Introducing the Java Message Service

Willy Farrell

June 08, 2004

This tutorial provides an introductory overview of the Java Message Service, its functionality, and its capabilities. You will learn the basic programming techniques for creating JMS programs and work with sample code that demonstrates these techniques. Note: This tutorial has been updated to include changes to JMS version 1.1.

Getting started

Should I take this tutorial?

This tutorial provides an overview of the Java Message Service (JMS) and offers the basics for developing programs that use it. JMS was developed by Sun Microsystems to provide a way for Java programs to access an enterprise messaging system, also known as message oriented middleware (MOM). MOM provides a mechanism for integrating applications in a loosely coupled, flexible manner by providing asynchronous delivery of data between applications in an indirect way through an intermediary.

Before taking this tutorial you should be familiar with Java programming and object-oriented programming concepts.

To write the programs described in this tutorial, you need an editing environment. It can be as basic as an operating system editor. In a development context, many people use an integrated development environment (IDE) because it possesses debuggers and other features designed specifically for writing and testing code.

To compile the programs, you'll need the Java compiler (javac.exe). You will also need the JMS classes in the package javax.jms and the Java Naming and Directory Interface (JNDI) classes in the package javax.naming. You can download these from Sun: JMS and JNDI.

To execute and test the programs, you need access to a vendor implementation of JMS. Most Java 2 Enterprise Edition (J2EE) vendors provide an implementation of JMS. See your vendor documentation for setting up the JMS runtime and executing programs.

Some of your programming decisions will depend on the JMS version in use (see New in JMS 1.1: Unified domains). JMS version 1.1 supports Point-to-point interfaces and Pub/sub interfaces only.
for backward compatibility, and you can safely skip the sections of this tutorial on those topics if you are doing strictly new development using a provider implementation based on JMS version 1.1.

**Introduction**

**Enterprise messaging systems**

The Java Message Service was developed by Sun Microsystems to provide a means for Java programs to access enterprise messaging systems. Before we discuss JMS, let’s take a look at enterprise messaging systems.

Enterprise messaging systems, often known as message oriented middleware (MOM), provide a mechanism for integrating applications in a loosely coupled, flexible manner. They provide asynchronous delivery of data between applications on a store and forward basis; that is, the applications do not communicate directly with each other, but instead communicate with the MOM, which acts as an intermediary.

The MOM provides assured delivery of messages (or at least makes its best effort) and relieves application programmers from knowing the details of remote procedure calls (RPC) and networking/communications protocols.

**Messaging flexibility**

As shown in the figure below, Application A communicates with Application B by sending a message through the MOM’s application programming interface (API).

![Diagram showing communication between Application A and Application B through the MOM](image)

The MOM routes the message to Application B, which can exist on a completely different computer; the MOM handles the network communications. If the network connection is not available, the MOM will store the message until the connection becomes available, and then forward it to Application B.

Another aspect of flexibility is that Application B might not even be executing when Application A sends its message. The MOM will hold the message until Application B begins execution and attempts to retrieve its messages. This also prevents Application A from blocking while it waits for Application B to receive the message.

This asynchronous communication requires applications to be designed somewhat differently from the way most are designed today, but it can be an extremely useful method for time-independent or parallel processing.
Loose coupling

The real power of enterprise messaging systems lies in the *loose coupling* of the applications. In the diagram on the previous section, Application A sends its messages indicating a particular destination, for example "order processing." Today, Application B provides order-processing capabilities.

But, in the future, we can replace Application B with a different order-processing program, and Application A will be none the wiser. It will continue to send its messages to "order processing" and the messages will continue to be processed.

Likewise, we could replace Application A, and as long as the replacement continued to send messages for "order processing," the order-processing program would not need to know there is a new application sending orders.

Publish and subscribe

Originally, enterprise messaging systems were developed to implement a *point-to-point model* (PTP) in which each message produced by an application is received by one other application. In recent years, a new model has emerged, called *publish and subscribe* (or pub/sub).

Pub/sub replaces the single destination in the PTP model with a content hierarchy, known as *topics*. Sending applications *publish* their messages, indicating that the message represents information about a topic in the hierarchy.

Applications wishing to receive those messages *subscribe* to that topic. Subscribing to a topic in the hierarchy that contains subtopics allows the subscriber to receive all messages published to the topic and its subtopics.

This figure illustrates the publish and subscribe model.
Multiple applications can both subscribe and publish messages to a topic, and the applications remain anonymous to one another. The MOM acts as a broker, routing the published messages for a topic to all subscribers for that topic.

What is JMS?

The Java Message Service specification 1.1 states:

JMS is a set of interfaces and associated semantics that define how a JMS client accesses the facilities of an enterprise messaging product.

Prior to JMS, each MOM vendor provided application access to its product through a proprietary API, often available in multiple languages, including the Java language. JMS provides a standard, portable way for Java programs to send and receive messages through a MOM product. Programs written with JMS can run on any MOM that implements the JMS standard.

The key to JMS portability is the fact that the JMS API is provided by Sun as a set of interfaces. Products that provide JMS functionality do so by supplying a provider that implements these interfaces.

As a developer, you build a JMS application by defining a set of messages and a set of client applications that exchange those messages.

JMS objectives

To better understand JMS, it helps to know the objectives set by the authors of the JMS specification.

There are many enterprise messaging products on the market today, and several of the companies that produce these products were involved in the development of JMS.

These existing systems vary in capability and functionality. The authors knew that JMS would be too complicated and unwieldy if it incorporated all of the features of all existing systems. Likewise, they believed that they could not limit themselves to only the features that all of the systems had in common.

The authors believed that it was important that JMS include all of the functionality required to implement "sophisticated enterprise applications."

