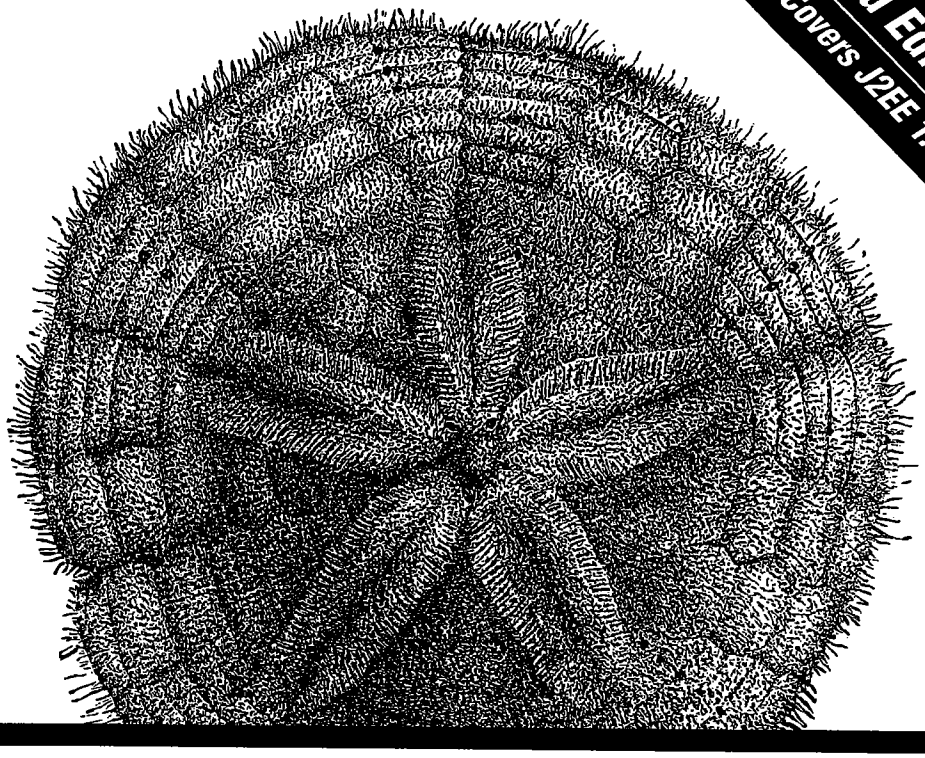
2nd Edition
Covers J2EE 1.3



JAVA™ ENTERPRISE IN A NUTSHELL

A Desktop Quick Reference

Jim Farley, William Crawford & David Flanagan

O'REILLY®

```
        // Check for a null Base URI, and provider it if so
        if((base == null) || (base.equals("/servlet/")))
         base = "http://www.oreilly.com/catalog/jentnut/";
        if(href == null)
         return null;
        return new StreamSource(base + href);
       }
    }
```

Finally, JAXP supports XSLT transformations that can convert between DOM, SAX, and streams, transform an XML document based on an XSL stylesheet, or both.

CHAPTER 10

Java Message Service

The standard Java networking APIs, as well as remote object systems such as RMI and CORBA, operate by default under the assumption of synchronous communications. In other words, if a client makes a request of a remote server (e.g., opens a socket and attempts to read some data from it, or makes a remote method call on a remote object), the thread that made the request will block until the response comes back from the server. In some situations, it might be necessary or useful to engage in asynchronous communications, e.g., a client sends a request to the server and then continues doing other work, while the server possibly invokes some kind of callback on the client when the request is complete. This is where the Java Message Service (JMS) comes in.

JMS is an API for performing asynchronous messaging. JMS is principally a client-focused API, in that it provides a standard, portable interface for Java/J2EE clients to interact with native message-oriented middleware (MOM) systems like IBM MQ Series, Sonic MQ, etc. JMS isn't intended to be a platform for implementing a full messaging system, since it doesn't provide a service-provider interface for all of the internals of a message-service implementation. In a sense, JMS plays a role with native messaging systems that is analogous with the role that JDBC plays with relational database systems, or the role JNDI plays with naming and directory services. Java clients using JMS to interact with messaging systems can (ostensibly) be more easily ported from one native messaging system to another, because they are insulated from the proprietary particulars of the underlying vendor's message system.

The JMS API is provided in the `javax.jms` package. The material in this chapter is based on Version 1.0.2b of the JMS specification, released in August 2001, which is the most current version at the time of this writing. The examples in the chapter have been tested against the JMS services embedded in Sun's J2EE 1.3 Reference Implementation and BEA's WebLogic 6.1 application server.

JMS in the J2EE Environment

The J2EE 1.2 specification requires that compliant J2EE servers support only JMS clients accessing external JMS providers. In other words, a J2EE 1.2 server only needs to provide the JMS API to allow components to interact with external JMS servers, and doesn't need to provide a JMS implementation of its own. J2EE 1.3 extended this requirement to include a full JMS provider, including support for both point-to-point and publish-subscribe message destinations (these are described in detail next). So any compliant J2EE 1.3 server will have its own JMS server capable of hosting its own message destinations.

Given this, the material concerning developing JMS clients is relevant regardless of whether you are using a J2EE 1.2- or 1.3-compliant application server. The material about the setup and configuration of JMS destinations requires a JMS provider, so you'll need a full JMS provider, either as part of a J2EE 1.3 server, an extended J2EE 1.2 server, or as a standalone JMS server.

Elements of Messaging with JMS

The principle players in a JMS system are *messaging clients*, *message destinations*, and a JMS-compatible *messaging provider*.

Messaging clients produce and consume messages. Typically messaging takes place asynchronously; a client produces a message and sends it to a message destination, and some time later another client receives the message. Message clients can be implemented using JMS, or they can use a native messaging API to participate in the messaging system. If a native message client (e.g., a client using the native IBM MQ Series APIs) produces a message to a message destination, a JMS connection to the native message system is responsible for retrieving the message, converting it into the appropriate JMS message representation, and delivering it to any relevant JMS-based clients.

Message destinations are places where JMS clients send and receive messages from. Message destinations are created within a JMS provider that manages all of the administrative and runtime functions of the messaging system. At a minimum, a JMS provider allows you to specify a network address for a destination, allowing clients to find the destination on the network. But providers may also support other administrative options on destinations, such as persistence options, resource limits, etc.

Messaging Styles: Point-to-Point and Publish-Subscribe

Generally speaking, asynchronous messaging usually comes in two flavors: a message can be addressed and sent to a single receiver (*point-to-point*), or a message can be published to particular channel or topic and any receiver that subscribes to that channel will receive the message (*publish-subscribe*). These two messaging styles have analogies at several levels in the distributed computing "stack," all the way from the network level (standard TCP packet delivery versus multicast networking) to the application level (email versus newsgroups). Figure 10-1 depicts the two message models supported by JMS, as well as the key

interfaces that come into play in a JMS cont
interfaces later in the chapter.

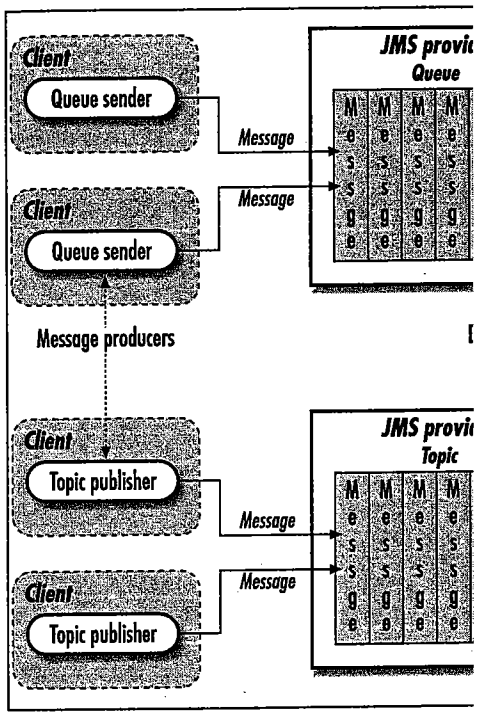


Figure 10-1: JMS message models

Most messaging providers support one or b
provides support for them both in its API. JM
interfaces, described next. Each style of m
subclasses of these generic interfaces.

Key JMS Interfaces

The following key interfaces represent the cc
client application, whether it is using
messaging. Information about all of the cla
JMS API can be found in Part III.

`Message`
: Messages are at the heart of JMS, natur
for their header fields, properties, and
face provide implementations for differe

`MessageListener`
: A `MessageListener` is attached to a Mes
a callback for each `Message` received by
the key to asynchronous message deliv

interfaces that come into play in a JMS context. We discuss the specifics of these interfaces later in the chapter.

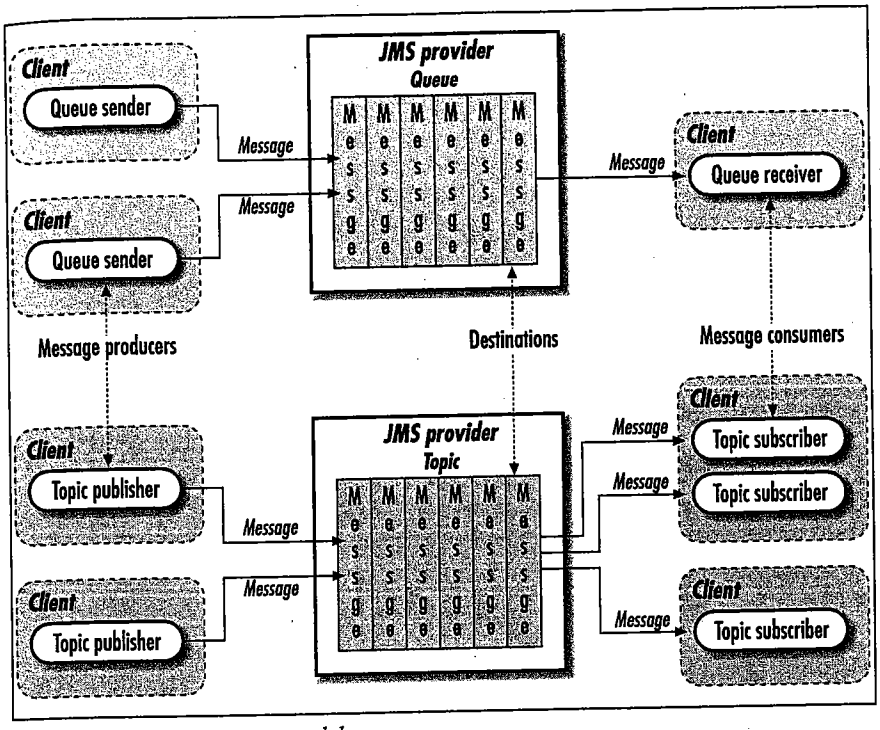


Figure 10-1: JMS message models

Most messaging providers support one or both of these messaging styles, so JMS provides support for them both in its API. JMS includes a set of generic messaging interfaces, described next. Each style of messaging is supported by specialized subclasses of these generic interfaces.

