener` method.

For example, if you define a class named `Listener` that implements the `MessageListener` interface, you can register the message listener as follows:

```
Listener myListener =
new Listener();
consumer.setMessageListener
(myListener);
```

After you register the message listener, you call the `start` method on the `Connection` to begin message delivery.

(If you call `start` before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener's `onMessage` method whenever a message is delivered.

The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to any of the other message types (see Message Bodies).

A message listener is not specific to a particular destination type.

The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created.

A message listener does, however, usually expect a specific message type and format.

Your `onMessage` method should handle all exceptions.

It must not throw checked exceptions, and throwing a `RuntimeException` is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session.

At any time, only one of the session's message listeners is running.

In the Java EE platform, a message-driven bean is a special kind of message listener.

For details, see Using Message-Driven Beans to Receive Messages Asynchronously.

# *JMS Message Selectors*

**If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in.**

Message selectors assign the work of filtering messages to the JMS provider rather than to the application.

For an example of an application that uses a message selector, see An Application That Uses the JMS API with a Session Bean.

A message selector is a `String` that contains an expression.

The syntax of the expression is based on a subset of the SQL92 conditional expression syntax.

The message selector in the example selects any message that has a `NewsType` property that is set to the value `'Sports'` or `'Opinion'`:

```
NewsType = 'Sports' OR

NewsType = 'Opinion'
```

The `createConsumer` and `createDurableSubscriber` methods allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector.

# (See Message Headers, and Message Properties.)

A message selector cannot select messages on the basis of the content of the message body.

# JMS Messages

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications.

JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message has three parts: a header, properties, and a body. Only the header is required.

The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the `Message` interface in the API documentation.

# *Message Headers*

A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages.

Table 45-1 lists the JMS message header fields and indicates how their values are set.

For example, every message has a unique identifier, which is represented in the header field `JMSMessageID`.

The value of another header field, `JMSDestination`, represents the queue or the topic to which the message is sent.

Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface.

Some header fields are intended to be set by a client, but many are set automatically by the `send` or the `publish` method, which overrides any client-set values.

# Table 45-1 How JMS Message Header Field Values Are Set

| Header Field | Set By |
|---|---|
| `JMSDestination` | `send` or `publish` method |
| `JMSDeliveryMode` | `send` or `publish` method |
| `JMSExpiration` | `send` or `publish` method |
| `JMSPriority` | `send` or `publish` method |
| `JMSMessageID` | `send` or `publish` method |
| `JMSTimestamp` | `send` or `publish` method |
| `JMSCorrelationID` | Client |
| `JMSReplyTo` | Client |
| `JMSType` | Client |
| `JMSRedelivered` | JMS provider |

# *Message Properties*

You can create and set properties for messages if you need values in addition to those provided by the header fields.

You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see JMS Message Selectors).

For an example of setting a property to be used as a message selector, see An Application That Uses the JMS API with a Session Bean.

The JMS API provides some predefined property names that a provider can support.

The use either of these predefined properties or of user-defined properties is optional.

# *Message Bodies*

The JMS API defines five message body formats, also called message types, which allow you to send and to receive data in many different forms and provide compatibility with existing messaging formats.

Table 45-2 describes these message types.

# Table 45-2 JMS Message Types

| Message Type | Body Contains |
|---|---|
| `TextMessage` | A `java.lang.String` object (for example, the contents of an XML file). |
| `MapMessage` | A set of name-value pairs, with names as `String` objects and values as primitive types in the Java programming language.<br><br>The entries can be accessed sequentially by enumerator or randomly by name.<br><br>The order of the entries is undefined. |
| `BytesMessage` | A stream of uninterpreted bytes.<br><br>This message type is for literally encoding a body to |

| | |
|---|---|
| | match an existing message format. |
| `StreamMessage` | A stream of primitive values in the Java programming language, filled and read sequentially. |
| `ObjectMessage` | A `Serializable` object in the Java programming language. |
| `Message` | Nothing.<br><br>Composed of header fields and properties only.<br><br>This message type is useful when a message body is not required. |

The JMS API provides methods for creating messages of each type and for filling in their contents.

For example, to create and send a TextMessage, you might use the following statements:

```
TextMessage message =
session.createTextMessage();
message.setText(msg_text);
```

```
// msg_text is a String
producer.send(message);
```

At the consuming end, a message arrives as a generic `Message` object and must be cast to the appropriate message type.

You can use one or more getter methods to extract the message contents.

# The following code fragment uses the `getText` method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
TextMessage message =
(TextMessage) m;
System.out.println
("Reading message: " +
message.getText());
} else { // Handle error }
```

# JMS Queue Browsers

You can create a `QueueBrowser` object to inspect the messages in a queue.

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them.

Therefore, the JMS API provides an object that allows you to browse the messages in the queue and display the header values for each message.

To create a `QueueBrowser` object, use the `Session.createBrowser` method.

For example:

```
QueueBrowser browser =
session.createBrowser(queue);
```

See A Simple Example of Browsing Messages in a Queue for an example of the use of a `QueueBrowser` object.

The `createBrowser` method allows you to specify a message selector as a second argument when you create a `QueueBrowser`.

For information on message selectors, see JMS Message Selectors.

The JMS API provides no mechanism for browsing a topic.

Messages usually disappear from a topic as soon as they appear: if there are no message consumers to consume them, the JMS provider removes them.

Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, no facility exists for examining them.

# JMS Exception Handling

The root class for exceptions thrown by JMS API methods is `JMSException`.

Catching `JMSException` provides a generic way of handling all exceptions related to the JMS API.

# The JMSException class includes the following subclasses, which are described in the API documentation:

- IllegalStateException
- InvalidClientIDException
- InvalidDestinationException
- InvalidSelectorException
- JMSSecurityException
- MessageEOFException
- MessageFormatException
- MessageNotReadableException

- `MessageNotWriteableException`
- `ResourceAllocationException`
- `TransactionInProgressException`
- `TransactionRolledBackException`

All the examples in the tutorial catch and handle `JMSException` when it is appropriate to do so.

# Creating Robust JMS Applications

This section explains how to use features of the JMS API to achieve the level of reliability and performance your application requires.

Many people choose to implement JMS applications because they cannot tolerate dropped or duplicate messages and require that every message be received once and only once.

The JMS API provides this functionality.

The most reliable way to produce a message is to send a `PERSISTENT` message within a transaction.

JMS messages are `PERSISTENT` by default.

A transaction is a unit of work into which you can group a series of operations, such as message sends and receives, so that the operations either all succeed or all fail.

For details, see Specifying Message Persistence and Using JMS API Local Transactions.

The most reliable way to consume a message is to do so within a transaction, either from a queue or from a durable subscription to a topic.

For details, see Creating Temporary Destinations, Creating Durable Subscriptions, and Using JMS API Local Transactions.

For other applications, a lower level of reliability can reduce overhead and improve performance.

You can send messages with varying priority levels (see Setting Message Priority Levels) and you can set them to expire after a certain length of time (see Allowing Messages to Expire).

The JMS API provides several ways to achieve various kinds and degrees of reliability.

This section divides them into two categories, basic and advanced.

The following sections describe these features as they apply to JMS clients.

Some of the features work differently in Java EE applications; in these cases, the differences are noted here and are explained in detail in Using the JMS API in Java EE Applications.

# Using Basic Reliability Mechanisms

The basic mechanisms for achieving or affecting reliable message delivery are as follows:

- **Controlling message acknowledgment:** You can specify various levels of control over message acknowledgment.

- **Specifying message persistence: You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.**

- **Setting message priority levels: You can set various priority levels for messages, which can affect the order in which the messages are delivered.**

- **Allowing messages to expire:** You can specify an expiration time for messages so that they will not be delivered if they are obsolete.

- **Creating temporary destinations:** You can create temporary destinations that last only for the duration of the connection in which they are created.

# *Controlling Message Acknowledgment*

**Until a JMS message has been acknowledged, it is not considered to be successfully consumed.**

The successful consumption of a message ordinarily takes place in three stages.

1. The client receives the message.
2. The client processes the message.
3. The message is acknowledged.

Acknowledgment is initiated either by the JMS provider or by the client, depending on the session acknowledgment mode.

In transacted sessions (see Using JMS API Local Transactions), acknowledgment happens automatically when a transaction is committed.

If a transaction is rolled back, all consumed messages are redelivered.

In nontransacted sessions, when and how a message is acknowledged depend on the value specified as the second argument of the `createSession` method.

The three possible argument values are as follows:

- `Session.AUTO_ACKNOWLEDGE`: The session automatically acknowledges a client's receipt of a message either when the client has successfully returned from a call to `receive` or when the `MessageListener` it has called to process the message returns successfully.

A synchronous receive in an `AUTO_ACKNOWLEDGE` session is the one exception to the rule that message consumption is a three-stage process as described earlier.

In this case, the receipt and acknowledgment take place in one step, followed by the processing of the message.

- `Session.CLIENT_ACKNOWLEDGE`: A client acknowledges a message by calling the message's `acknowledge` method.

  In this mode, acknowledgment takes place on the session level: Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been consumed by its session.

For example, if a message consumer consumes ten messages and then acknowledges the fifth message delivered, all ten messages are acknowledged.

- `Session.DUPS_OK_ACKNOWLEDGE`: This option instructs the session to lazily acknowledge the delivery of messages.

This is likely to result in the delivery of some duplicate messages if the JMS provider fails, so it should be used only by consumers that can tolerate duplicate messages.

(If the JMS provider redelivers a message, it must set the value of the `JMSRedelivered` message header to `true`.) This option can reduce session overhead by minimizing the work the session does to prevent duplicates.

If messages have been received from a queue but not acknowledged when a session terminates, the JMS provider retains them and redelivers them when a consumer next accesses the queue.

The provider also retains unacknowledged messages for a terminated session that has a durable `TopicSubscriber`.

(See Creating Durable Subscriptions.)

Unacknowledged messages for a nondurable `TopicSubscriber` are dropped when the session is closed.

If you use a queue or a durable subscription, you can use the `Session.recover` method to stop a nontransacted session and restart it with its first unacknowledged message.

In effect, the session's series of delivered messages is reset to the point after its last acknowledged message.

The messages it now delivers may be different from those that were originally delivered, if messages have expired or if higher-priority messages have arrived.

For a nondurable `TopicSubscriber`, the provider may drop unacknowledged messages when its session is recovered.

The sample program in XREF the next section demonstrates two ways to ensure that a message will not be acknowledged until processing of the message is complete.

# *Specifying Message Persistence*

The JMS API supports two delivery modes for messages to specify whether messages are lost if the JMS provider fails.

These delivery modes are fields of the `DeliveryMode` interface.

- The `PERSISTENT` delivery mode, which is the default, instructs the JMS provider to take extra care to ensure that a message is not lost in transit in case of a JMS provider failure.

  A message sent with this delivery mode is logged to stable storage when it is sent.

- The `NON_PERSISTENT` delivery mode does not require the JMS provider to store the message or otherwise guarantee that it is not lost if the provider fails.

You can specify the delivery mode in either of two ways.

- You can use the `setDeliveryMode` method of the `MessageProducer` interface to set the delivery mode for all messages sent by that producer.

For example, the following call sets the delivery mode to `NON_PERSISTENT` for a producer:

```
producer.setDeliveryMode
 (DeliveryMode.NON_PERSISTENT);
```

- You can use the long form of the `send` or the `publish` method to set the delivery mode for a specific message.

  The second argument sets the delivery mode.

  For example, the following `send` call sets the delivery mode for `message` to `NON_PERSISTENT`:

```
producer.send(message,
DeliveryMode.NON_PERSISTENT, 3,
10000);
```

The third and fourth arguments set the priority level and expiration time, which are described in the next two subsections.

If you do not specify a delivery mode, the default is `PERSISTENT`.

Using the `NON_PERSISTENT` delivery mode may improve performance and reduce storage overhead, but you should use it only if your application can afford to miss messages.

# *Setting Message Priority Levels*

**You can use message priority levels to instruct the JMS provider to deliver urgent messages first.**

**You can set the priority level in either of two ways.**

- You can use the `setPriority` method of the `MessageProducer` interface to set the priority level for all messages sent by that producer.

  For example, the following call sets a priority level of 7 for a producer:

```
producer.setPriority(7);
```

- You can use the long form of the `send` or the `publish` method to set the priority level for a specific message.

  The third argument sets the priority level.

  For example, the following `send` call sets the priority level for `message` to 3:

```
producer.send(message,
DeliveryMode.NON_PERSISTENT, 3,
10000);
```

The ten levels of priority range from 0 (lowest) to 9 (highest).

If you do not specify a priority level, the default level is 4.

A JMS provider tries to deliver higher-priority messages before lower-priority ones but does not have to deliver messages in exact order of priority.

# *Allowing Messages to Expire*

**By default, a message never expires.**

**If a message will become obsolete after a certain period, however, you may want to set an expiration time.**

**You can do this in either of two ways.**

- You can use the `setTimeToLive` method of the `MessageProducer` interface to set a default expiration time for all messages sent by that producer.

  For example, the following call sets a time to live of one minute for a producer:

```
producer.setTimeToLive(60000);
```

- You can use the long form of the `send` or the `publish` method to set an expiration time for a specific message.

  The fourth argument sets the expiration time in milliseconds.

  For example, the following `send` call sets a time to live of 10 seconds:

```
producer.send(message,
DeliveryMode.NON_PERSISTENT, 3,
10000);
```

If the specified `timeToLive` value is `0`, the message never expires.

When the message is sent, the specified `timeToLive` is added to the current time to give the expiration time.

Any message not delivered before the specified expiration time is destroyed.

The destruction of obsolete messages conserves storage and computing resources.

# *Creating Temporary Destinations*

Normally, you create JMS destinations (queues and topics) administratively rather than programmatically.

Your JMS provider includes a tool that you use to create and remove destinations, and it is common for destinations to be long-lasting.

The JMS API also enables you to create destinations (`TemporaryQueue` and `TemporaryTopic` objects) that last only for the duration of the connection in which they are created.

You create these destinations dynamically using the `Session.createTemporaryQueue` and the `Session.createTemporaryTopic` methods.

The only message consumers that can consume from a temporary destination are those created by the same connection that created the destination.

Any message producer can send to the temporary destination.

If you close the connection that a temporary destination belongs to, the destination is closed and its contents are lost.

You can use temporary destinations to implement a simple request/reply mechanism.

If you create a temporary destination and specify it as the value of the `JMSReplyTo` message header field when you send a message, then the consumer of the message can use the value of the `JMSReplyTo` field as the destination to which it sends a reply.

The consumer can also reference the original request by setting the `JMSCorrelationID` header field of the reply message to the value of the `JMSMessageID` header field of the request.

For example, an `onMessage` method can create a session so that it can send a reply to the message it receives.

# It can use code such as the following:

```
producer = session.createProducer
(msg.getJMSReplyTo());
replyMsg =
session.createTextMessage
("Consumer " + "processed message:"
+ msg.getText());
replyMsg.setJMSCorrelationID
(msg.getJMSMessageID());
```

```
producer.send(replyMsg);
```

For more examples, see Chapter 46, Java Message Service Examples.

# Using Advanced Reliability Mechanisms

The more advanced mechanisms for achieving reliable message delivery are the following:

- **Creating durable subscriptions:** You can create durable topic subscriptions, which receive messages published while the subscriber is not active.

Durable subscriptions offer the reliability of queues to the publish/subscribe message domain.

- Using local transactions: You can use local transactions, which allow you to group a series of sends and receives into an atomic unit of work.

# Transactions are rolled back if they fail at any time.

# *Creating Durable Subscriptions*

To ensure that a pub/sub application receives all published messages, use `PERSISTENT` delivery mode for the publishers.

In addition, use durable subscriptions for the subscribers.

The `Session.createConsumer` method creates a nondurable subscriber if a topic is specified as the destination.

A nondurable subscriber can receive only messages that are published while it is active.

At the cost of higher overhead, you can use the `Session.createDurableSubscriber` method to create a durable subscriber.

A durable subscription can have only one active subscriber at a time.

A durable subscriber registers a durable subscription by specifying a unique identity that is retained by the JMS provider.

Subsequent subscriber objects that have the same identity resume the subscription in the state in which it was left by the preceding subscriber.

If a durable subscription has no active subscriber, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of a durable subscriber by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscriber

You set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console.

After using this connection factory to create the connection and the session, you call the `createDurableSubscriber` method with two arguments: the topic and a string that specifies the name of the subscription:

```
String subName = "MySub";
MessageConsumer topicSubscriber =
session.createDurableSubscriber
(myTopic, subName);
```

The subscriber becomes active after you start the `Connection` or `TopicConnection`.

Later, you might close the subscriber:

```
topicSubscriber.close();
```

The JMS provider stores the messages sent or published to the topic, as it would store messages sent to a queue.

If the program or another application calls `createDurableSubscriber` using the same connection factory and its client ID, the same topic, and the same subscription name, the subscription is reactivated, and the JMS provider delivers the messages that were published while the subscriber was inactive.

To delete a durable subscription, first close the subscriber, and then use the `unsubscribe` method, with the subscription name as the argument:

```
topicSubscriber.close();
session.unsubscribe("MySub");
```

The `unsubscribe` method deletes the state that the provider maintains for the subscriber.

Figure 45-6 and Figure 45-7 show the difference between a nondurable and a durable subscriber.

With an ordinary, nondurable subscriber, the subscriber and the subscription begin and end at the same point and are, in effect, identical.

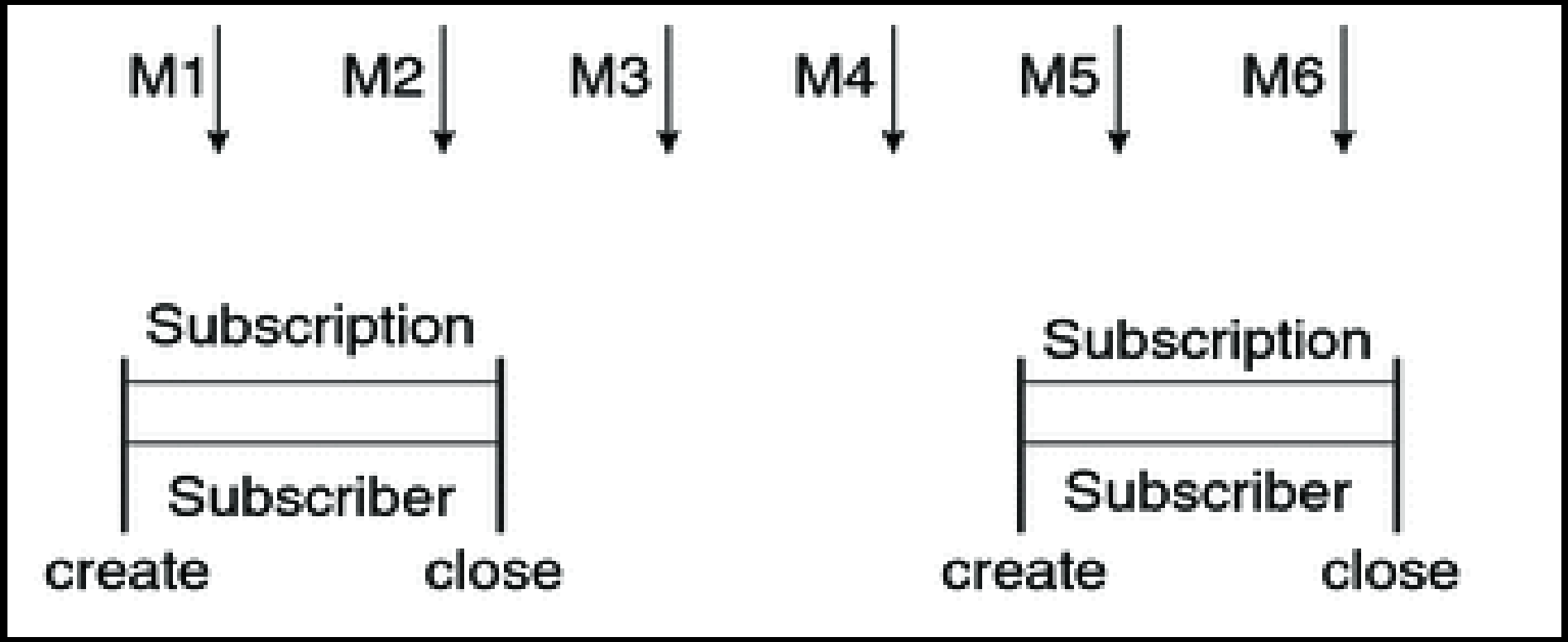
When a subscriber is closed, the subscription also ends.

Here, `create` stands for a call to `Session.createConsumer` with a `Topic` argument, and `close` stands for a call to `MessageConsumer.close`.

Any messages published to the topic between the time of the first `close` and the time of the second `create` are not consumed by the subscriber.

# In Figure 45-6, the subscriber consumes messages M1, M2, M5, and M6, but messages M3 and M4 are lost.

Figure 45-6 Nondurable Subscribers and Subscriptions



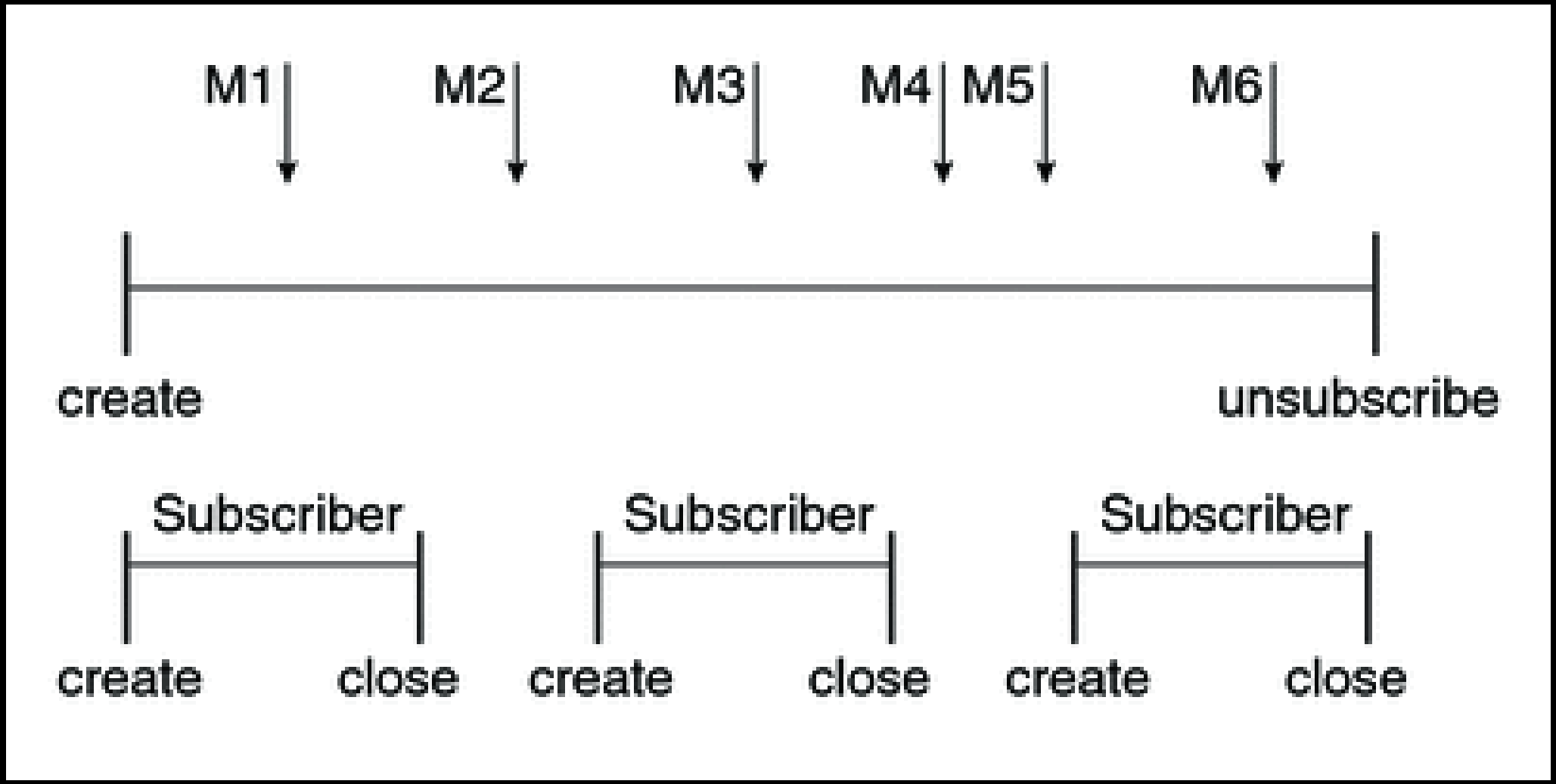
With a durable subscriber, the subscriber can be closed and re-created, but the subscription continues to exist and to hold messages until the application calls the `unsubscribe` method.

In Figure 45-7, `create` stands for a call to `Session.createDurableSubscriber`, `close` stands for a call to `MessageConsumer.close`, and `unsubscribe` stands for a call to `Session.unsubscribe`.

Messages published while the subscriber is closed are received when the subscriber is created again.

So even though messages M2, M4, and M5 arrive while the subscriber is closed, they are not lost.

Figure 45-7 A Durable Subscriber and Subscription



See A Message Acknowledgment Example, A Durable Subscription Example, and An Application That Uses the JMS API with a Session Bean for examples of Java EE applications that use durable subscriptions.

# *Using JMS API Local Transactions*

You can group a series of operations into an atomic unit of work called a transaction.

If any one of the operations fails, the transaction can be rolled back, and the operations can be attempted again from the beginning.

If all the operations succeed, the transaction can be committed.

In a JMS client, you can use local transactions to group message sends and receives.

The JMS API `Session` interface provides `commit` and `rollback` methods that you can use in a JMS client.

A transaction commit means that all produced messages are sent and all consumed messages are acknowledged.

A transaction rollback means that all produced messages are destroyed and all consumed messages are recovered and redelivered unless they have expired (see Allowing Messages to Expire).

A transacted session is always involved in a transaction.

As soon as the `commit` or the `rollback` method is called, one transaction ends and another transaction begins.

Closing a transacted session rolls back its transaction in progress, including any pending sends and receives.

In an Enterprise JavaBeans component, you cannot use the `Session.commit` and `Session.rollback` methods.

Instead, you use distributed transactions, which are described in Using the JMS API in Java EE Applications.

You can combine several sends and receives in a single JMS API local transaction.

If you do so, you need to be careful about the order of the operations.

You will have no problems if the transaction consists of all sends or all receives or if the receives come before the sends.

But if you try to use a request/reply mechanism, whereby you send a message and then try to receive a reply to the sent message in the same transaction, the program will hang,

because the send cannot take place until the transaction is committed.

The following code fragment illustrates the problem:

```
// Don't do this!
outMsg.setJMSReplyTo(replyQueue);
producer.send(outQueue, outMsg);
consumer =
session.createConsumer(replyQueue);
inMsg = consumer.receive();
session.commit();
```

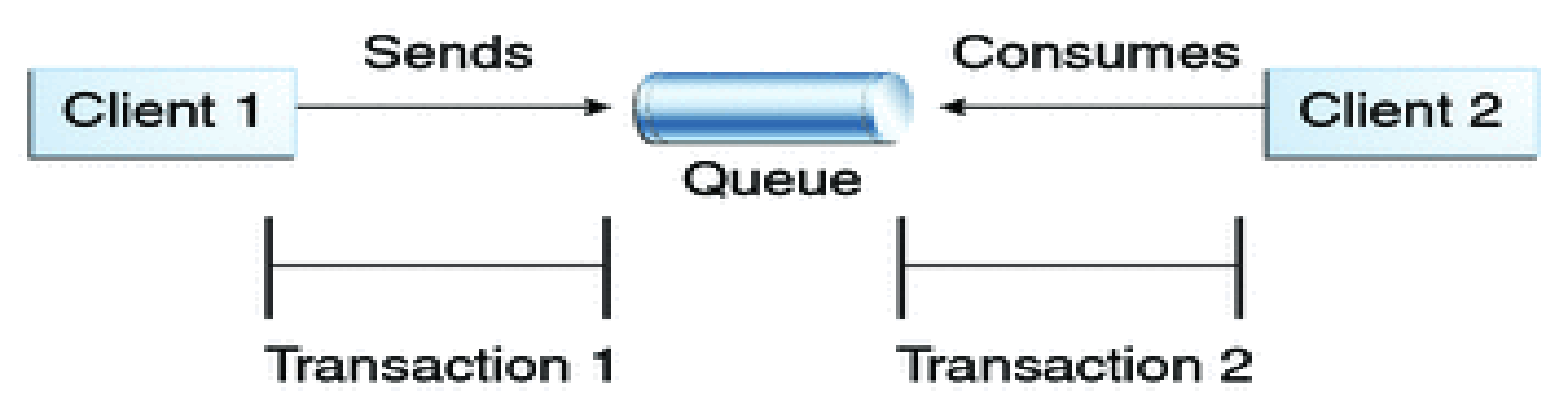
Because a message sent during a transaction is not actually sent until the transaction is committed, the transaction cannot contain any receives that depend on that message's having been sent.

In addition, the production and the consumption of a message cannot both be part of the same transaction.

The reason is that the transactions take place between the clients and the JMS provider, which intervenes between the production and the consumption of the message.

Figure 45-8 illustrates this interaction.

Figure 45-8 Using JMS API Local Transactions



The sending of one or more messages to one or more destinations by client 1 can form a single transaction, because it forms a single set of interactions with the JMS provider using a single session.

Similarly, the receiving of one or more messages from one or more destinations by client 2 also forms a single transaction using a single session.

But because the two clients have no direct interaction and are using two different sessions, no transactions can take place between them.

Another way of putting this is that the act of producing and/or consuming messages in a session can be transactional, but the act of producing and consuming a specific message across different sessions cannot be transactional.

This is the fundamental difference between messaging and synchronized processing.

Instead of tightly coupling the sending and receiving of data, message producers and consumers use an alternative approach to reliability, one that is built on a JMS provider's ability to supply a once-and-only-once message delivery guarantee.

When you create a session, you specify whether it is transacted.

The first argument to the `createSession` method is a `boolean` value.

A value of `true` means that the session is transacted; a value of `false` means that it is not transacted.

The second argument to this method is the acknowledgment mode, which is relevant only to nontransacted sessions (see Controlling Message Acknowledgment).

If the session is transacted, the second argument is ignored, so it is a good idea to specify 0 to make the meaning of your code clear.

For example:

```
session =
connection.createSession(true, 0);
```

The `commit` and the `rollback` methods for local transactions are associated with the session.

You can combine queue and topic operations in a single transaction if you use the same session to perform the operations.

For example, you can use the same session to receive a message from a queue and send a message to a topic in the same transaction.

You can pass a client program's session to a message listener's constructor function and use it to create a message producer.

In this way, you can use the same session for receives and sends in asynchronous message consumers.

A Local Transaction Example provides an example of the use of JMS API local transactions.

# Using the JMS API in Java EE Applications

This section describes the ways in which using the JMS API in enterprise bean applications or web applications differs from using it in application clients.

A general rule in the Java EE platform specification applies to all Java EE components that use the JMS API within EJB or web containers:

Application components in the web and EJB containers must not attempt to create more than one active (not closed) `Session` object per connection.

This rule does not apply to application clients.

The application client container supports the creation of multiple sessions for each connection.

# Using `@Resource` Annotations in Enterprise Bean or Web Components

When you use the `@Resource` annotation in an application client component, you normally declare the JMS resource static:

```
@Resource(
lookup = "jms/ConnectionFactory")
private static ConnectionFactory
connectionFactory;
@Resource(lookup = "jms/Queue")
private static Queue queue;
```

However, when you use this annotation in a session bean, a message-driven bean, or a web component, do not declare the resource static:

```
@Resource(lookup =
"jms/ConnectionFactory")
private ConnectionFactory
connectionFactory;
@Resource(lookup = "jms/Topic")
private Topic topic;
```

**If you declare the resource static, runtime errors will result.**

# Using Session Beans to Produce and to Synchronously Receive Messages

An application that produces messages or synchronously receives them can use a session bean to perform these operations.

The example in An Application That Uses the JMS API with a Session Bean uses a stateless session bean to publish messages to a topic.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in an enterprise bean.

Instead, use a timed synchronous receive, or use a message-driven bean to receive messages asynchronously.

For details about blocking and timed synchronous receives, see Writing the Clients for the Synchronous Receive Example.

Using the JMS API in an enterprise bean or web application is in many ways similar to using it in an application client.

The main differences are the areas of resource management and transactions.

# *Resource Management*

The JMS API resources are a JMS API connection and a JMS API session.

In general, it is important to release JMS resources when they are no longer being used.

Here are some useful practices to follow.

- If you wish to maintain a JMS API resource only for the life span of a business method, it is a good idea to close the resource in a `finally` block within the method.

- If you would like to maintain a JMS API resource for the life span of an enterprise bean instance, it is a good idea to use a `@PostConstruct` callback method to create the resource and to use a `@PreDestroy` callback method to close the resource.

If you use a stateful session bean and you wish to maintain the JMS API resource in a cached state, you must close the resource in a `@PrePassivate` callback method and set its value to `null`, and you must create it again in a `@PostActivate` callback method.

# *Transactions*

Instead of using local transactions, you use container-managed transactions for bean methods that perform sends or receives, allowing the EJB container to handle transaction demarcation.

Because container-managed transactions are the default, you do not have to use an annotation to specify them.

You can use bean-managed transactions and the `javax.transaction.UserTransaction` interface's transaction demarcation methods, but you should do so only if your application has special requirements and you are an expert in using transactions.

Usually, container-managed transactions produce the most efficient and correct behavior.

This tutorial does not provide any examples of bean-managed transactions.

# Using Message-Driven Beans to Receive Messages Asynchronously

The sections What Is a Message-Driven Bean? and How Does the JMS API Work with the Java EE Platform? describe how the Java EE platform supports a special kind of enterprise bean, the message-driven bean, which allows Java EE applications to process JMS messages asynchronously.

Session beans allow you to send messages and to receive them synchronously but not asynchronously.

A message-driven bean is a message listener that can reliably consume messages from a queue or a durable subscription.

The messages can be sent by any Java EE component (from an application client, another enterprise bean, or a web component) or from an application or a system that does not use Java EE technology.

Like a message listener in an application client, a message-driven bean contains an `onMessage` method that is called automatically when a message arrives.

Like a message listener, a message-driven bean class can implement helper methods invoked by the `onMessage` method to aid in message processing.

A message-driven bean, however, differs from an application client's message listener in the following ways:

- Certain setup tasks are performed by the EJB container.

- The bean class uses the `@MessageDriven` annotation to specify properties for the bean or the connection factory, such as a destination type, a durable subscription, a message selector, or an acknowledgment mode.

The examples in Chapter 46, Java Message Service Examples show how the JMS resource adapter works in the GlassFish Server.

The EJB container automatically performs several setup tasks that a stand-alone client has to do:

- Creating a message consumer to receive the messages.

Instead of creating a message consumer in your source code, you associate the message-driven bean with a destination and a connection factory at deployment time.

If you want to specify a durable subscription or use a message selector, you do this at deployment time also.

- Registering the message listener.

  You must not call `setMessageListener`.

- Specifying a message acknowledgment mode.

  The default mode, `AUTO_ACKNOWLEDGE`, is used unless it is overriden by a property setting.

If JMS is integrated with the application server using a resource adapter, the JMS resource adapter handles these tasks for the EJB container.

Your message-driven bean class must implement the `javax.jms.MessageListener` interface and the `onMessage` method.

It may implement a `@PostConstruct` callback method to create a connection, and a `@PreDestroy` callback method to close the connection.

Typically, it implements these methods if it produces messages or does synchronous receives from another destination.

The bean class commonly injects a `MessageDrivenContext` resource, which provides some additional methods that you can use for transaction management.

The main difference between a message-driven bean and a session bean is that a message-driven bean has no local or remote interface.

Instead, it has only a bean class.

A message-driven bean is similar in some ways to a stateless session bean: Its instances are relatively short-lived and retain no state for a specific client.

The instance variables of the message-driven bean instance can contain some state across the handling of client messages: for example, a JMS API connection, an open database connection, or an object reference to an enterprise bean object.

Like a stateless session bean, a message-driven bean can have many interchangeable instances running at the same time.

The container can pool these instances to allow streams of messages to be processed concurrently.

The container attempts to deliver messages in chronological order when it does not impair the concurrency of message processing,

but no guarantees are made as to the exact order in which messages are delivered to the instances of the message-driven bean class.

Because concurrency can affect the order in which messages are delivered, you should write your applications to handle messages that arrive out of sequence.

For example, your application could manage conversations by using application-level sequence numbers.

An application-level conversation control mechanism with a persistent conversation state could cache later messages until earlier messages have been processed.

Another way to ensure order is to have each message or message group in a conversation require a confirmation message that the sender blocks on receipt of.

This forces the responsibility for order back on the sender and more tightly couples senders to the progress of message-driven beans.

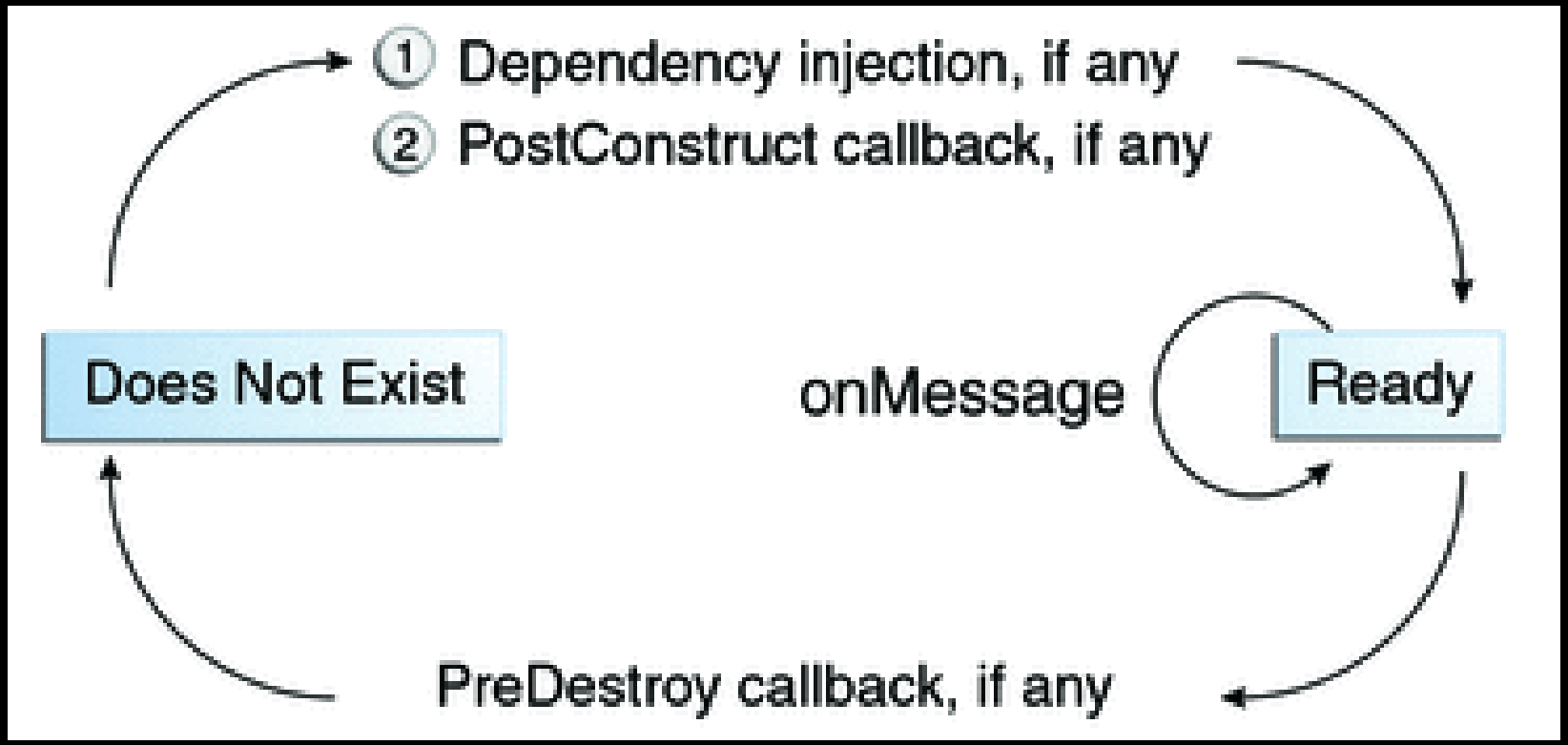
# To create a new instance of a message-driven bean, the container does the following:

- Instantiates the bean
- Performs any required resource injection
- Calls the `@PostConstruct` callback method, if it exists

To remove an instance of a message-driven bean, the container calls the `@PreDestroy` callback method.

Figure 45-9 shows the lifecycle of a message-driven bean.

Figure 45-9 Lifecycle of a Message-Driven Bean



# Managing Distributed Transactions

JMS client applications use JMS API local transactions (described in Using JMS API Local Transactions), which allow the grouping of sends and receives within a specific JMS session.

Java EE applications commonly use distributed transactions to ensure the integrity of accesses to external resources.

For example, distributed transactions allow multiple applications to perform atomic updates on the same database, and they allow a single application to perform atomic updates on multiple databases.

In a Java EE application that uses the JMS API, you can use transactions to combine message sends or receives with database updates and other resource manager operations.

You can access resources from multiple application components within a single transaction.

For example, a servlet can start a transaction, access multiple databases, invoke an enterprise bean that sends a JMS message, invoke another enterprise bean that modifies an EIS system using the Connector architecture, and finally commit the transaction.

Your application cannot, however, both send a JMS message and receive a reply to it within the same transaction; the restriction described in Using JMS API Local Transactions still applies.

Distributed transactions within the EJB container can be either of two kinds:

- **Container-managed transactions: The EJB container controls the integrity of your transactions without your having to call `commit` or `rollback`.**

  **Container-managed transactions are recommended for Java EE applications that use the JMS API.**

  **You can specify appropriate transaction attributes for your enterprise bean methods.**

Use the `Required` transaction attribute (the default) to ensure that a method is always part of a transaction.

If a transaction is in progress when the method is called, the method will be part of that transaction; if not, a new transaction will be started before the method is called and will be committed when the method returns.

- **Bean-managed transactions:** You can use these in conjunction with the `javax.transaction.UserTransaction` interface, which provides its own `commit` and `rollback` methods that you can use to delimit transaction boundaries.

  Bean-managed transactions are recommended only for those who are experienced in programming transactions.

You can use either container-managed transactions or bean-managed transactions with message-driven beans.

To ensure that all messages are received and handled within the context of a transaction, use container-managed transactions and use the `Required` transaction attribute (the default) for the `onMessage` method.

This means that if there is no transaction in progress, a new transaction will be started before the method is called and will be committed when the method returns.

When you use container-managed transactions, you can call the following `MessageDrivenContext` methods:

- `setRollbackOnly`: Use this method for error handling.

  If an exception occurs, `setRollbackOnly` marks the current transaction so that the only possible outcome of the transaction is a rollback.

- `getRollbackOnly`: Use this method to test whether the current transaction has been marked for rollback.

If you use bean-managed transactions, the delivery of a message to the `onMessage` method takes place outside the distributed transaction context.

The transaction begins when you call the `UserTransaction.begin` method within the `onMessage` method, and it ends when you call `UserTransaction.commit` or `UserTransaction.rollback`.

Any call to the `Connection.createSession` method must take place within the transaction.

If you call `UserTransaction.rollback`, the message is not redelivered, whereas calling `setRollbackOnly` for container-managed transactions does cause a message to be redelivered.

Neither the JMS API specification nor the Enterprise JavaBeans specification (available from http://jcp.org/en/jsr/detail?id=318) specifies how to handle calls to JMS API methods outside transaction boundaries.

The Enterprise JavaBeans specification does state that the EJB container is responsible for acknowledging a message that is successfully processed by the `onMessage` method of a message-driven bean that uses bean-managed transactions.

Using bean-managed transactions allows you to process the message by using more than one transaction or to have some parts of the message processing take place outside a transaction context.

In most cases, however, container-managed transactions provide greater reliability and are therefore preferable.

When you create a session in an enterprise bean, the container ignores the arguments you specify, because it manages all transactional properties for enterprise beans.

It is still a good idea to specify arguments of `true` and `0` to the `createSession` method to make this situation clear:

```
session =
connection.createSession(true, 0);
```

When you use container-managed transactions, you normally use the `Required` transaction attribute (the default) for your enterprise bean's business methods.

You do not specify a message acknowledgment mode when you create a message-driven bean that uses container-managed transactions.

The container acknowledges the message automatically when it commits the transaction.

If a message-driven bean uses bean-managed transactions, the message receipt cannot be part of the bean-managed transaction,

so the container acknowledges the message outside the transaction.

If the `onMessage` method throws a `RuntimeException`, the container does not acknowledge processing the message.

In that case, the JMS provider will redeliver the unacknowledged message in the future.

# Using the JMS API with Application Clients and Web Components

An application client in a Java EE application can use the JMS API in much the same way that a stand-alone client program does.

It can produce messages, and it can consume messages by using either synchronous receives or message listeners.

See Chapter 25, A Message-Driven Bean Example for an example of an application client that produces messages.

For an example of using an application client to produce and to consume messages, see An Application Example That Deploys a Message-Driven Bean on Two Servers.

The Java EE platform specification does not impose strict constraints on how web components should use the JMS API.

In the GlassFish Server, a web component can send messages and consume them synchronously but cannot consume them asynchronously.

Because a blocking synchronous receive ties up server resources, it is not a good programming practice to use such a `receive` call in a web component.

Instead, use a timed synchronous receive.

For details about blocking and timed synchronous receives, see Writing the Clients for the Synchronous Receive Example.

# Further Information about JMS

For more information about JMS, see:

- Java Message Service web site:

  http://www.oracle.com/technetwork/java/index-jsp-142945.html

- Java Message Service specification, version 1.1, available from

  http://www.oracle.com/technetwork/java/docs-136352.html