The objectives of JMS, as stated in the specification, are to:

- Define a common set of messaging concepts and facilities.
- Minimize the concepts a programmer must learn to use enterprise messaging.
- Maximize the portability of messaging applications.
- Minimize the work needed to implement a provider.
- Provide client interfaces for both point-to-point and pub/sub domains. "Domains" is the JMS term for the messaging models discussed earlier. (Note: A provider need not implement both domains.)
New in JMS 1.1: Unified domains

In versions of JMS prior to version 1.1, each domain has its own set of client interfaces specific to that domain. JMS version 1.1 provides a single set of interfaces that allows clients to send and receive messages in both domains. These "domain-independent interfaces" retain the semantics and behavior of each domain and are the preferred choice for implementing JMS clients. The domain-specific interfaces are supported for backward compatibility and should not be used for new development.

The benefits of unifying the domains are:

- It presents simpler programming model for client programming.
- Actions on both queues and topics can be part of the same transaction.
- It provides opportunities for JMS providers to optimize their implementations.

What JMS does not provide

The following features, common in MOM products, are not addressed by the JMS specification. Although acknowledged by the JMS authors as important for the development of robust messaging applications, these features are considered JMS provider-specific.

JMS providers are free to implement these features in any manner they please, if at all:

- Load balancing and fault tolerance
- Error and advisory system messages and notification
- Administration
- Security
- Wire protocol
- Message type repository

JMS overview and architecture

Applications

A JMS application comprises the following elements:

- **JMS clients.** Java programs that send and receive messages using the JMS API.
- **Non-JMS clients.** It is important to realize that legacy programs will often be part of an overall JMS application, and their inclusion must be anticipated in planning.
- **Messages.** The format and content of messages to be exchanged by JMS and non-JMS clients is integral to the design of a JMS application.
- **JMS provider.** As was stated previously, JMS defines a set of interfaces for which a provider must supply concrete implementations specific to its MOM product.
- **Administered objects.** An administrator of a messaging-system provider creates objects that are isolated from the proprietary technologies of the provider.

Administered objects

Providers of MOM products differ significantly in the mechanisms and techniques they use to implement messaging. To keep JMS clients portable, objects that implement the JMS interfaces must be isolated from a provider's proprietary technologies.
The mechanism for doing this is *administered objects*. These objects, which implement JMS interfaces, are created by an administrator of the provider's messaging system and are placed in the JNDI namespace.

The objects are then retrieved by JMS programs and accessed through the JMS interfaces that they implement. The JMS provider must supply a tool that allows creation of administered objects and their placement in the JNDI namespace.

There are two types of administered objects:

- **ConnectionFactory**: Used to create a connection to the provider's underlying messaging system.
- **Destination**: Used by the JMS client to specify the destination of messages being sent or the source of messages being received.

Although the administered objects themselves are instances of classes specific to a provider's implementation, they are retrieved using a portable mechanism (JNDI) and accessed through portable interfaces (JMS). The JMS program needs to know only the JNDI name and the JMS interface type of the administered object; no provider-specific knowledge is required.

### Interfaces

JMS defines a set of high-level interfaces that encapsulate various messaging concepts. In turn, these interfaces are further defined and customized for the two messaging domains -- PTP and pub/sub.

The high-level interfaces are:

- **ConnectionFactory**: An administered object that creates a **Connection**.
- **Connection**: An active connection to a provider.
- **Destination**: An administered object that encapsulates the identity of a message destination, such as where messages are sent to or received from.
- **Session**: A single-threaded context for sending and receiving messages. For reasons of simplicity and because **Session**'s control transactions, concurrent access by multiple threads is restricted. Multiple **Session**'s can be used for multithreaded applications.
- **MessageProducer**: Used for sending messages.
- **MessageConsumer**: Used for receiving messages.

### Interfaces (continued)

The following table identifies the domain-specific interfaces inherited from each high-level interface.

<table>
<thead>
<tr>
<th>High-level interface</th>
<th>PTP domain</th>
<th>Pub/sub domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
</tbody>
</table>

---

Introducing the Java Message Service
**Interfaces: What's changed in JMS 1.1**

In previous versions of JMS, the high-level interfaces were parents of the domain-specific interfaces and contained only those functions common to both domains. No implementations of the high-level interfaces were provided by the JMS provider. In JMS 1.1, the high-level interfaces are now considered "common interfaces" and contain all of the functionality of both domains; JMS providers must provide implementations of these interfaces. Although the common interfaces are still parents to the domain-specific interfaces, they are now the preferred method for JMS client programming, and the domain-specific interfaces are provided only for backward compatibility.

The following restates the table on the previous section, showing the common interfaces.

<table>
<thead>
<tr>
<th>JMS common interface</th>
<th>PTP domain</th>
<th>Pub/sub domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
<tr>
<td>Destination</td>
<td>Queue</td>
<td>Topic</td>
</tr>
<tr>
<td>Session</td>
<td>QueueSession</td>
<td>TopicSession</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>QueueSender</td>
<td>TopicPublisher</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>QueueReceiver, QueueBrowser</td>
<td>TopicSubscriber</td>
</tr>
</tbody>
</table>

Unification of domains with the common interfaces results in some domain-specific classes inheriting methods that are not suited for that domain. The JMS provider is required to throw an `IllegalStateException` should this occur in client code.

**Developing a JMS program**

A typical JMS program goes through the following steps to begin producing and consuming messages:

1. Look up a `ConnectionFactory` through JNDI.
2. Look up one or more `Destination`s through JNDI.
3. Use the `ConnectionFactory` to create a `Connection`.
4. Use the `Connection` to create one or more `Session`s.
5. Use a `Session` and a `Destination` to create the required `MessageProducer`s and `MessageConsumer`s.
6. Start the `Connection`.