Key JMS Interfaces

The following key interfaces represent the concepts that come into play in any JMS client application, whether it is using point-to-point or publish-subscribe messaging. Information about all of the classes, interfaces, and exceptions in the JMS API can be found in Part III.

`Message`
: `Messages` are at the heart of JMS, naturally. `Messages` have accessor methods for their header fields, properties, and body contents. Subtypes of this interface provide implementations for different types of content.

`MessageListener`
: A `MessageListener` is attached to a `MessageConsumer` by a client, and receives a callback for each `Message` received by that consumer. `MessageListeners` are the key to asynchronous message delivery to clients, since the client attaches

a listener to a consumer and then carries on with its thread(s) of control. MessageListeners must implement an onMessage() method, which is the callback used to notify the listener that a message has arrived.

ConnectionFactory
: A ConnectionFactory creates connections to a JMS provider. ConnectionFactory references are obtained from a JMS provider through a JNDI lookup. A QueueConnectionFactory creates connections in a point-to-point context; TopicConnectionFactory creates connections in publish-subscribe contexts.

Destination
: A Destination represents a network location, managed by a JMS provider, that can be used to exchange messages. A JMS client sends messages to Destinations, and attaches MessageListeners to Destinations to receive messages from other clients. A client obtains references to Destinations using JNDI lookups. Queues and Topics are the Destinations in point-to-point and publish-subscribe contexts, respectively.

Connection
: A Connection is a live connection to a JMS provider, and is used for the receipt and delivery of messages. Before a client can exchange any messages with a JMS destination, it must have a live connection that has been started by the client. A Connection is obtained from a ConnectionFactory using its createXXXConnection() methods. The QueueConnectionFactory.createQueueConnection() methods return QueueConnections, and the TopicConnectionFactory.createTopicConnection() methods return TopicConnections.

Session
: A Session can be thought of as a single, serialized flow of messages between a client and a JMS provider. A Session is used to create message consumers and producers, and to create Messages that a client wishes to send. A Session is used within a single thread of control on a client. Since a Session is only accessed from within a single thread, the messages sent or received through its consumers and producers are serialized with respect to the client. Sessions also provide a context for defining transactions around message operations; details on transactional messaging can be found in the section Transactional Messaging. Sessions are created from Connections using their createXXXSession() methods. The QueueConnection.createQueueSession() method returns a QueueSession, and the TopicConnection.createTopicSession() method returns a TopicSession.

MessageProducer/MessageConsumer
: MessageProducers and MessageConsumers are used to send and receive messages from a destination, respectively. Producers and consumers are created using various createXXX() methods on Sessions, using the target Destination as the argument. In a point-to-point context, QueueSenders are created using the QueueSession.createSender() method, and QueueReceivers are created using the QueueSession.createReceiver() methods. In a publish-subscribe context, TopicPublishers are created using TopicSession.createPublisher(), and TopicSubscribers are created using TopicSession.createSubscriber() and TopicSession.createDurableSubscriber() methods.

A Generic JMS Client

A JMS client follows the same general
whether it's using point-to-point or pub
walk through these steps here, using the
strate. For the most part, the same pse
subscribe interfaces by just substituting "T
this section.

General setup

The very first step for a JMS client is to
the JNDI service of the JMS provider.
obtaining a JNDI Context can be found i
create an InitialContext using a set of
type of the JNDI service associated with

```
Properties props = ...;
Context ctx = new InitialContext(
```

Next, the JMS client needs to acquire a
using a JNDI lookup. The client would
used to publish the ConnectionFacto
QueueConnectionFactory registered in J

```
QueueConnectionFactory qFactory =
  (QueueConnectionFactory)ctx.loo
```

An administrator would have to set
provider and associate it with this JNDI

The client also uses JNDI to find Des
Here, we look up a Queue published u

```
Queue queue = (Queue)ctx.lookup(
```

Once we have a ConnectionFactory a
need to create a Connection with the J
through which messages will be physi
be started before messages can be
always be used to send messages, re
Normally, a client won't start() a
process messages. Here, we use
QueueConnection, and defer starting
receive messages:

```
QueueConnection qConn = qFactor
```

Client identifiers

When a client makes a connection t
ated with the client. The client ide
provider on behalf of the client, and

A Generic JMS Client

A JMS client follows the same general sequence of operations, regardless of whether it's using point-to-point or publish-subscribe messaging, or both. We'll walk through these steps here, using the point-to-point JMS interfaces to demonstrate. For the most part, the same pseudocode can be used with the publish-subscribe interfaces by just substituting "Topic" for "Queue" in the code samples in this section.

General setup

The very first step for a JMS client is to get a reference to an `InitialContext` for the JNDI service of the JMS provider. Full details on the various options for obtaining a JNDI `Context` can be found in Chapter 7, but in general, the client will create an `InitialContext` using a set of `Properties` that specify the location and type of the JNDI service associated with the JMS provider:

```
Properties props = ...;
Context ctx = new InitialContext(props);
```

Next, the JMS client needs to acquire a `ConnectionFactory` from the JMS provider using a JNDI lookup. The client would have to know what name the JMS provider used to publish the `ConnectionFactory` in JNDI space. Here, we lookup a `QueueConnectionFactory` registered in JNDI under the name "jms/someQFactory":

```
QueueConnectionFactory qFactory =
  (QueueConnectionFactory)ctx.lookup("jms/someQFactory");
```

An administrator would have to set up this `ConnectionFactory` on the JMS provider and associate it with this JNDI name on the server.

The client also uses JNDI to find `Destinations` published by the JMS provider. Here, we look up a `Queue` published under the JNDI name "jms/someQ":

```
Queue queue = (Queue)ctx.lookup("jms/someQ");
```

Once we have a `ConnectionFactory` and one or more `Destinations` to talk to, we need to create a `Connection` with the JMS provider. This `Connection` is the conduit through which messages will be physically sent and received. A `Connection` has to be started before messages can be received through it, but a `Connection` can always be used to send messages, regardless of whether it's started or stopped. Normally, a client won't `start()` a `Connection` until it's ready to receive and process messages. Here, we use our `QueueConnectionFactory` to create a `QueueConnection`, and defer starting it until we create a `MessageConsumer` to receive messages:

```
QueueConnection qConn = qFactory.createQueueConnection(...);
```

Client identifiers

When a client makes a connection to a JMS provider, a client identifier is associated with the client. The client identifier is used to maintain state on the JMS provider on behalf of the client, and the state data can persist beyond the lifetime

of a client connection. The server-side state can be retrieved for the client when it reconnects using the same client ID. The only client state information defined by the JMS specification is durable topic subscriptions (described in the section "Durable Subscriptions"), but a JMS provider may support its own state information on behalf of clients as well. Only one client is allowed to be associated with a client ID (and its state information) on the JMS provider, so only a single connection with a given client ID can be made to a JMS provider at any given time.

The JMS client identifier can be set in two ways. A client can set a client ID on any Connections that it makes with the JMS provider, using the Connection.setClientID() method:

```
qConn.setClientID("client-1");
```

Again, only a single connection with a given client ID is allowed at any given time. If a client with this same client ID (even this one) already has a connection with the client ID, then an InvalidClientIDException will be thrown when setClientID() is called. Alternatively, a ConnectionFactory can be configured on the JMS provider with a client ID that is applied to any Connections that are created through it. The ConnectionFactory interface doesn't provide a facility for the client to set the factory's client ID; this is a function that would have to be provided in the JMS provider's administrative interface. A ConnectionFactory with a preset client ID is, by definition, intended to be used by a single client.

Authenticated connections

When a client creates a connection, they have the option to provide a username and password that will be authenticated by the JMS provider. This is done using overloaded versions of the createXXXConnection() methods on a ConnectionFactory. We can create an authenticated QueueConnection, for example, with a call like this:

```
QueueConnection authQConn =
  qFactory.createQueueSession("JimFarley", "myJMSPassword");
```

If this is successful, the client will operate under the given principal name and be given the appropriate rights. JMS providers aren't required to support authentication of connections. If a JMS provider does support authenticated connections, the principals and access rights will be administered on the JMS server.

Sessions

Once a connection to the JMS provider is established, we need to create one or more Sessions to be used to send and receive messages. Again, Sessions are a single-threaded context for handling messages, so we need a separate Session for each concurrent thread that we plan to use for messaging. Sessions are created from Connections. Here, we create a QueueSession from our QueueConnection:

```
QueueSession qSess =
  qConn.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
```

When creating either QueueSessions or TopicSessions, there are two arguments used to create the Session. The first is a boolean flag indicating whether we want the Session to be transacted. (See the section "Transactional Messaging" for details

on transactional sessions.) The second
Session to acknowledge received messag
options for the acknowledge mode of
static final values on the Session class:

Session.AUTO_ACKNOWLEDGE
: This instructs the Session to ackn
client. A message is acknowledg
MessageListener handles the messa
until the listener's onMessage() me
because of a call to receive() on a
ment is sent immediately after the c

Session.DUPS_OK_ACKNOWLEDGE
: This option instructs the Sessior
acknowledgments can be delayed i
lead to a message being delivered
between delivery and acknowledg
timeout and it assumed the message

Session.CLIENT_ACKNOWLEDGE
: This option is used when the
messages, by calling the acknowled

Sending messages

Messages are sent to Destinations us
from Sessions. When they are cre
Destination, and any Messages sent
Destination using the Connection from

In a point-to-point context, the messa
from QueueSessions using the Queue the

```
QueueSender qSender = qSess.creat
```

Once a producer has been created,
Messages to be sent. Messages are also
TextMessage from our QueueSession, a
text we want to send to the Queue:

```
TextMessage tMsg = qSess.createT
tMsg.setText("The sky is blue.")
```

To actually send the message, we sim
MessageProducer. Here, we call send()

```
qSender.send(tMsg);
```

Note that it's not necessary to ensure
we generated our Session) is starte
Connection is only required to cc
Destination to the client.

:e can be retrieved for the client when it
only client state information defined by
ıubscriptions (described in the section
.der may support its own state informa-
client is allowed to be associated with a
ɜ JMS provider, so only a single connec-
a JMS provider at any given time.

ways. A client can set a client ID on any
JMS provider, using the Connection.

given client ID is allowed at any given
(even this one) already has a connection
ientIDException will be thrown when
:onnectionFactory can be configured on
is applied to any Connections that are
ry interface doesn't provide a facility for
his is a function that would have to be
.tive interface. A ConnectionFactory with
:d to be used by a single client.

have the option to provide a username
by the JMS provider. This is done using
ɔnnection() methods on a Connection-
d QueueConnection, for example, with a

```
Farley", "myJMSPassword");
```

e under the given principal name and be
ers aren't required to support authentica-
es support authenticated connections, the
ıistered on the JMS server.

is established, we need to create one or
receive messages. Again, Sessions are a
ısages, so we need a separate Session for
use for messaging. Sessions are created
ueSession from our QueueConnection:

```
Session.AUTO_ACKNOWLEDGE);
```

TopicSessions, there are two arguments
boolean flag indicating whether we want
ction "Transactional Messaging" for details

on transactional sessions.) The second argument indicates how we want the Session to acknowledge received messages with the JMS provider. There are three options for the acknowledge mode of a Session, and they are specified using static final values on the Session class:

`Session.AUTO_ACKNOWLEDGE`
: This instructs the Session to acknowledge messages automatically for the client. A message is acknowledged when received by the client. If a MessageListener handles the message, then the acknowledgment is not sent until the listener's onMessage() method returns. If the message is received because of a call to receive() on a MessageConsumer, then the acknowledgement is sent immediately after the call to receive() returns.

