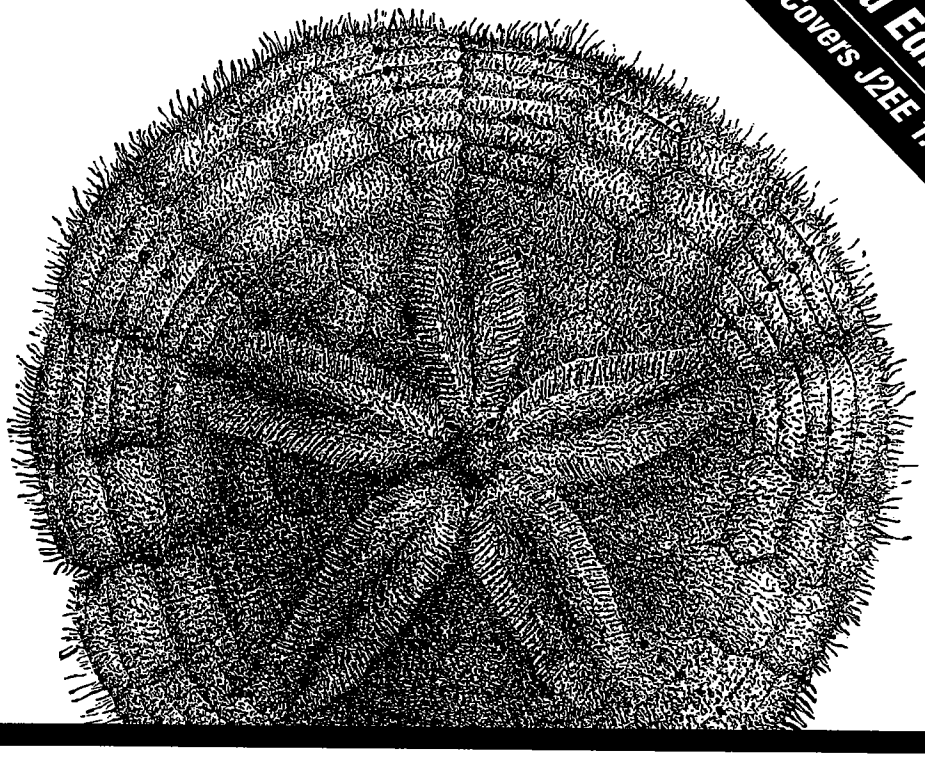
2nd Edition
Covers J2EE 1.3



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.

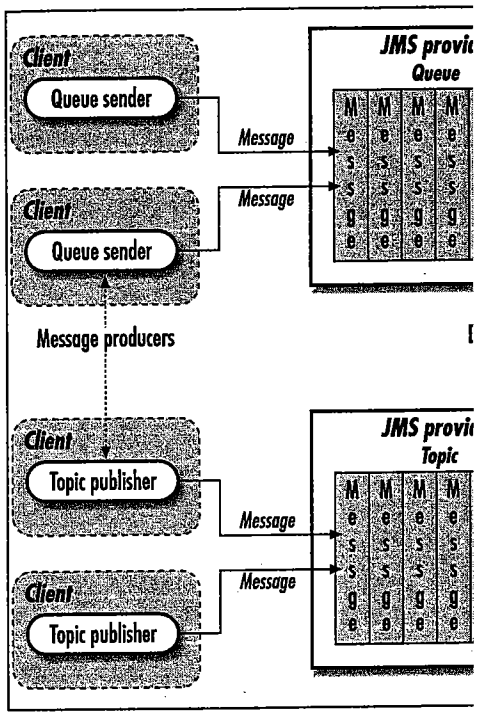


Figure 10-1: JMS message models

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.

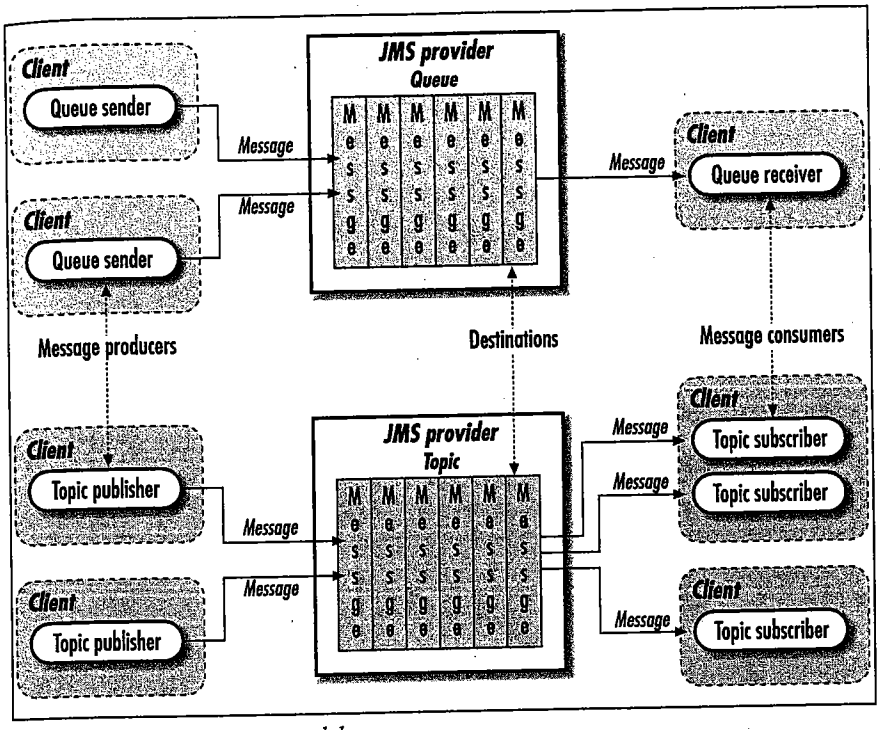


Figure 10-1: JMS message models

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