At this point, messages can begin to flow, and the application can receive, process, and send messages, as required. In later sections, we'll develop JMS programs and you'll get to see this setup in detail.
Messages

At the heart of a messaging system are, of course, messages. JMS provides several message types for different types of content, but all messages derive from the Message interface.

A `Message` is divided into three constituent parts:

- The **header** is a standard set of fields that are used by both clients and providers to identify and route messages.
- **Properties** provide a facility for adding optional header fields to a message. If your application needs to categorize or classify a message in a way not provided by the standard header fields, you can add a property to the message to accomplish that categorization or classification. `set<Type>Property(...)` and `get<Type>Property(...)` methods are provided to set and get properties of a variety of Java types, including `Object`. JMS defines a standard set of properties that are optional for providers to supply.
- The **body** of the message contains the content to be delivered to a receiving application. Each message interface is specialized for the type of content it supports.

**Header fields**

The following list gives the name of each header field of `Message`, its corresponding Java type, and a description of the field:

- **JMSMessageID** -- type `String`  
  Uniquely identifies each message that is sent by a provider. This field is set by the provider during the send process; clients cannot determine the `JMSMessageID` for a message until after it has been sent.
- **JMSDestination** -- type `Destination`  
  The Destination to which the message is sent; set by the provider during the send process.
- **JMSDeliveryMode** -- type `int`  
  Contains the value `DeliveryMode.PERSISTENT` or `DeliveryMode.NON_PERSISTENT`. A persistent message is delivered "once and only once"; a non-persistent message is delivered "at most once." Be aware that "at most once" includes not being delivered at all. A non-persistent message might be lost by a provider during application or system failure. Extra care will be taken to assure that a persistent message is not affected by failures. There is often considerable overhead in sending persistent messages, and the trade-offs between reliability and performance must be carefully considered when deciding the delivery mode of a message.
- **JMSTimestamp** -- type `long`  
  The time that the message was delivered to a provider to be sent; set by the provider during the send process.
- **JMSExpiration** -- type `long`  
  The time when a message should expire. This value is calculated during the send process as the sum of the time-to-live value of the sending method and the current time. Expired messages should not be delivered by the provider. A value of 0 indicates that the message will not expire.
- **JMSPriority** -- type `int`
The priority of the message; set by the provider during the send process. A priority of 0 is the lowest priority; a priority of 9 is the highest priority.

- **JMSCorrelationID** -- type `string`  
  Typically used to link a response message with a request message; set by the JMS program sending the message. A JMS program responding to a message from another JMS program would copy the `JMSMessageID` of the message it is responding to into this field, so that the requesting program could correlate the response to the particular request that it made.

- **JMSReplyTo** -- type `Destination`  
  Used by a requesting program to indicate where a reply message should be sent; set by the JMS program sending the message.

- **JMSType** -- type `string`  
  Can be used by a JMS program to indicate the type of the message. Some providers maintain a repository of message types and will use this field to reference the type definition in the repository; in this case, the JMS program should not use this field.

- **JMSRedelivered** -- type `boolean`  
  Indicates that the message was delivered earlier to the JMS program, but that the program did not acknowledge its receipt; set by the provider during receive processing.

### Standard properties

The following list gives the name of each standard property of `Message`, its corresponding Java type, and a description of the property. Support for standard properties by a provider is optional. JMS reserves the "JMSX" property name for these and future JMS-defined properties.

- **JMSXUserID** -- type `string`  
  Identity of the user sending the message.

- **JMSXApplID** -- type `string`  
  Identity of the application sending the message.

- **JMSXDeliveryCount** -- type `int`  
  Number of times delivery of the message has been attempted.

- **JMSXGroupID** -- type `string`  
  Identity of the message group to which this message belongs.

- **JMSXGroupSeq** -- type `int`  
  Sequence number of this message within the message group.

- **JMSXProducerTXID** -- type `string`  
  Identity of the transaction within which this message was produced.

- **JMSXConsumerTXID** -- type `string`  
  Identity of the transaction within which this message was consumed.

- **JMSXRcvTimestamp** -- type `long`  
  The time JMS delivered the message to the consumer.

- **JMSXState** -- type `int`  
  Used by providers that maintain a message warehouse of messages; generally not of interest to JMS producers or consumers.

- **JMSX_<vendor_name>**  
  Reserved for provider-specific properties.
Message body

There are five forms of message body, and each form is defined by an interface that extends Message. These interfaces are:

- **StreamMessage**: Contains a stream of Java primitive values that are filled and read sequentially using standard stream operations.
- **MapMessage**: Contains a set of name-value pairs; the names are of type `String` and the values are Java primitives.
- **TextMessage**: Contains a `String`.
- **ObjectMessage**: Contains a `Serializable` Java object; JDK 1.2 collection classes can be used.
- **BytesMessage**: Contains a stream of uninterpreted bytes; allows encoding a body to match an existing message format.

Each provider supplies classes specific to its product that implement these interfaces. It is important to note that the JMS specification mandates that providers must be prepared to accept and handle a `Message` object that is not an instance of one of its own `Message` classes.

Although these "alien" objects might not be handled by a provider as efficiently as one of the provider's own implementations, they must be handled to ensure interoperability of all JMS providers.

Transactions

A JMS transaction groups a set of produced messages and a set of consumed messages into an atomic unit of work. If an error occurs during a transaction, the production and consumption of messages that occurred before the error can be "undone."

Session objects control transactions, and a `Session` can be denoted as **transacted** when it is created. A transacted `Session` always has a current transaction, that is, there is no `begin()`; `commit()` and `rollback()` end one transaction and automatically begin another.