`Session.DUPS_OK_ACKNOWLEDGE`
: This option instructs the Session to do "lazy acknowledgment," where acknowledgments can be delayed if the Session decides to do so. This could lead to a message being delivered to a client more than once, if the delay between delivery and acknowledgment is longer than the JMS provider's timeout and it assumed the message was never received.

`Session.CLIENT_ACKNOWLEDGE`
: This option is used when the client wants to manually acknowledge messages, by calling the acknowledge() method on the Message.

Sending messages

Messages are sent to Destinations using MessageProducers, which are created from Sessions. When they are created, producers are associated with a Destination, and any Messages sent using the producer are delivered to that Destination using the Connection from which the Session was generated.

In a point-to-point context, the message producers are QueueSenders, generated from QueueSessions using the Queue the sender should point to:

```
QueueSender qSender = qSess.createSender(queue);
```

Once a producer has been created, the client needs to create and initialize Messages to be sent. Messages are also created from a Session. Here, we create a TextMessage from our QueueSession, and set its text body to be some interesting text we want to send to the Queue:

```
TextMessage tMsg = qSess.createTextMessage();
tMsg.setText("The sky is blue.");
```

To actually send the message, we simply invoke the appropriate method on our MessageProducer. Here, we call send() on our QueueSender:

```
qSender.send(tMsg);
```

Note that it's not necessary to ensure that the underlying Connection (from which we generated our Session) is started in order to send messages. Starting the Connection is only required to commence delivery of messages from the Destination to the client.

Receiving messages

Receiving messages involves creating a `MessageConsumer` that is associated with a particular `Destination`. This establishes a consumer with the JMS provider, and the provider is responsible for delivering any appropriate messages that arrive at the `Destination` to the new consumer. `MessageConsumers` are also generated from `Sessions`, in order to associate them with a serialized flow of messages. In a point-to-point context, a `QueueReceiver` is generated from a `QueueSession` using its `createReceiver()` methods. Here, we simply create a new receiver tied to our `Queue`. Other options for creating `QueueReceivers` are discussed in "Point-to-Point Messaging."

```
QueueReceiver qReceiver = qSess.createReceiver(queue);
```

By creating a `MessageConsumer`, all we've done is told the JMS provider that we want to receive messages from a particular `Destination`. We haven't specified what to do with the `Messages` on the client side. Since JMS is an asynchronous message delivery system, it uses the same listener pattern that is used in Swing GUI programming or JavaBeans event handling (two other asynchronous event contexts). `Messages` in JMS are processed using `MessageListeners`. A client needs to implement a `MessageListener` with an `onMessage()` method that does something useful with the `Messages` coming from the `Destination`. Example 10-1 shows a basic `MessageListener`—a `TextLogger` that simply prints the contents of any `TextMessages` it encounters.

Example 10-1: Simple MessageListener Implementation

```
import javax.jms.*;

public class TextLogger implements MessageListener {
  // Default constructor
  public TextLogger() {}

  // Message handler
  public void onMessage(Message msg) {
    // If it's a text message, print it to stdout
    if (msg instanceof TextMessage) {
      TextMessage tMsg = (TextMessage)msg;
      try {
        System.out.println("Received message: " + tMsg.getText());
      }
      catch (JMSException je) {
        System.out.println("Error retrieving message text: " +
                           je.getMessage());
      }
    }
    // For other types of messages, print an error
    else {
      System.out.println("Unsupported message type encountered.");
    }
  }
}
```

Once a `MessageListener` has been defin
register it with a `MessageConsumer`. In our
`TextLoggers` and associate it wi
`setMessageListener()` method:

```
MessageListener listener = new Text
qReceiver.setMessageListener(listen
```

It's important to remember that no mes
lying `Connection` until it's been started.
`QueueConnection` but never started it, s
`Messages` to our `QueueReceiver`, and from

```
qConn.start();
```

Temporary destinations

A client can create its own temporary d
are visible only to the `Connection` that c
tion of the `Connection` used to create the
only for the life of the `Connection` it v
methods on the `Session`. For example, to

```
Queue tempQueue = qSession.createT
```

Temporary destinations can be used,
messages that are sent with a `JMSReplyT`

```
TextMessage request = qSession.cre
request.setJMSReplyTo(tempQueue);
```

They can also be used to exchanging a
the same client.

Cleaning up

`Connections` and `Sessions` require resc
(similar to how JDBC connections use
idea to free them up explicitly when yo

Sessions are closed by simply calling cl

```
qSess.close();
```

When a `Session` is closed, all `Message(`
with it are rendered unusable. If you tr
provider, they will throw an `IllegalS`
will block until any pending pro
`MessageListener's onMessage()` metho

Closing a `Session` doesn't close the u
You can close one `Session` and open
active. To close a `Connection` and free
method:

```
qConn.close();
```

Once a MessageListener has been defined, the client needs to create one and register it with a MessageConsumer. In our running example, we create one of our TextLoggers and associate it with our QueueReceiver using its setMessageListener() method:

```
MessageListener listener = new TextLogger();
qReceiver.setMessageListener(listener);
```

It's important to remember that no messages will be delivered over our underlying Connection until it's been started. In our running example, we created our QueueConnection but never started it, so we do that now to start delivery of Messages to our QueueReceiver, and from there to our TextLogger listener:

```
qConn.start();
```

Temporary destinations

A client can create its own temporary destinations, which are Destinations that are visible only to the Connection that created it, and that only live for the duration of the Connection used to create them. Although a temporary destination lives only for the life of the Connection it was created from, they are created using methods on the Session. For example, to create a TemporaryQueue:

```
Queue tempQueue = qSession.createTemporaryQueue();
```

Temporary destinations can be used, for example, to receive responses to messages that are sent with a JMSReplyTo header;

```
TextMessage request = qSession.createTextMessage();
request.setJMSReplyTo(tempQueue);
```

They can also be used to exchanging asynchronous messages between threads in the same client.

Cleaning up

Connections and Sessions require resources to be allocated by the JMS provider (similar to how JDBC connections use up resources on a RDBMS), so it's a good idea to free them up explicitly when you are done with them.

Sessions are closed by simply calling close() on them:

```
qSess.close();
```

When a Session is closed, all MessageConsumers and MessageProducers associated with it are rendered unusable. If you try to use them to communicate with the JMS provider, they will throw an IllegalStateException. A call to Session.close() will block until any pending processing of incoming Messages (e.g., a MessageListener's onMessage() method) is complete.

Closing a Session doesn't close the underlying Connection from which it came. You can close one Session and open up another one as long as the Connection is active. To close a Connection and free up its server-side resources, call its close() method:

```
qConn.close();
```

All Sessions (and, subsequently, all of their consumers and producers) generated from a Connection become unusable once it is closed. The call to Connection.close() will block until incoming Message processing has completed on all of the Sessions associated with it.

The Anatomy of Messages

Creating a messaging-based application involves more than establishing communication channels between participants. Players in a message-driven system need to understand the content of the messages and know what to do with them.

Native messaging systems, such as IBM MQ Series or Microsoft MQ (MSMQ), define their own proprietary formats for messages. JMS attempts to bridge these native messaging systems by defining its own standard message format. All JMS clients can interact with any messaging system that supports JMS. "Supports" in this case can mean one of two things. The messaging system can be implemented in a native, proprietary architecture, with a JMS bridge that maps the JMS message formats (and other aspects of the JMS specification) to the native scheme and back again. Or, the messaging system can be written to use the JMS message format as its native format.

JMS messages are made up of a set of standard header fields, optional client-defined properties, and a body. JMS also provides a set of subclasses of Message that support various types of message bodies.

Message Header Fields and Properties

Table 10-1 lists the standard header fields that any JMS message can have. The table indicates the name and type of the field, when the field is set in the message delivery process, and a short description of the semantics of the field.

Table 10-1: Standard JMS Message Headers

Field Name	*Data Type*	*When Set*	*Description*
JMSCorrelationID	String	Before send	Correlates multiple messages. This field can be used in addition to the JMSMessageID header as an application-defined message identifier (JMSMessageIDs are assigned by the provider).
JMSDestination	Destination	During send	Indicates to the message receiver which Destination the Message was sent to.
JMSDeliveryMode	int	During send	Indicates which delivery mode to use to deliver this message, DeliveryMode.PERSISTENT or DeliveryMode.NON_PERSISTENT. PERSISTENT delivery indicates that the messaging provider should take measures to ensure that the message is delivered despite failures on the JMS server. NON_PERISTENT delivery doesn't require the provider to deliver the message if a failure occurs on the JMS server.

Table 10-1: Standard JMS Message Heade

Field Name	*Data Type*	*Whe*
JMSExpiration	long	Duri send
JMSMessageID	String	Dur sen
JMSPriority	int	Dur sen
JMSRedelivered	boolean	Bef del
JMSReplyTo	Destination	Be ser
JMSTimestamp	long	Du ser
JMSType	String	Be se

These standard message headers are sors on the Message interface. The J setJMSTimestamp(), and read using ge

ıeir consumers and producers) generated
ıce it is closed. The call to Connection.
je processing has completed on all of the

nvolves more than establishing communi-
ıyers in a message-driven system need to
nd know what to do with them.

1 MQ Series or Microsoft MQ (MSMQ),
· messages. JMS attempts to bridge these
s own standard message format. All JMS
system that supports JMS. "Supports" in
ıe messaging system can be implemented
a JMS bridge that maps the JMS message
ecification) to the native scheme and back
written to use the JMS message format as

f standard header fields, optional client-
ɔ provides a set of subclasses of Message
dies.

ɔperties

ds that any JMS message can have. The
field, when the field is set in the message
of the semantics of the field.

rs

n Set	Description
re	Correlates multiple messages. This field can be used in addition to the JMSMessageID header as an application-defined message identifier (JMSMessageIDs are assigned by the provider).
ıg	Indicates to the message receiver which Destination the Message was sent to.
ıg	Indicates which delivery mode to use to deliver this message, DeliveryMode.PERSISTENT or DeliveryMode.NON_PERSISTENT. PERSISTENT delivery indicates that the messaging provider should take measures to ensure that the message is delivered despite failures on the JMS server. NON_PERISTENT delivery doesn't require the provider to deliver the message if a failure occurs on the JMS server.