Distributed transactions can be supported by the Java Transaction API (JTA) `XAResource` API, though this is optional for providers.

Acknowledgment

Acknowledgment is the mechanism whereby a provider is informed that a message has been successfully received.

If the `Session` receiving the message is transacted, acknowledgment is handled automatically. If the `Session` is not transacted, then the type of acknowledgment is determined when the `Session` is created.

There are three types of acknowledgment:
• **Session.DUPS_OK_ACKNOWLEDGE**: Lazy acknowledgment of message delivery; reduces overhead by minimizing work done to prevent duplicates; should be used only if duplicate messages are expected and can be handled.
• **Session.AUTO_ACKNOWLEDGE**: Message delivery is automatically acknowledged upon completion of the method that receives the message.
• **Session.CLIENT_ACKNOWLEDGE**: Message delivery is explicitly acknowledged by calling the `acknowledge()` method on the `Message`.

**Message selection**

JMS provides a mechanism, called a *message selector*, for a JMS program to filter and categorize the messages it receives.

The message selector is a `String` that contains an expression whose syntax is based on a subset of SQL92. The message selector is evaluated when an attempt is made to receive a message, and only messages that match the selection criteria of the selector are made available to the program.

Selection is based on matches to header fields and properties; body values cannot be used for selection. The syntax for message selectors is provided in detail in the JMS specification.

**JMS and XML**

The authors of JMS included the `TextMessage` message type on the presumption that `String` messages will be used extensively.

Their reasoning is that XML will be a popular, if not the most popular, means of representing the content of messages. A portable transport mechanism (JMS) coupled with a portable data representation (XML) is proving to be a powerful tool in enterprise application integration (EAI) and other areas of data exchange.

**JMS and J2EE**

J2EE version 1.2 requires compliant application servers to have the JMS API present but did not mandate the presence of a JMS provider.

J2EE version 1.3 requires application servers to supply a JMS provider. J2EE version 1.3 also introduced the *message-driven bean*, adding asynchronous notification abilities to Enterprise JavaBeans containers, as part of the EJB 2.0 specification. A message-driven bean implements the `MessageListener` interface (see `MessageListener` later in this tutorial), and is invoked by the EJB container on the arrival of a message at a destination designated at deployment time. Message-driven beans contain the business logic to process the message, including, if needed, invoking other enterprise beans.

J2EE version 1.4 requires J2EE products to include a JMS version 1.1 provider that supports both point-to-point and publish/subscribe messaging. It specifies that J2EE applications will not use the JMS client APIs for transaction handling; transactions will be handled by the J2EE containers. J2EE version 1.4 also requires that components in Web and EJB containers not create more than one active `Session` per connection.
New in JMS 1.1: Common interfaces

JMS 1.1 introduction

In this section, we'll look at each of the important JMS common interfaces for JMS client programming and some of their methods. It is important to remember that although the common interfaces are domain-independent, the behavior of client code matches the domain-specific behavior for the domain in operation at runtime.

In the next section, we'll look at some sample code that performs JMS message processing.

ConnectionFactory

ConnectionFactory is an administered object that is retrieved from JNDI to create a connection to a provider. It contains a createConnection() method, which returns a Connection object.

Connection

Connection encapsulates an active connection to a provider. Some of its methods are:

- createSession(boolean, int): Returns a Session object. The boolean parameter indicates whether the Session is transacted or not; the int indicates the acknowledgment mode (see Acknowledgment).
- start(): Activates the delivery of messages from the provider.
- stop(): Temporarily stops delivery of messages; delivery can be restarted with start().
- close(): Closes the connection to the provider and releases all resources held in its behalf.

Session

Session is the single-threaded context for sending and receiving messages. Some of its methods are:

- createProducer(Destination): Returns a MessageProducer object to send messages to the specified Destination.
- createConsumer(Destination): Returns a MessageConsumer object to receive messages from the specified Destination.
- commit(): Commits all consumed or produced messages for the current transaction.
- rollback(): Rolls back all consumed or produced messages for the current transaction.
- create<MessageType>Message(...): A variety of methods that return a <MessageType>Message -- for example, MapMessage, TextMessage, and so on.

Destination

Destination encapsulates a destination for messages. It is an administered object that is retrieved from JNDI.

MessageProducer

MessageProducer is used to send messages. Some of its methods are:
• send(Message): Sends the indicated Message.
• setDeliveryMode(int): Sets the delivery mode for subsequent messages sent; valid values are DeliveryMode.PERSISTENT and DeliveryMode.NON_PERSISTENT.
• setPriority(int): Sets the priority for subsequent messages sent; valid values are 0 through 9.
• setTimeToLive(long): Sets the duration before expiration, in milliseconds, of subsequent messages sent.

**MessageConsumer**

MessageConsumer is used to receive messages. Some of its methods are:

• receive(): Returns the next message that arrives; this method blocks until a message is available.
• receive(long): Receives the next message that arrives within long milliseconds; this method returns null if no message arrives within the time limit.
• receiveNoWait: Receives the next message if one is immediately available; this method returns null if no message is available.
• setMessageListener(MessageListener): Sets the MessageListener; the MessageListener object receives messages as they arrive, that is, asynchronously (see MessageListener).

**MessageListener**

MessageListener is an interface with a single method -- onMessage(Message) -- that provides asynchronous receipt and processing of messages.

This interface should be implemented by a client class and an instance of that class passed to the MessageConsumer object with the setMessageListener(MessageListener) method. As a message arrives at a destination, it is passed to the object with the onMessage(Message) method.

**Client programming with common interfaces**

**Client programming with common interfaces introduction**

In this section, we'll walk through two programs that do JMS client messaging -- Sender.java and Receiver.java.

We'll look at the code in small sections and describe what each section does. You can see the complete listings in the Appendix: Code listing for Sender.java and Code listing for Receiver.java.

**Sender: Prompt for JNDI names**

All of the sample programs are command-line programs that use System.in for input and System.out for output.

The Sender class has two methods: main(String[]) and send(). The main(String[]) method merely instantiates a Sender and calls its send() method.
The first section of the `send()` method prompts for the JNDI names of the administered objects that will be used to send messages.

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

public class Sender {
    public static void main(String[] args) {
        new Sender().send();
    }

    public void send() {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            // Prompt for JNDI names
            System.out.println("Enter ConnectionFactory name: ");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name: ");
            String destinationName = reader.readLine();
            ...
        }
    }
}
```

**Sender: Look up administered objects**

The next section of the `send()` method looks up the administered objects in JNDI, using the names input earlier.

JNDI is accessed by instantiating an `InitialContext` object; the administered objects are retrieved by calling the `lookup(String)` method, passing in the name of the object to be retrieved. Note that the `lookup(String)` method returns `Object`, so a typecast must be performed on the returned object.

```java
// Look up administered objects
InitialContext initContext = new InitialContext();
ConnectionFactory factory = (ConnectionFactory) initContext.lookup(factoryName);
Destination destination = (Destination) initContext.lookup(destinationName);
initContext.close();
```

**Sender: Create JMS objects**

Now, we create the JMS objects we need to send messages. Note that we don’t directly instantiate these objects using `new`. All of the objects are created by calling a method on another object.

First, we use the `ConnectionFactory` to create a `Connection`. We then use that `Connection` to create a `Session`.

The `Session` is not transacted (`false`) and will use automatic acknowledgment (`Session.AUTO_ACKNOWLEDGE`).

Finally, we create the `Sender` to send messages to the `Destination` we retrieved from JNDI.
Sender: Send messages

Now we’re ready to send messages. In this section, we enter a loop where we prompt for the text of a message to send. If the user types `quit`, the loop exits.

Otherwise, we build a `TextMessage` from the entered text and use the `MessageProducer` to send the message, then return to the top of the loop.

```java
//Send messages
String messageText = null;
while (true) {
    System.out.println("Enter message to send or 'quit':");
    messageText = reader.readLine();
    if ("quit".equals(messageText))
        break;
    TextMessage message = session.createTextMessage(messageText);
    sender.send(message);
}
```

Sender: Exit

Once the loop exits, we close the `Connection`. Closing the `Connection` automatically closes the `Session` and `MessageProducer`.

```java
//Exit
System.out.println("Exiting...");
reader.close();
connection.close();
System.out.println("Goodbye!");

} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

Receiver: Prompt for JNDI names and look up administered objects

The `Receiver` class, like the `Sender` class, has a `main(String[])` method that simply instantiates a `Receiver` and calls its primary method, `receive()`.

The code for prompting for JNDI names and doing the lookup of administered objects is identical to that in `Sender`.

There are two differences in this class, however:
• The `boolean` `stop` instance variable is used to indicate that the program should exit.
• `Receiver` implements the `MessageListener` interface in order to receive messages asynchronously.

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

public class Receiver implements MessageListener {
    private boolean stop = false;

    public static void main(String[] args) {
        new Receiver().receive();
    }

    public void receive() {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            // Prompt for JNDI names
            System.out.println("Enter ConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Destination name:");
            String destinationName = reader.readLine();
            reader.close();

            // Look up administered objects
            InitialContext initContext = new InitialContext();
            ConnectionFactory factory = (ConnectionFactory) initContext.lookup(factoryName);
            Destination destination = (Destination) initContext.lookup(destinationName);
            initContext.close();

            // Create JMS objects
            Connection connection = factory.createConnection();
            Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
            MessageConsumer receiver = session.createConsumer(queue);
            receiver.setMessageListener(this);
            connection.start();
        }
    }
}
```

**Receiver: Create JMS objects**

The `Connection` and `Session` are created as they are in `Sender`, and then a `MessageConsumer` is created.

Next, `setMessageListener()` is called, passing in `this` -- the local instance of `Receiver`, which you will recall implements the `MessageListener` interface.

Finally, the `Connection` is started to allow messages to be received.
**Receiver: Wait for stop and exit**

Next, the program goes into a loop that will exit when the `stop` variable becomes true. In the loop, the thread sleeps for one second. Once the loop has exited, the `Connection` is closed and the program terminates.

```java
...  // Wait for stop
  while (!stop) {
      Thread.sleep(1000);
  }

  // Exit
  System.out.println("Exiting...");
  connection.close();
  System.out.println("Goodbye!");
  } catch (Exception e) {
      e.printStackTrace();
      System.exit(1);
  }
}
```

**Receiver: onMessage(Message) method**

The presence of the `Receiver` class's `onMessage(Message)` method is required because `Receiver` implements the `MessageListener` interface.

When a message is received, this method is called with the `Message` passed as the parameter.

In our implementation, we get the text content of the message and print it to `System.out`. We then check to see if the message equals `stop`, and if it does, the `stop` variable is set to `true`; this allows the loop in the `receive()` method to terminate.