Table 10-1: Standard JMS Message Headers (continued)

Field Name	Data Type	When Set	Description
JMSExpiration	long	During send	The time the message will expire on the provider. If no client receives the message by this time, the provider drops the message. It is calculated as the sum of the current time plus the time-to-live of the MessageProducer that sent the message. The value is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT). A value of zero indicates no expiration time.
JMSMessageID	String	During send	A unique message ID assigned by the provider. Message Ids always start with the prefix "ID:". These IDs are unique for a given JMS provider. Applications can set their own message identifier using the JMSCorrelationID header.
JMSPriority	int	During send	A provider-assigned value indicating the priority with which the message will be delivered. JMS providers aren't required to implement strict priority ordering of messages. This field is simply a "hint" from the server about how the message will be handled. Message priorities and how they are assigned are determined by the configuration of the JMS provider.
JMSRedelivered	boolean	Before delivery	A provider-provided value that indicates to the receiver that the message may have been delivered in the past with no acknowledgment from the client. On the sender, this header value is always unassigned.
JMSReplyTo	Destination	Before send	A Destination, set by the sending client, indicating where a reply message should be sent.
JMSTimestamp	long	During send	The time at which the message was handed off to the JMS provider to be sent. This value is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).
JMSType	String	Before send	A message type, set by the sending client. Some JMS providers require that this header be set, so it's a good idea to set it even if your application isn't using it. Some JMS providers also allow an administrator to configure a set of message types that will be matched against this header, and used to selectively set handling of the message based on its type.

These standard message headers are read and written using corresponding accessors on the Message interface. The JMSTimestamp field, for example, is set using setJMSTimestamp(), and read using getJMSTimestamp().

A client can also create its own custom properties on a Message, using a set of generic property accessors on the Message interface. Custom message properties can be boolean, byte, short, int, long, float, double, or String values, and they are accessed using corresponding get/setXXXProperty() methods on Message. A boolean header can be set using the setBooleanProperty() method, for example. Each custom property has to be given a unique name, specified when the value is set. For example, we could set a custom boolean property with the name "reviewed" on a message, like so:

```
TextMessage tMsg = ...;
tMsg.setBooleanProperty("reviewed", false);
```

Custom property names have certain restrictions on them. They have to be valid Java identifiers, they can't begin with "JMSX" or "JMS_" (these are reserved for JMS-defined and vendor-defined properties, respectively), and they can't be one of the following reserved words: NULL, TRUE, FALSE, NOT, AND, OR, BETWEEN, LIKE, IN, IS, or ESCAPE. Custom property names are also case-sensitive, so "reviewed" isn't the same property as "Reviewed."

JMS Message Types

To support various application scenarios, JMS provides the following subclasses of Message, each providing a different type of message body.

TextMessage
: Arguably the most popular type of Message, this has a body that is a simple String. The format of the String contents is left to the application to interpret. The String may contain a simple informational phrase, it may contain conversational text input by a user in a collaboration application, or it may contain formatted text such as XML.

BytesMessage
: This type of message contains an array of bytes as its body. BytesMessages can be used to send binary data in a message, and/or it can be used to wrap a native message format with a JMS message.

ObjectMessage
: The body of this message is a serialized Java object.

MapMessage
: The body of this message is a set of name/value pairs. The names are Strings, and the values are Java primitive types, like double, int, String, etc. The values of the entries are accessed using get/setXXX() methods on MapMessage. Note that these entries are stored in the body of the message: they aren't message header properties and can't be used for message selection (see the section "Filtering Messages").

StreamMessage
: A StreamMessage contains a stream of Java primitive data types (double, int, String, etc.). Data elements are written sequentially to the body of the message using various writeXXX() methods, and they are read sequentially on the receiving end using corresponding readXXX() methods. If the receiver doesn't know the types of data in the : the readObject() method:

```
StreamMessage sMsg = ...;
Object item = sMsg.readObject() .
if (item instanceof Float) {
  float fData = ((Float)item).f
  ...
}
```

Accessing Message Content

When a client receives a Message, its bod body of a received Message will cause thrown.

When a message sender first creates a Me unset. For TextMessages and ObjectMes: with a null value, and for MapMessages message body. For BytesMessages and St only mode, and they can't be read by the method. If you create a BytesMessage o body content before calling reset() on it thrown.

A Message's body can be emptied by clearBody() method. This reverts its b affect any of the header or property val on a BytesMessage or StreamMessage pu until a subsequent call to BytesMessag made. The sender can clear any custor clearProperties() method. The standa using their specific accessors on the Mes call clearBody(), clearProperties(), the Message is read-only at this point.

Filtering Messages

JMS allows messaging participants to s from a JMS provider. This is done using that filter messages based on the values ties. The syntax of message selectors is b

A message selector is associated with a Session. Each type of Session (QueueS versions of their consumer create me String argument. For example, the receives only messages that have a cust whose JMSType header field is acknowle

```
String selector = "JMSType = 'ack

QueueReceiver receiver = qSessior
```

properties on a Message, using a set of
ge interface. Custom message properties
loat, double, or String values, and they
tXXXProperty() methods on Message. A
ooleanProperty() method, for example.
nique name, specified when the value is
tom boolean property with the name

false);

rictions on them. They have to be valid
X" or "JMS_" (these are reserved for JMS-
spectively), and they can't be one of the
SE, NOT, AND, OR, BETWEEN, LIKE, IN, IS, or
) case-sensitive, so "reviewed" isn't the

JMS provides the following subclasses of
f message body.

essage, this has a body that is a simple
ntents is left to the application to inter-
ole informational phrase, it may contain
n a collaboration application, or it may

ay of bytes as its body. BytesMessages
message, and/or it can be used to wrap
essage.

d Java object.

of name/value pairs. The names are
itive types, like double, int, String, etc.
ssed using get/setXXX() methods on
ıre stored in the body of the message:
s and can't be used for message selec-
es").

f Java primitive data types (double, int,
itten sequentially to the body of the
thods, and they are read sequentially on
ng readXXX() methods. If the receiver

doesn't know the types of data in the message, they can be introspected using the readObject() method:

```
StreamMessage sMsg = ...;
Object item = sMsg.readObject();
if (item instanceof Float) {
  float fData = ((Float)item).floatValue()l
  ...
}
```


Accessing Message Content

When a client receives a Message, its body is read-only. Attempting to change the body of a received Message will cause a MessageNotWriteableException to be thrown.

When a message sender first creates a Message object, the body of the Message is unset. For TextMessages and ObjectMessages, this means that their body starts with a null value, and for MapMessages this means there are no entries in the message body. For BytesMessages and StreamMessages, their bodies start in write-only mode, and they can't be read by the sending client until it calls their reset() method. If you create a BytesMessage or StreamMessage and attempt to read its body content before calling reset() on it, a MessageNotReadableException will be thrown.

A Message's body can be emptied by the sender at any point by calling its clearBody() method. This reverts its body back to its initial state, but doesn't affect any of the header or property values on the Message. Calling clearBody() on a BytesMessage or StreamMessage puts their body back into write-only mode, until a subsequent call to BytesMessage.reset() or StreamMessage.reset() is made. The sender can clear any custom properties on a Message by calling its clearProperties() method. The standard JMS header fields have to be updated using their specific accessors on the Message interface. Receivers of messages can't call clearBody(), clearProperties(), or reset() on a received Message, since the Message is read-only at this point.

Filtering Messages

JMS allows messaging participants to selectively filter what Messages it receives from a JMS provider. This is done using message selectors, which are expressions that filter messages based on the values found in their headers and custom properties. The syntax of message selectors is based on SQL92 conditional expressions.

A message selector is associated with a MessageConsumer when it is created from a Session. Each type of Session (QueueSession and TopicSession) has overloaded versions of their consumer create methods that take a message selector as a String argument. For example, the following creates a QueueReceiver that receives only messages that have a custom property named transaction-type and whose JMSType header field is acknowledge:

```
String selector = "JMSType = 'acknowledge'
                                    AND transaction-type IS NOT NULL";
QueueReceiver receiver = qSession.createReceiver(queue, selector);
```

Message filtering is performed by the JMS provider. When the provider determines that a message should be delivered to a particular `MessageConsumer`, based on the rules of the particular message context (point-to-point or publish-subscribe), it first checks that consumer's message selector, if one exists. If the selector evaluates to `true` when the message's headers and properties are applied to it, then the message is delivered; otherwise it isn't. Undelivered messages are handled differently, depending on the message context, as described in the following sections.

Point-to-Point Messaging

Point-to-point messaging involves the sending of messages from one or more senders to a single receiver through a message queue. Point-to-point messaging is analogous to email messaging: a client delivers a message to a named mailbox (queue), and the owner of the mailbox (queue) reads them in the order they were received. Queues attempt to maintain the send order of messages generated by the sender(s) attached to them. In other words, if sender A sends messages A1, A2, and A3, in that order, to a queue, then the receiver attached to the queue will receive message 2 after message 1, and message 3 after message 2 (assuming that no message selectors filter out any of these messages). If there are multiple senders attached to a queue, then the relative order of each individual sender is preserved by the queue when it delivers the messages, but the queue doesn't attempt to impose a predefined absolute order on the messages across all senders. So if there is another sender, B, attached to the same queue as A, and it sends messages B1, B2, and B3, in that order, then the receiver will receive B2 after B1, and B3 after B2, but the messages from sender A may be interleaved with the messages from sender B. The receiver may receive the messages in order A1, A2, B1, A3, B2, B3, the messages may be delivered in order B1, B2, B3, A1, A2, A3, or some other order altogether. There is nothing in the JMS specification that dictates how a JMS provider should queue messages from multiple senders.

Point-to-point messaging is performed in JMS using the queue-related interfaces and classes in the `javax.jms` package. Queues are represented by `Queue` objects, which are looked up in JNDI from the JMS provider. `QueueConnectionFactory` objects are looked up in JNDI as well, and used to create `QueueConnections`. `QueueConnections` and `Queues` are used to create `QueueSessions`, which are in turn used to create `QueueSenders` and `QueueReceivers`.

Sample Client

Example 10-2 shows a full point-to-point messaging client. The `PTPMessagingClient` is capable of sending and receiving a message from a given queue, as well as browsing the current contents of the queue.