```java
...  public void onMessage(Message message) {
      try {
          String msgText = ((TextMessage) message).getText();
          System.out.println(msgText);
          if ("stop".equals(msgText))
              stop = true;
      } catch (JMSException e) {
          e.printStackTrace();
          stop = true;
      }
  }
```

**Running the programs**

As indicated in the **Getting started**, you need the `javax.naming` and `javax.jms` packages to compile the `Sender` and `Receiver` programs.

Before you run the programs, you need to use the administration tool supplied by your JMS provider to create the `ConnectionFactory` and `Destination` administered objects and place them in the JNDI namespace.
You also need to make sure that the provider’s JMS implementation classes are in your classpath.

You can then run both of these programs at the same time, supplying the same JNDI names for the `ConnectionFactory` and `Destination`, and send messages from the `Sender` to the `Receiver`.

**Point-to-point interfaces**

**Point-to-point interfaces introduction**

In this section, we'll look at each of the important JMS interfaces for point-to-point programming and some of their methods.

In the next section (Point-to-point programming), we'll look at some sample code that performs point-to-point message processing.

Remember: The point-to-point and pub/sub interfaces are included in JMS version 1.1 only for backward compatibility. If you are doing new development, use the common interfaces covered in the previous two sections (New in JMS 1.1: Common interfaces and Client programming with common interfaces).

**QueueConnectionFactory**

`QueueConnectionFactory` is an administered object that is retrieved from JNDI to create a connection to a provider. It contains a `createQueueConnection()` method, which returns a `QueueConnection` object.

**QueueConnection**

`QueueConnection` encapsulates an active connection to a provider. Some of its methods are:

- `createQueueSession(boolean, int)`: Returns a `QueueSession` object. The `boolean` parameter indicates whether the `QueueSession` is transacted or not; the `int` indicates the acknowledgment mode (see `Acknowledgment`).
- `start()` (inherited from `Connection`): Activates the delivery of messages from the provider.
- `stop()` (inherited from `Connection`): Temporarily stops delivery of messages; delivery can be restarted with `start()`.
- `close()` (inherited from `Connection`): Closes the connection to the provider and releases all resources held in its behalf.

**QueueSession**

`QueueSession` is the single-threaded context for sending and receiving PTP messages. Some of its methods are:

- `createSender(Queue)`: Returns a `QueueSender` object to send messages to the specified `Queue`.
- `createReceiver(Queue)`: Returns a `QueueReceiver` object to receive messages from the specified `Queue`.
• createBrowser(Queue) (inherited from Session): Returns a QueueBrowser object to browse messages on the specified Queue.
• commit() (inherited from Session): Commits all consumed or produced messages for the current transaction.
• rollback() (inherited from Session): Rolls back all consumed or produced messages for the current transaction.
• create<MessageType>Message(...) (inherited from Session): A variety of methods that return a <MessageType>Message -- for example, MapMessage, TextMessage, and so on.

Queue
Queue encapsulates a point-to-point destination. It is an administered object that is retrieved from JNDI.

QueueSender
QueueSender is used to send point-to-point messages. Some of its methods are:

• send(Message): Sends the indicated Message.
• setDeliveryMode(int) (inherited from MessageProducer): Sets the delivery mode for subsequent messages sent; valid values are DeliveryMode.PERSISTENT and DeliveryMode.NON_PERSISTENT.
• setPriority(int) (inherited from MessageProducer): Sets the priority for subsequent messages sent; valid values are 0 through 9.
• setTimeToLive(long) (inherited from MessageProducer): Sets the duration before expiration, in milliseconds, of subsequent messages sent.

QueueReceiver
QueueReceiver is used to receive point-to-point messages. Some of its methods are:

• receive() (inherited from MessageConsumer): Returns the next message that arrives; this method blocks until a message is available.
• receive(long) (inherited from MessageConsumer): Receives the next message that arrives within long milliseconds; this method returns null if no message arrives within the time limit.
• receiveNoWait (inherited from MessageConsumer): Receives the next message if one is immediately available; this method returns null if no message is available.
• setMessageListener(MessageListener) (inherited from MessageConsumer): Sets the MessageListener; the MessageListener object receives messages as they arrive, that is, asynchronously (see MessageListener).

QueueBrowser
When QueueReceiver is used to receive messages, the messages are removed from the queue when they are received. QueueBrowser is used to look at messages on a queue without removing them. The method for doing that is getEnumeration(), which returns a java.util Enumeration that can be used to scan the messages in the queue; changes to the queue (arriving and expiring of messages) may or may not be visible.
Point-to-point programming

Point-to-point programming introduction

In this section, we'll walk through two programs that do point-to-point messaging -- QSender.java and Q Receiver.java.

We'll look at the code in small sections and describe what each section does. You can see the complete listings in the Appendix: Code listing for QSender.java and Code listing for QReceiver.java.

QSender: Prompt for JNDI names

All of the sample programs are command-line programs that use System.in for input and System.out for output.

The QSender class has two methods: main(String[]) and send(). The main(String[]) method merely instantiates a QSender and calls its send() method.

The first section of the send() method prompts for the JNDI names of the administered objects that will be used to send messages.

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

public class QSender {
    public static void main(String[] args) {
        new QSender().send();
    }
    public void send() {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            // Prompt for JNDI names
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();
        }...
    }
}
```

QSender: Look up administered objects

The next section of the send() method looks up the administered objects in JNDI, using the names input earlier.

JNDI is accessed by instantiating an InitialContext object; the administered objects are retrieved by calling the lookup(String) method, passing in the name of the object to be retrieved. Note that the lookup(String) method returns Object, so a typecast must be performed on the returned object.
QSender: Create JMS objects

Now, we create the JMS objects we need to send messages. Note that we don't directly instantiate these objects using `new`. All of the objects are created by calling a method on another object.

First, we use the `QueueConnectionFactory` to create a `QueueConnection`. We then use that `QueueConnection` to create a `QueueSession`. The `QueueSession` is not transacted (`false`) and will use automatic acknowledgment (`Session.AUTO_ACKNOWLEDGE`).

Finally, we create the `QueueSender` to send messages to the `Queue` we retrieved from JNDI.

QSender: Send messages

Now we're ready to send messages. In this section, we enter a loop where we prompt for the text of a message to send. If the user types `quit`, the loop exits.

Otherwise, we build a `TextMessage` from the entered text and use the `QueueSender` to send the message, then return to the top of the loop.

QSender: Exit

Once the loop exits, we close the `QueueConnection`. Closing the `QueueConnection` automatically closes the `QueueSession` and `QueueSender`.

```java
//Look up administered objects
InitialContext initContext = new InitialContext();
QueueConnectionFactory factory =
  (QueueConnectionFactory) initContext.lookup(factoryName);
Queue queue = (Queue) initContext.lookup(queueName);
initContext.close();

//Create JMS objects
QueueConnection connection = factory.createQueueConnection();
QueueSession session =
  connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueSender sender = session.createSender(queue);

//Send messages
String messageText = null;
while (true) {
  System.out.println("Enter message to send or 'quit':");
  messageText = reader.readLine();
  if ("quit".equals(messageText))
    break;
  TextMessage message = session.createTextMessage(messageText);
  sender.send(message);
}

//Exit
QueueConnection connection = null;
QueueSession session = null;
QueueSender sender = null;
```
QReceiver: Prompt for JNDI names and look up administered objects

The QReceiver class, like the QSender class, has a main(String[]) method that simply instantiates a QReceiver and calls its primary method, receive().

The code for prompting for JNDI names and doing the lookup of administered objects is identical to that in QSender.

There are two differences in this class, however:

- The boolean stop instance variable is used to indicate that the program should exit.
- QReceiver implements the MessageListener interface in order to receive messages asynchronously.

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

public class QReceiver implements MessageListener {
    private boolean stop = false;

    public static void main(String[] args) {
        new QReceiver().receive();
    }

    public void receive() {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        try {
            // Prompt for JNDI names
            System.out.println("Enter QueueConnectionFactory name:");
            String factoryName = reader.readLine();
            System.out.println("Enter Queue name:");
            String queueName = reader.readLine();
            reader.close();

            // Look up administered objects
            InitialContext initContext = new InitialContext();
            QueueConnectionFactory factory = (QueueConnectionFactory) initContext.lookup(factoryName);
            Queue queue = (Queue) initContext.lookup(queueName);
            initContext.close();

            // Code to process messages...
        }
    }
}
```
QReceiver: Create JMS objects

The QueueConnection and QueueSession are created as they are in QSender, and then a QueueReceiver is created.

Next, setMessageListener() is called, passing in this -- the local instance of QReceiver, which you will recall implements the MessageListener interface.

Finally, the QueueConnection is started to allow messages to be received.

```java
//Create JMS objects
QueueConnection connection = factory.createQueueConnection();
QueueSession session =
    connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueReceiver receiver = session.createReceiver(queue);
receiver.setMessageListener(this);
connection.start();
```

QReceiver: Wait for stop and exit

Next, the program goes into a loop that will exit when the stop variable becomes true. In the loop, the thread sleeps for one second. Once the loop has exited, the QueueConnection is closed and the program terminates.

```java
//Wait for stop
while (!stop) {
    Thread.sleep(1000);
}
//Exit
System.out.println("Exiting...");
connection.close();
System.out.println("Goodbye!");

} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

QReceiver: onMessage(Message) method

The presence of the QReceiver class's onMessage(Message) method is required because QReceiver implements the MessageListener interface.

When a message is received, this method is called with the Message passed as the parameter.

In our implementation, we get the text content of the message and print it to System.out. We then check to see if the message equals stop, and if it does, the stop variable is set to true; this allows the loop in the receive() method to terminate.
```java
public void onMessage(Message message) {
    try {
        String msgText = ((TextMessage) message).getText();
        System.out.println(msgText);
        if ("stop".equals(msgText))
            stop = true;
    } catch (JMSException e) {
        e.printStackTrace();
        stop = true;
    }
}
}
```

**Pub/sub interfaces**

**Pub/sub interfaces introduction**

Now let's look at the pub/sub interfaces. As we go through them, notice how they are very much like the PTP interfaces, except for the names and a few other differences.

Remember: The point-to-point and pub/sub interfaces are included in JMS version 1.1 only for backward compatibility. If you are doing new development, use the common interfaces covered in the New in JMS 1.1: Common interfaces and Client programming with common interfaces sections.

**TopicConnectionFactory**

`TopicConnectionFactory` is an administered object that is retrieved from JNDI in order to create a connection to a provider. It contains a `createTopicConnection()` method, which returns a `TopicConnection` object.

**TopicConnection**

`TopicConnection` encapsulates an active connection to a provider. Some of its methods are:

- `createTopicSession(boolean, int)`: Returns a `TopicSession` object. The `boolean` parameter indicates whether the `TopicSession` is transacted; the `int` indicates the acknowledgment mode (see `Acknowledgment`).
- `start()` (inherited from `Connection`): Activates the delivery of messages from the provider.
- `stop()` (inherited from `Connection`): Temporarily stops delivery of messages; delivery can be restarted with `start()`.
- `close()` (inherited from `Connection`): Closes the connection to the provider and releases all resources held in its behalf.