Example 10-2: Point-to-Point Messaging Client

```
import java.util.*;
import javax.naming.*;
import javax.jms.*;
import java.io.*;
```

Example 10-2: Point-to-Point Messaging Clien

```
public class PTPMessagingClient implements

  // Our connection to the JMS provider.
  private QueueConnection mQueueConn = nu

  // The queue used for message-passing
  private Queue mQueue = null;

  // Our message receiver - only need one
  private QueueReceiver mReceiver = null;

  // A single session for sending and rec
  private QueueSession mSession = null;

  // The message type we tag all our mes
  private static String MSG_TYPE = "Java

  // Constructor, with client name, and
  // connection factory and queue that w
  public PTPMessagingClient(String cFact
    init(cFactoryJNDIName, queueJNDIName
  }

  // Do all the JMS-setup for this clie
  // configured (perhaps using jndi.pro
  // InitialContext points to the JMS p
  protected boolean init(String cFactor
    boolean success = true;

    Context ctx = null;
    // Attempt to make connection to JN
    try {
      ctx = new InitialContext();
    }
    catch (NamingException ne) {
      System.out.println("Failed to co
      ne.printStackTrace();
      success = false;
    }

    // If no JNDI context, bail out h
    if (ctx == null) {
      return success;
    }

    // Attempt to lookup JMS connecti
    QueueConnectionFactory connFactor
    try {
      connFactory = (QueueConnectionI
      System.out.println("Got JMS co
    }
    catch (NamingException ne2) {
```

provider. When the provider deter-
a particular MessageConsumer, based
context (point-to-point or publish-
ıessage selector, if one exists. If the
's headers and properties are applied
se it isn't. Undelivered messages are
ssage context, as described in the

ng of messages from one or more
ge queue. Point-to-point messaging is
/ers a message to a named mailbox
ıe) reads them in the order they were
d order of messages generated by the
if sender A sends messages A1, A2,
receiver attached to the queue will
age 3 after message 2 (assuming that
se messages). If there are multiple
'e order of each individual sender is
ıe messages, but the queue doesn't
r on the messages across all senders.
the same queue as A, and it sends
the receiver will receive B2 after B1,
ıder A may be interleaved with the
ceive the messages in order A1, A2,
d in order B1, B2, B3, A1, A2, A3, or
in the JMS specification that dictates
om multiple senders.

S using the queue-related interfaces
es are represented by Queue objects,
S provider. QueueConnectionFactory
d used to create QueueConnections.
ate QueueSessions, which are in turn
ers.

-point messaging client. The
d receiving a message from a given
ts of the queue.

Example 10-2: Point-to-Point Messaging Client (continued)

```
public class PTPMessagingClient implements Runnable {

  // Our connection to the JMS provider.  Only one is needed for this client.
  private QueueConnection mQueueConn = null;

  // The queue used for message-passing
  private Queue mQueue = null;

  // Our message receiver - only need one.
  private QueueReceiver mReceiver = null;

  // A single session for sending and receiving from all remote peers.
  private QueueSession mSession = null;

  // The message type we tag all our messages with
  private static String MSG_TYPE = "JavaEntMessage";

  // Constructor, with client name, and the JNDI locations of the JMS
  // connection factory and queue that we want to use.
  public PTPMessagingClient(String cFactoryJNDIName, String queueJNDIName) {
    init(cFactoryJNDIName, queueJNDIName);
  }

  // Do all the JMS-setup for this client.  Assumes that the JVM is
  // configured (perhaps using jndi.properties) so that the default JNDI
  // InitialContext points to the JMS provider's JNDI service.
  protected boolean init(String cFactoryJNDIName, String queueJNDIName) {
    boolean success = true;

    Context ctx = null;
    // Attempt to make connection to JNDI service
    try {
      ctx = new InitialContext();
    }
    catch (NamingException ne) {
      System.out.println("Failed to connect to JNDI provider:");
      ne.printStackTrace();
      success = false;
    }

    // If no JNDI context, bail out here
    if (ctx == null) {
      return success;
    }

    // Attempt to lookup JMS connection factory from JNDI service
    QueueConnectionFactory connFactory = null;
    try {
      connFactory = (QueueConnectionFactory)ctx.lookup(cFactoryJNDIName);
      System.out.println("Got JMS connection factory.");
    }
    catch (NamingException ne2) {
```

Example 10-2: Point-to-Point Messaging Client (continued)

```
      System.out.println("Failed to get JMS connection factory: ");
      ne2.printStackTrace();
      success = false;
    }

    try {
      // Make a connection to the JMS provider and keep it.
      // At this point, the connection is not started, so we aren't
      // receiving any messages.
      mQueueConn = connFactory.createQueueConnection();
      // Try to find our designated queue
      mQueue = (Queue)ctx.lookup(queueJNDIName);
      // Make a session for queueing messages: no transactions,
      // auto-acknowledge
      mSession =
        mQueueConn.createQueueSession(false,
                                      javax.jms.Session.AUTO_ACKNOWLEDGE);
    }
    catch (JMSException e) {
      System.out.println("Failed to establish connection/queue:");
      e.printStackTrace();
      success = false;
    }
    catch (NamingException ne) {
      System.out.println("JNDI Error looking up factory or queue:");
      ne.printStackTrace();
      success = false;
    }

    try {
      // Make our receiver, for incoming messages.
      // Set the message selector to only receive our type of messages,
      // in case the same queue is being used for other purposes.
      mReceiver = mSession.createReceiver(mQueue,
                                          "JMSType = '" + MSG_TYPE + "'");
    }
    catch (JMSException je) {
      System.out.println("Error establishing message receiver:");
      je.printStackTrace();
    }

    return success;
  }

  // Send a message to the queue
  public void sendMessage(String msg) {
    try {
      // Create a JMS msg sender connected to the destination queue
      QueueSender sender = mSession.createSender(mQueue);
      // Use the session to create a text message
      TextMessage tMsg = mSession.createTextMessage();
      tMsg.setJMSType(MSG_TYPE);
      // Set the body of the message
```


Example 10-2: Point-to-Point Messaging Clien

```
      tMsg.setText(msg);
      // Send the message using the sender
      sender.send(tMsg);
      System.out.println("Sent the message
    }
    catch (JMSException je) {
      System.out.println("Error sending me
      je.printStackTrace();
    }
  }

  // Register a MessageListener with the c
  // messages asynchronously
  public void registerListener(MessageLis
    try {
      // Set the listener on the receiver
      mReceiver.setMessageListener(listen
      // Start the connection, in case it
      mQueueConn.start();
    }
    catch (JMSException je) {
      System.out.println("Error registeri
      je.printStackTrace();
    }
  }

  // Perform an synchronous receive of a
  // TextMessage, print the contents.
  public String receiveMessage() {
    String msg = "-- No message --";
    try {
      Message m = mReceiver.receive();
      if (m instanceof TextMessage) {
        msg = ((TextMessage)m).getText()
      }
      else {
        msg = "-- Unsupported message ty
      }
    }
    catch (JMSException je) {
    }
    return msg;
  }

  // Print the current contents of the
  // so that we don't remove any messag
  public void printQueue() {
    try {
      QueueBrowser browser = mSession.c
      Enumeration msgEnum = browser.get
      System.out.println("Queue content
      while (msgEnum.hasMoreElements()
```

ent (continued)

```
JMS connection factory: ");

ovider and keep it.
s not started, so we aren't

ueConnection();
e
DIName);
sages: no transactions,

se,
vax.jms.Session.AUTO_ACKNOWLEDGE);

blish connection/queue:");

ing up factory or queue:");

messages.
receive our type of messages,
used for other purposes.
(mQueue,
 "JMSType = '" + MSG_TYPE + "'");

ing message receiver:");

d to the destination queue
eSender(mQueue);
 message
extMessage();
```

Example 10-2: Point-to-Point Messaging Client (continued)

```
      tMsg.setText(msg);
      // Send the message using the sender
      sender.send(tMsg);
      System.out.println("Sent the message");
    }
    catch (JMSException je) {
      System.out.println("Error sending message " + msg + " to queue");
      je.printStackTrace();
    }
  }

  // Register a MessageListener with the queue to receive
  // messages asynchronously
  public void registerListener(MessageListener listener) {
    try {
      // Set the listener on the receiver
      mReceiver.setMessageListener(listener);
      // Start the connection, in case it's still stopped
      mQueueConn.start();
    }
    catch (JMSException je) {
      System.out.println("Error registering listener: ");
      je.printStackTrace();
    }
  }

  // Perform an synchronous receive of a message from the queue.  If it's a
  // TextMessage, print the contents.
  public String receiveMessage() {
    String msg = "-- No message --";
    try {
      Message m = mReceiver.receive();
      if (m instanceof TextMessage) {
        msg = ((TextMessage)m).getText();
      }
      else {
        msg = "-- Unsupported message type received --";
      }
    }
    catch (JMSException je) {
    }
    return msg;
  }

  // Print the current contents of the message queue, using a QueueBrowser
  // so that we don't remove any messages from the queue
  public void printQueue() {
    try {
      QueueBrowser browser = mSession.createBrowser(mQueue);
      Enumeration msgEnum = browser.getEnumeration();
      System.out.println("Queue contents:");
      while (msgEnum.hasMoreElements()) {
```

Example 10-2: Point-to-Point Messaging Client (continued)

```
        System.out.println("\t" + (Message)msgEnum.nextElement());
      }
    }
    catch (JMSException je) {
      System.out.println("Error browsing queue: " + je.getMessage());
    }
  }

  // When run within a thread, just wait for messages to be delivered to us
  public void run() {
    while (true) {
      try { this.wait(); } catch (Exception we) {}
    }
  }

  // Take command-line arguments and send or receive messages from the
  // named queue
  public static void main(String args[]) {
    if (args.length < 3) {
      System.out.println("Usage: PTPMessagingClient" +
                         " connFactoryName queueName" +
                         " [send|listen|recv_synch] <messageToSend>");
      System.exit(1);
    }

    // Get the JNDI names of the connection factory and
    // queue, from the command-line
    String factoryName = args[0];
    String queueName = args[1];

    // Get the command to execute (send, recv, recv_synch)
    String cmd = args[2];

    // Create and initialize the messaging participant
    PTPMessagingClient msger =
      new PTPMessagingClient(factoryName, queueName);

    // Run the participant in its own thread, so that it can react to
    // incoming messages
    Thread listen = new Thread(msger);
    listen.start();

    // Send a message to the queue
    if (cmd.equals("send")) {
      String msg = args[3];
      msger.sendMessage(msg);
      System.exit(0);
    }
    // Register a listener
    else if (cmd.equals("listen")) {
```


Example 10-2: Point-to-Point Messaging

```
      MessageListener listener = new T
      msger.registerListener(listener)
      System.out.println("Client liste
                         + "...");
      System.out.flush();
      try { listen.wait(); } catch (Ex
    }
    // Synchronously receive a message
    else if (cmd.equals("recv_synch"))
      String msg = msger.receiveMessag
      System.out.println("Received mes
      System.exit(0);
    }
    else if (cmd.equals("browse")) {
      msger.printQueue();
      System.exit(0);
    }
  }
}
```

The `main()` method takes a minimum o
two are the JNDI names of a target J
order. The third argument is a command

- *send* sends a message, using the ne
  `TextMessage`.
- *recv* registers a listener with the que
- *recv_synch* synchronously polls the
- *browse* is a request to print the cur
  ing it, using a `QueueBrowser`.

The `main()` method creates a `PTPMessag`
constructor passes these to the `init()`
we've discussed takes place. The clien
and get its `InitialContext` first. Th
`InitialContext` constructor, so the envi
ties specified in a *jndi.properties* file, o
the JVM. Once the `Context` is
`QueueConnectionFactory` and `Queue` fro
and a `QueueSession`, so that it can lat
Finally, the `init()` method creates a Q
needed later. The connection hasn't
receiving messages from the JMS provid

Back in the `main()` method, once the
run. This is useful for the case where
listener. Finally, the requested commar
call the client's `sendMessage()` meth
`TextMessage` (using the last command-
method). Then the message is sent b

Client (continued)

```
sage)msgEnum.nextElement());

ng queue: " + je.getMessage());

it for messages to be delivered to us

)tion we) {}

)nd or receive messages from the

) {

sagingClient" +
ame queueName" +
[recv_synch] <messageToSend>");

tion factory and

, recv, recv_synch)

ing participant

e, queueName);

)read, so that it can react to
```

Example 10-2: Point-to-Point Messaging Client (continued)

```
      MessageListener listener = new TextLogger();
      msger.registerListener(listener);
      System.out.println("Client listening to queue " + queueName
                         + "...");
      System.out.flush();
      try { listen.wait(); } catch (Exception we) {}
    }
    // Synchronously receive a message from the queue
    else if (cmd.equals("recv_synch")) {
      String msg = msger.receiveMessage();
      System.out.println("Received message: " + msg);
      System.exit(0);
    }
    else if (cmd.equals("browse")) {
      msger.printQueue();
      System.exit(0);
    }
  }
}
```

The `main()` method takes a minimum of three command-line arguments. The first two are the JNDI names of a target JMS connection factory and queue, in that order. The third argument is a command indicating what to do:

- *send* sends a message, using the next command-line argument as the text of a `TextMessage`.
- *recv* registers a listener with the queue and waits for messages to come in.
- *recv_synch* synchronously polls the queue for the next message that's sent.
- *browse* is a request to print the current contents of the queue without emptying it, using a `QueueBrowser`.

The `main()` method creates a `PTPMessagingClient` using the two JNDI names. The constructor passes these to the `init()` method, where all of the JMS initialization we've discussed takes place. The client attempts to connect to its JNDI provider and get its `InitialContext` first. There are no properties provided to the `InitialContext` constructor, so the environment would have to have these properties specified in a *jndi.properties* file, or on the command line using `-D` options to the JVM. Once the `Context` is acquired, the client looks up the `QueueConnectionFactory` and `Queue` from JNDI. It also creates a `QueueConnection` and a `QueueSession`, so that it can later create senders and receivers as needed. Finally, the `init()` method creates a `QueueReceiver` from the session, in case it's needed later. The connection hasn't been started yet, so the receiver is not receiving messages from the JMS provider yet.

Back in the `main()` method, once the client is created, it's put into a `Thread` and run. This is useful for the case where we're going to wait for messages sent to a listener. Finally, the requested command is checked. If the command is *send*, we call the client's `sendMessage()` method, which creates a `QueueSender` and a `TextMessage` (using the last command-line argument, passed in from the `main()` method). Then the message is sent by passing it to the send() method on the

QueueSender. If a "recv" command is given, we create a TextLogger (see Example 10-1) and attach it as a MessageListener to our QueueReceiver, by calling the client's registerListener() method where the call to the receiver's setMessageListener() method is made. If a *recv_synch* command is given, then we call the client's receiveMessage() method, where the receive() method on the QueueReceiver is called. This will block until the next message is sent to the queue. Finally, a *browse* command causes a call to the client's printQueue() method, where a QueueBrowser is created from our session, then asked for an Enumeration of the current messages in the queue. Each message is printed to the console, in the order they would be received.

Browsing Queues

In addition to the conventional use of queues for sending and receiving of messages, a client can also browse the contents of a queue without actually pulling the messages from the queue. This is done using a QueueBrowser, which is generated from a client's QueueSession using its createQueueBrowser() methods:

```
QueueBrowser browser = qSession.createQueueBrowser(queue);
```

Like QueueReceivers, QueueBrowsers can use message selectors to filter what messages they see in the queue:

```
QueueBrowser filterBrowser =
  qSession.createQueueBrowser(queue, "transaction-type = 'update'");
```

This QueueBrowser "sees" only messages in the queue that have a transaction-type property set to update.

To iterate over the messages in the queue, a client asks the browser for an Enumeration of the messages in the queue that match the browser's message selector, if it has one:

```
Enumeration msgEnum = browser.getEnumeration();
while (msgEnum.hasMoreElements()) {
  Message msg = (Message)msgEnum.nextElement();
  System.out.println("Found message, ID = " + msg.getJMSMessageID());
}
```

The Enumeration returns messages in the order that they would be delivered to the client, using the message selector set on the QueueBrowser. So if you had an existing QueueReceiver and wanted to look ahead in the queue to see what messages would be delivered based on the current contents of the queue, you could create a browser using the same message selector as the receiver:

```
QueueReceiver recvr = ...;
QueueBrowser recvrBrowser =
  qSession.createQueueBrowser(queue, recvr.getMessageSelector());
```

Publish-Subscribe Messaging

Publish-subscribe messaging involves one or more MessageProducers "publishing" messages to a particular topic, and one or more MessageConsumers "subscribing" to the topic and receiving any messages published to it. The JMS provider is responsible for delivering a copy of any message sent to a topic to all subscribers of the

topic at the time that the message is re
where messages are kept on the queue u
received at a topic while a subscriber is
topic yet, or subscribed and then went ou
to that subscriber.

Publish-subscribe messaging is performec
and classes in the javax.jms package.
which are looked up in JNDI from the
objects are looked up in JNDI as well
TopicConnections and Topics are used t
used to create TopicPublishers and Topi

Sample Client

Example 10-3 shows a publish-subscri
mirrors the PTPMessagingClient in Exa
the client is virtually identical to that des
that topics, subscribers, and publishers a
senders. The only significant differenc
"browse" option, since browsing a top
deliver their messages to any subscribe
wise they are dropped, so browsing a to

Example 10-3: Publish-Subscribe Client

```
import java.util.*;
import javax.naming.*;
import javax.jms.*;
import java.io.*;

public class PubSubMessagingClient imp

  // Our connection to the JMS provid
  private TopicConnection mTopicConn =

  // The topic used for message-passi
  private Topic mTopic = null;

  // Our message subscriber - only ne
  private TopicSubscriber mSubscriber

  // A single session for sending and
  private TopicSession mSession = nul

  // The message type we tag all our
  private static String MSG_TYPE = "

  // Constructor, with client name, a
  // connection factory and topic tha
  public PubSubMessagingClient(Strin
    init(cFactJNDIName, topicJNDIName
  }
```

given, we create a TextLogger (see
stener to our QueueReceiver, by calling
l where the call to the receiver's
a *recv_synch* command is given, then
hod, where the receive() method on
k until the next message is sent to the
:s a call to the client's printQueue()
from our session, then asked for an
queue. Each message is printed to the
d.

queues for sending and receiving of
contents of a queue without actually
is done using a QueueBrowser, which is
g its createQueueBrowser() methods:

```
eQueueBrowser(queue);
```

use message selectors to filter what

```
"transaction-type = 'update'");
```

ı the queue that have a transaction-

ıe, a client asks the browser for an
ıe that match the browser's message

```
ɛration();
```

```
:lement();
ID = " + msg.getJMSMessageID());
```

ɔrder that they would be delivered to
ı the QueueBrowser. So if you had an
ɔk ahead in the queue to see what
ɛ current contents of the queue, you
ıge selector as the receiver:

```
ecvr.getMessageSelector());
```

more MessageProducers "publishing"
ɔre MessageConsumers "subscribing" to
ıed to it. The JMS provider is respon-
ɛnt to a topic to all subscribers of the

topic at the time that the message is received. Unlike point-to-point messaging, where messages are kept on the queue until a receiver reads them, any messages received at a topic while a subscriber is not active (e.g., hasn't subscribed to the topic yet, or subscribed and then went out of scope or exited) are lost with respect to that subscriber.

Publish-subscribe messaging is performed in JMS using the topic-related interfaces and classes in the javax.jms package. Topics are represented by Topic objects, which are looked up in JNDI from the JMS provider. TopicConnectionFactory objects are looked up in JNDI as well, and used to create TopicConnections. TopicConnections and Topics are used to create TopicSessions, which are in turn used to create TopicPublishers and TopicSubscribers.

Sample Client

Example 10-3 shows a publish-subscribe client, PubSubMessagingClient, that mirrors the PTPMessagingClient in Example 10-2. The structure and function of the client is virtually identical to that described for the PTPMessagingClient, except that topics, subscribers, and publishers are used instead of queues, receivers, and senders. The only significant difference with this client is it doesn't have a "browse" option, since browsing a topic is not possible. As they arrive, topics deliver their messages to any subscribers currently attached to the topic, otherwise they are dropped, so browsing a topic's contents doesn't make much sense.

Example 10-3: Publish-Subscribe Client

```
import java.util.*;
import javax.naming.*;
import javax.jms.*;
import java.io.*;

public class PubSubMessagingClient implements Runnable {

  // Our connection to the JMS provider.  Only one is needed for this client.
  private TopicConnection mTopicConn = null;

  // The topic used for message-passing
  private Topic mTopic = null;

  // Our message subscriber - only need one.
  private TopicSubscriber mSubscriber = null;

  // A single session for sending and receiving from all remote peers.
  private TopicSession mSession = null;

  // The message type we tag all our messages with
  private static String MSG_TYPE = "JavaEntMessage";

  // Constructor, with client name, and the JNDI location of the JMS
  // connection factory and topic that we want to use.
  public PubSubMessagingClient(String cFactJNDIName, String topicJNDIName) {
    init(cFactJNDIName, topicJNDIName);
  }
```

Example 10-3: Publish-Subscribe Client (continued)

```
// Do all the JMS-setup for this client.  Assumes that the JVM is
// configured (perhaps using jndi.properties) so that the default JNDI
// InitialContext points to the JMS provider's JNDI service.
protected boolean init(String cFactoryJNDIName, String topicJNDIName) {
  boolean success = true;

  Context ctx = null;

  // Attempt to make connection to JNDI service
  try {
    ctx = new InitialContext();
  }
  catch (NamingException ne) {
    System.out.println("Failed to connect to JNDI provider:");
    ne.printStackTrace();
    success = false;
  }

  // If no JNDI context, bail out here
  if (ctx == null) {
    return success;
  }

  // Attempt to lookup JMS connection factory from JNDI service
  TopicConnectionFactory connFactory = null;
  try {
    connFactory = (TopicConnectionFactory)ctx.lookup(cFactoryJNDIName);
    System.out.println("Got JMS connection factory.");
  }
  catch (NamingException ne2) {
    System.out.println("Failed to get JMS connection factory: ");
    ne2.printStackTrace();
    success = false;
  }

  try {
    // Make a connection to the JMS provider and keep it
    // At this point, the connection is not started, so we aren't
    // receiving any messages.
    mTopicConn = connFactory.createTopicConnection();
    // Try to find our designated topic
    mTopic = (Topic)ctx.lookup(topicJNDIName);
    // Make a session for topicing messages
    // no transactions, auto-acknowledge
    mSession =
      mTopicConn.createTopicSession(false,
                                    javax.jms.Session.AUTO_ACKNOWLEDGE);
  }
  catch (JMSException e) {
    System.out.println("Failed to establish connection/topic:");
```


Example 10-3: Publish-Subscribe Client (

```
    e.printStackTrace();
    success = false;
  }
  catch (NamingException ne) {
    System.out.println("JNDI Error l
    ne.printStackTrace();
    success = false;
  }

  try {
    // Make our subscriber, for inc
    // Set the message selector to
    // in case the same topic is be
    // Also indicate we don't want
    mSubscriber =
      mSession.createSubscriber(
                                mTopic
  }
  catch (JMSException je) {
    System.out.println("Error estat
    je.printStackTrace();
  }

  return success;
}

// Send a message to the topic
public void publishMessage(String
  try {
    // Create a JMS msg publisher
    TopicPublisher publisher = mSe
    // Use the session to create a
    TextMessage tMsg = mSession.cr
    tMsg.setJMSType(MSG_TYPE);
    // Set the body of the message
    tMsg.setText(msg);
    // Send the message using the
    publisher.publish(tMsg);
    System.out.println("Published
  }
  catch (JMSException je) {
    System.out.println("Error sen
    je.printStackTrace();
  }
}

// Register a MessageListener wit
// messages asynchronously
public void registerListener(Mess
  try {
    // Set the listener on the s
```

Example 10-3: Publish-Subscribe Client (continued)

```
    e.printStackTrace();
    success = false;
  }
  catch (NamingException ne) {
    System.out.println("JNDI Error looking up factory or topic:");
    ne.printStackTrace();
    success = false;
  }

  try {
    // Make our subscriber, for incoming messages
    // Set the message selector to only receive our type of messages,
    // in case the same topic is being used for other purposes
    // Also indicate we don't want any message sent from this connection
    mSubscriber =
      mSession.createSubscriber(
                              mTopic, "JMSType = '" + MSG_TYPE + "'", true);
  }
  catch (JMSException je) {
    System.out.println("Error establishing message subscriber:");
    je.printStackTrace();
  }

  return success;
}

// Send a message to the topic
public void publishMessage(String msg) {
  try {
    // Create a JMS msg publisher connected to the destination topic
    TopicPublisher publisher = mSession.createPublisher(mTopic);
    // Use the session to create a text message
    TextMessage tMsg = mSession.createTextMessage();
    tMsg.setJMSType(MSG_TYPE);
    // Set the body of the message
    tMsg.setText(msg);
    // Send the message using the publisher
    publisher.publish(tMsg);
    System.out.println("Published the message");
  }
  catch (JMSException je) {
    System.out.println("Error sending message " + msg + " to topic");
    je.printStackTrace();
  }
}

// Register a MessageListener with the topic to receive
// messages asynchronously
public void registerListener(MessageListener listener) {
  try {
    // Set the listener on the subscriber
```

Example 10-3: Publish-Subscribe Client (continued)

```
      mSubscriber.setMessageListener(listener);
      // Start the connection, in case it's still stopped
      mTopicConn.start();
    }
    catch (JMSException je) {
      System.out.println("Error registering listener: ");
      je.printStackTrace();
    }
  }

  // Perform an synchronous receive of a message from the topic.  If it's a
  // TextMessage, print the contents.
  public String receiveMessage() {
    String msg = "-- No message --";
    try {
      Message m = mSubscriber.receive();
      if (m instanceof TextMessage) {
        msg = ((TextMessage)m).getText();
      }
      else {
        msg = "-- Unsupported message type received --";
      }
    }
    catch (JMSException je) {
    }
    return msg;
  }

  // When run within a thread, just wait for messages to be delivered to us
  public void run() {
    while (true) {
      try { this.wait(); } catch (Exception we) {}
    }
  }

  // Take command-line arguments and send or receive messages from the
  // named topic
  public static void main(String args[]) {
    if (args.length < 3) {
      System.out.println("Usage: PubSubMessagingClient" +
                         " connFactoryName topicName" +
                         " [publish|subscribe|recv_synch] <messageToSend>");
      System.exit(1);
    }

    // Get our client name, and the JNDI name of the connection factory and
    // topic, from the command-line
    String factoryName = args[0];
    String topicName = args[1];

    // Get the command to execute (publish, subscribe, recv_synch)
    String cmd = args[2];
```


Example 10-3: Publish-Subscribe Client (con

```
    // Create and initialize the messagir
    PubSubMessagingClient msger =
      new PubSubMessagingClient(factoryN

    // Run the participant in its own th
    // incoming messages
    Thread listen = new Thread(msger);
    listen.start();

    // Send a message to the topic
    if (cmd.equals("publish")) {
      String msg = args[3];
      msger.publishMessage(msg);
      System.exit(0);
    }
    // Register a listener
    else if (cmd.equals("subscribe")) {
      MessageListener listener = new Te
      msger.registerListener(listener);
      System.out.println("Client listen
                         + "...");
      try { listen.wait(); } catch (Exc
    }
    // Synchronously receive a message
    else if (cmd.equals("recv_synch"))
      String msg = msger.receiveMessage
      System.out.println("Received mess
      System.exit(0);
    }
  }
}
```

Durable Subscriptions

If a client needs to guarantee delivery c
time of a single subscriber, it can regi
provider for the target Topic. A durable
createDurableSubscriber() methods o
durable subscriber is created by specifyi

```
TopicConnection tConn = ...;
tConn.setClientID("client-1");
TopicSession tSession =
  tConn.createTopicSession(false,
TopicSubscriber durableSub =
  tSession.createDurableSubscribe
```

This registers a durable subscription tc
Durable subscriptions and their names
client ID of the client that created then
for details on client IDs). Here, we're
setClientID() on the TopicConnectior

Example 10-3: Publish-Subscribe Client (continued)

```
    // Create and initialize the messaging participant
    PubSubMessagingClient msger =
      new PubSubMessagingClient(factoryName, topicName);

    // Run the participant in its own thread, so that it can react to
    // incoming messages
    Thread listen = new Thread(msger);
    listen.start();

    // Send a message to the topic
    if (cmd.equals("publish")) {
      String msg = args[3];
      msger.publishMessage(msg);
      System.exit(0);
    }
    // Register a listener
    else if (cmd.equals("subscribe")) {
      MessageListener listener = new TextLogger();
      msger.registerListener(listener);
      System.out.println("Client listening to topic " + topicName
                         + "...");
      try { listen.wait(); } catch (Exception we) {}
    }
    // Synchronously receive a message from the topic
    else if (cmd.equals("recv_synch")) {
      String msg = msger.receiveMessage();
      System.out.println("Received message: " + msg);
      System.exit(0);
    }
  }
}
```

Durable Subscriptions

If a client needs to guarantee delivery of messages from a `Topic` beyond the lifetime of a single subscriber, it can register a durable subscription with the JMS provider for the target `Topic`. A durable subscription to a `Topic` is made using the `createDurableSubscriber()` methods on a `TopicSession`. In its simplest form, a durable subscriber is created by specifying a `Topic` and a subscriber name:

```
TopicConnection tConn = ...;
tConn.setClientID("client-1");
TopicSession tSession =
  tConn.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
TopicSubscriber durableSub =
  tSession.createDurableSubscriber(topic, "subscriber-1");
```

This registers a durable subscription to the `Topic` under the name "subscriber-1." Durable subscriptions and their names are associated by the JMS provider with the client ID of the client that created them (see the earlier section "Client identifiers," for details on client IDs). Here, we're setting our client ID to "client-1" by calling `setClientID()` on the `TopicConnection`.

As long as this TopicSubscriber is live, it will receive any messages published to the Topic, as it would if it were a nondurable subscriber. But if the subscriber dies (goes out of scope, or the client dies), the JMS provider will retain messages on behalf of the named subscriber (based on its client identifier), until another durable subscriber attaches to the topic from the same connection using the same client ID and specifying the same subscriber name. Any pending messages will be delivered to the new subscriber when it attaches.

It's important to remember that durable subscriptions can be a costly resource on the JMS provider. The provider will have to create database records, or otherwise allocate server resources, in order to preserve the subscription information and any pending messages for the client. If there are many durable subscriptions, or if the number of pending messages being held on the server for subscribers becomes large, this can eventually have a significant impact on the performance of the JMS provider. So durable subscriptions should be used with discretion.

Transactional Messaging

JMS supports transactional messaging in two ways. In its simplest form, a Session is created with the transactional option (the first argument to QueueConnection.createQueueSession() and TopicConnection.createTopicSession()).

```
QueueSession xactSession =
  qConn.createQueueSession(TRUE, Session.AUTO_ACKNOWLEDGE);
```

When using a transactional Session, the client performs a series of "transactions" with the Session (sends and/or receives messages from consumers and producers associated with the Session). These sends and receives are either committed by calling the Session's commit() method, or cancelled by calling rollback(). If a Session is committed, all of the sends and receives are committed to the JMS provider, which causes the new state of the Destination(s) affected to be committed. If a Session is rolled back, all changes to the resources on the JMS provider are rolled back. In either case, the transaction is closed and a new one is started automatically, for any subsequent messaging actions.

JMS providers can also support transactional messaging through the Java Transaction API (JTA), which allows messaging transactions to be integrated with other resources, like databases. These JTA-based transactions are distributed: the underlying transactional resources can be distributed across the enterprise. The JMS API supports this form of transactional messaging with a set of interfaces that provide access to JTA-aware Connections and Sessions. If a JMS provider supports JTA, it can export an XAConnectionFactory in its JNDI space. An XAConnectionFactory is used to create XAConnections, and XAConnections are used to create XASessions. An XASession is a specialization of Session that overloads the commit() and rollback() methods to implement them within a JTA context. There are subclasses of these XA interfaces for point-to-point and publish-subscribe messaging. For example, to create a JTA-aware TopicSession:

```
XATopicConnectionFactory xFactory =
  (XATopicConnectionFactory)ctx.lookup("xact-factory");
XATopicConnection xConn = xFactory.createXATopicConnection();
XATopicSession xSession = xConn.createXATopicSession();
```

When a client performs a series of sends actions are performed in the context of exists. For example, if we use our XA (xPublisher) and a TopicSubscriber (x transaction and use it to commit or roll

```
javax.transaction.UserTransaction
xaction.start();
try {
  Message request = ...;
  Message response = ...;
  xPublisher.publish(request);
  response = xSubscriber.receive
  // Made it here, so commit the
  xaction.commit();
}
catch (JMSException je) {
  // Something bad happened, so
  // by our message sends/receiv
  xaction.rollback();
}
```

Message Selector Syntax

JMS message selectors are used by J server delivers to a given Message (optionally) when a MessageConsume createReceiver() or the TopicSessic

A message selector is a string that message the provider wants to delive uates to true, the message is delive point-to-point messaging, when mes the message remains in the queue message times out, and the server re messaging, messages that are filtered

Structure of a Selector

A message selector is made up c together by logical operators and gr

```
(<expression1> OR <expression2
(<expression4> AND NOT <expre
```

A message selector is evaluated l example, expression2 is evaluated precedence than OR), and if they ev

Expressions are made up of literal headers or properties), conditional

When a client performs a series of sends/receives with a JTA-aware Session, these actions are performed in the context of the surrounding UserTransaction, if one exists. For example, if we use our XATopicSession to create a TopicPublisher (xPublisher) and a TopicSubscriber (xSubscriber), we can create our own JTA transaction and use it to commit or roll back a series of message operations:

```
javax.transaction.UserTransaction xaction = ...;
xaction.start();
try {
  Message request = ...;
  Message response = ...;
  xPublisher.publish(request);
  response = xSubscriber.receive();
  // Made it here, so commit the topic changes
  xaction.commit();
}
catch (JMSException je) {
  // Something bad happened, so cancel the topic changes caused
  // by our message sends/receives
  xaction.rollback();
}
```

Message Selector Syntax

JMS message selectors are used by JMS clients to filter the messages that a JMS server delivers to a given MessageConsumer. A message selector is provided (optionally) when a MessageConsumer is created, using either the QueueSession.createReceiver() or the TopicSession.createSubscriber() methods.

A message selector is a string that specifies a predicate to be applied to each message the provider wants to deliver to a MessageConsumer. If the predicate evaluates to true, the message is delivered; if false, the message isn't delivered. In point-to-point messaging, when messages are filtered out by a message selector, the message remains in the queue until the client eventually reads it, or the message times out, and the server removes it from the queue. In publish-subscribe messaging, messages that are filtered are never delivered to the subscriber.

Structure of a Selector

A message selector is made up of one or more boolean expressions, joined together by logical operators and grouped using parentheses. For example:

```
(<expression1> OR <expression2> AND <expression3>) OR
(<expression4> AND NOT <expression5>) ...
```

A message selector is evaluated left to right in precedence order. So in this example, expression2 is evaluated followed by expression3 (since AND has higher precedence than OR), and if they evaluate to false, expression1 is evaluated, etc.

Expressions are made up of literal values, identifiers (referring to either message headers or properties), conditional operators, and arithmetic operators.

Identifiers

An identifier refers to either a standard JMS header field name or a custom message property name. Any JMS header field name can be used as an identifier, except for `JMSDestination`, `JMSExpiration`, `JMSRedelivered`, and `JMSReplyTo`, which can be used as identifiers in a message selector. `JMSDestination` and `JMSReplyTo` are `Destination` values, and message selector operators support only numeric, boolean or string values. The time at which message selectors are applied to messages isn't specified in the JMS specification, so using the value of `JMSExpiration` doesn't provide a consistent, well-defined result. Using `JMSRedelivered` in a selector can result in unexpected results. If, for example, a selector checks for `JMSRedelivered` being `true`, the first delivery attempt by a provider will fail the selector because the redelivered flag should be `false`, but the provider can then immediately redeliver the message and pass the selector, making the redelivered part of the predicate ineffective.

Identifier names are case-sensitive and follow the same general rules as Java identifiers. They must start with a valid Java identifier start character as determined by the `java.lang.Character.isJavaIdentifierStart()` method. For example, a letter, currency symbol, or connecting punctuation character such as an underscore _ contain valid Java identifier characters as determined by the `Character.isJavaIdentifierPart()` method. You can't use these reserved words for identifiers: `NULL`, `TRUE`, `FALSE`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, `IS`, or `ESCAPE`.

The type of an identifier is the type of the header field or property being referenced as its value is set in the message. It's important to remember that the evaluation of an identifier in a message selector doesn't apply type conversion functions according to the context in which it's used. If you attempt to refer to a numeric property value in an expression with a string comparison operator, for example, the expression always evaluates to `false`. If the named header field or property isn't present in a message, the identifier evaluates to a `null` value.

Literals

String literals are indicated with single quotes. For example:

```
JMSType = 'updateAck'
```

If you need to use a single quote in a string literal, use two single quotes:

```
JMSCorrelationID = 'Joe''s message'
```

Numeric literals are either integer values or floating-point values. Integer values follow the rules for Java integer literals. They are all numerals, with no decimal point, and can have a value in the same range as a Java `long` value:

```
42, 149, -273
```

Floating-point literals follow the syntax of Java floating-point literals. They are numerals with a decimal point:

```
3.14, 98.6, -273.0
```

They are also in scientific notation:

```
31.4e-1, 6.022e23, 2.998e8
```

Literals can also be the boolean values `tr`

Operators

Operators compose identifiers and litera
be logical operators, arithmetic operators,

Logical Operators

The logical operators are `NOT`, `AND`, and
have the usual boolean logic semantics.
fields or properties whose value is `null`,

- `AND`ing a `null` value with a `false` va
  a `true` or `null` value evaluates to a
- `OR`ing a `null` value with a `true` val
  `false` or a `null` value evaluates to a
- Applying `NOT` to a `null` value evalua

Arithmetic Operators

The arithmetic operators, in precedence
- (binary). These have the usual arithm
is applied to one or more `null` values e

Comparison Operators

The comparison operators can be loos
range comparisons. The basic equality
are =, >, >=, <, <=, and < >. These bi
values of the same type, else the expr
value is `null`, the result of the comp
operators `IS NULL` and `IS NOT NULL` t
check for the presence of a header or p

```
timezone IS NOT NULL AND country
```

There are also set and range compa
check the range of numeric values:

```
userid BETWEEN 00000000 AND 0999
currRate NOT BETWEEN 0.0 AND 0.9
```

The `IN` operator can set memberships

```
JMSType IN ('msgAck', 'queryAck'
JMSType NOT IN ('msgBroadcast',
```

There is also a string comparison oper
on string values. A pattern is used f

MS header field name or a custom
ld name can be used as an identifier,
ı, JMSRedelivered, and JMSReplyTo,
essage selector. JMSDestination and
ssage selector operators support only
me at which message selectors are
IS specification, so using the value of
sistent, well-defined result. Using
inexpected results. If, for example, a
true, the first delivery attempt by a
elivered flag should be false, but the
he message and pass the selector,
neffective.

low the same general rules as Java
/a identifier start character as deter-
valIdentifierStart() method. For
ecting punctuation character such as
er characters as determined by the
You can't use these reserved words
, BETWEEN, LIKE, IN, IS, or ESCAPE.

header field or property being refer-
It's important to remember that the
ector doesn't apply type conversion
it's used. If you attempt to refer to a
th a string comparison operator, for
false. If the named header field or
fier evaluates to a null value.

For example:

eral, use two single quotes:

floating-point values. Integer values
y are all numerals, with no decimal
as a Java long value:

ava floating-point literals. They are

They are also in scientific notation:

```
31.4e-1, 6.022e23, 2.998e8
```

Literals can also be the boolean values true or false.

Operators

Operators compose identifiers and literals into larger expressions. Operators can be logical operators, arithmetic operators, or comparison operators.

Logical Operators

The logical operators are NOT, AND, and OR. These are in precedence order. These have the usual boolean logic semantics. If a logical operator is applied to header fields or properties whose value is null, then the following rules apply:

- ANDing a null value with a false value evaluates to false; ANDing a null with a true or null value evaluates to a null (or unknown) value.
- ORing a null value with a true value evaluates to true; ORing a null with a false or a null value evaluates to a null (or unknown) value.
- Applying NOT to a null value evaluates to a null (or unknown) value.

Arithmetic Operators

The arithmetic operators, in precedence order, are + and - (unary), * and /, + and - (binary). These have the usual arithmetic semantics. Any arithmetic operator that is applied to one or more null values evaluates to a null value.

Comparison Operators

The comparison operators can be loosely grouped into equality comparisons and range comparisons. The basic equality comparison operators, in precedence order, are =, >, >=, <, <=, and < >. These binary operators have to be applied to two values of the same type, else the expression always evaluates to false. If either value is null, the result of the comparison is null. There are also the equality operators IS NULL and IS NOT NULL to compare a value to null. This can also check for the presence of a header or property:

```
timezone IS NOT NULL AND country = 'United Kingdom'
```

There are also set and range comparison operators. The BETWEEN operator can check the range of numeric values:

```
userid BETWEEN 00000000 AND 09999999
currRate NOT BETWEEN 0.0 AND 0.9999
```

The IN operator can set memberships operations on string values:

```
JMSType IN ('msgAck', 'queryAck', 'updateAck')
JMSType NOT IN ('msgBroadcast', 'synchMessage')
```

There is also a string comparison operator, LIKE, that allows for wildcard matching on string values. A pattern is used for the right side of the LIKE operator. The

pattern consists of a valid string literal in which the underscore character matches against any single character, and the % character matches any sequence of zero or more characters. For example:

```
JMSType like '%Ack'
label not like 'Step _'
```

The _ and % characters can be used in these string comparison operators if they are escaped by a backslash \:

```
slogan LIKE '99 44/100\% pure'
```

Expressions

Expressions are simply literals and identifiers assembled together using the various operators described earlier. A message selector must eventually evaluate to a boolean value, so its combination of expressions must be structured to result in a boolean value. Expressions can be grouped in a message selector using parentheses in order to control the order of evaluation.

Arithmetic expressions are composed of arithmetic operators used with numeric literals and identifier values. Arithmetic expressions can be combined to form compound arithmetic expressions:

```
(userid + 10000) / (callerid - 10000)
```

Conditional expressions are made up of comparison and logical operators used with numeric, string or boolean literals or identifiers, and evaluate to `true`, `false` or `null` (i.e., unknown). Conditional expressions can also be combined to form compound conditional expressions:

```
(JMSType like '%Ack') AND ((userid + 10000) / (callerid - 10000) < 1.0)
```

Notice that, although the last example includes an arithmetic expression fragment:

```
(userid + 10000) / (callerid - 1000)
```

it becomes part of a conditional expression when used with a comparison operator with the numeric literal 1.0.

Every complete message selector must be a conditional expression. A message selector that evaluates to `true` matches the message; one that evaluates to `false` or `null` doesn't match the message.

CHAP

Jav

The JavaMail APIs provide a platform
build Java-based mail and messaging
Internet email. JavaMail can be used
This includes such applications as ma
primary application for JavaMail migl
existing and new applications. For e
implement a web-based mail reading
tions, receive commands from users, o

Sun Microsystems, Inc. included a basi
JDK. However, because this class cou
with Sun's implementation of the JDI
advanced electronic mail capability int
from scratch. The JavaMail API, first i
1.2, fills this niche, giving Java ap
framework.

Frankly speaking, the JavaMail API i
classes provide an interface to a gen
storing, and transporting messages. D
ment custom transport protocols and
an add-on, the distribution include
protocols and a set of helper classes
the Internet. Also, since JavaMail
Internet Mail Extensions (MIME)* for
extremely rich, opening up a range o

* MIME was originally defined in RFCs 152
through 2049. The complete set, along w
links, is available at *http://www.nacs.uc*