**TopicSession**

`TopicSession` is the single-threaded context for sending and receiving pub/sub messages. Some of its methods are:

- `createPublisher(Topic)`: Returns a `TopicPublisher` object to send messages to the specified `Topic`. 
• createSubscriber(Topic): Returns a TopicSubscriber object to receive messages from the specified Topic. This subscriber is non-durable; that is, the subscription will last only for the lifetime of the object and will receive messages only when it is active.
• createDurableSubscriber(Topic, String): Returns a TopicSubscriber object to receive messages from the specified Topic, giving the String name to the subscriber. Messages for a durable subscriber will be retained by JMS if the object is not active and will be delivered to subsequent subscriber objects that are created with the same name.
• unsubscribe(String): Ends the subscription with the String name.
• commit() (inherited from Session): Commits all consumed or produced messages for the current transaction.
• rollback() (inherited from Session): Rolls back all consumed or produced messages for the current transaction.
• create<MessageType>Message(...) (inherited from Session): A variety of methods that return a <MessageType>Message, such as MapMessage, TextMessage, and so on.

Topic

Topic encapsulates a pub/sub destination. It is an administered object that is retrieved from JNDI.

TopicPublisher

TopicPublisher is used to send pub/sub messages. Some of its methods are:

• publish(Message): Publishes the indicated Message.
• setDeliveryMode(int) (inherited from MessageProducer): Sets the delivery mode for subsequent messages sent; valid values are DeliveryMode.PERSISTENT and DeliveryMode.NON_PERSISTENT.
• setPriority(int) (inherited from MessageProducer): Sets the priority for subsequent messages sent; valid values are 0 through 9.
• setTimeToLive(long) (inherited from MessageProducer): Sets the duration before expiration, in milliseconds, of subsequent messages sent.

TopicSubscriber

TopicSubscriber is used to receive point-to-point messages. Some of its methods are:

• receive() (inherited from MessageConsumer): Returns the next message that arrives; this method blocks until a message is available.
• receive(long) (inherited from MessageConsumer): Receives the next message that arrives within long milliseconds; this method returns null if no message arrives within the time limit.
• receiveNowait (inherited from MessageConsumer): Receives the next message if one is immediately available; this method returns null if no message is available.
• setMessageListener(MessageListener) (inherited from MessageConsumer): Sets the MessageListener; the MessageListener object receives messages as they arrive, that is, asynchronously (see MessageListener).
Pub/sub programming

The same, but different

Two pub/sub programs are available in the Appendix -- Code listing for TPublisher.java and Code listing for TSubscriber.java. We won't go through them step-by-step as we did the PTP programs because, other than the types of JMS interfaces used, they are identical to QSender.java and QReceiver.java.

You need to set up TopicConnectionFactory and Topic administered objects before you run these programs.

You'll see the difference between these and the PTP programs once you run them. If you run multiple instances of QReceiver using the same QueueConnectionFactory and Queue, you'll see that as you send messages from QSender, only one of the QReceiver instances receives each message sent.

If you run multiple instances of TSubscriber, you'll see that all messages sent from TPublisher are received by all instances of TSubscriber.

Wrap-up

Summary

This tutorial has provided an introduction and overview of the Java Message Service and its functionality and capabilities. It has also demonstrated basic programming techniques for creating JMS programs and provided sample code to illustrate those programs.

We did not look at every interface and class in the JMS API, nor did we look at every method on those interfaces we did examine. The Related topics provide some pointers to materials to help you do that.

The goal here is to get you started with JMS and give you some basic working programs to learn from. Once you have the sample programs up and running, experiment by modifying them to use message selection, durable subscriptions, and some of the other capabilities of JMS that we touched on here but did not demonstrate in the sample programs.

Appendix

Code listing for Sender.java

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

public class Sender {
    public static void main(String[] args) {
```
new Sender().send();
}

public void send() {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        // Prompt for JNDI names
        System.out.println("Enter ConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Destination name:");
        String destinationName = reader.readLine();

        // Look up administered objects
        InitialContext initContext = new InitialContext();
        ConnectionFactory factory =
            (ConnectionFactory) initContext.lookup(factoryName);
        Destination destination = (Destination) initContext.lookup(destinationName);
        initContext.close();

        // Create JMS objects
        Connection connection = factory.createConnection();
        Session session =
            connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageProducer sender = session.createProducer(queue);

        // Send messages
        String messageText = null;
        while (true) {
            System.out.println("Enter message to send or 'quit':");
            messageText = reader.readLine();
            if ("quit".equals(messageText))
                break;
            TextMessage message = session.createTextMessage(messageText);
            sender.send(message);
        }

        // Exit
        System.out.println("Exiting...");
        reader.close();
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
}

Code listing for Receiver.java

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

public class Receiver implements MessageListener {
    private boolean stop = false;
    public static void main(String[] args) {
        new Receiver().receive();
    }
}
```
public void receive() { 

    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

    try {
        //Prompt for JNDI names
        System.out.println("Enter ConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Destination name:");
        String destinationName = reader.readLine();
        reader.close();

        //Look up administered objects
        InitialContext initContext = new InitialContext();
        ConnectionFactory factory =
            (ConnectionFactory) initContext.lookup(factoryName);
        Destination destination = (Destination) initContext.lookup(destinationName);
        initContext.close();

        //Create JMS objects
        Connection connection = factory.createConnection();
        Session session =
            connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        MessageConsumer receiver = session.createConsumer(queue);
        receiver.setMessageListener(this);
        connection.start();

        //Wait for stop
        while (!stop) {
            Thread.sleep(1000);
        }

        //Exit
        System.out.println("Exiting...");
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}

public void onMessage(Message message) {

    try {
        String msgText = ((TextMessage) message).getText();
        System.out.println(msgText);
        if ("stop".equals(msgText))
            stop = true;
    } catch (JMSException e) {
        e.printStackTrace();
        stop = true;
    }
}

Code listing for QSender.java

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

public class QSender {

    public static void main(String[] args) {

Introducing the Java Message Service
new QSender().send();
}

public void send() {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        // Prompt for JNDI names
        System.out.println("Enter QueueConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Queue name:");
        String queueName = reader.readLine();
        // Look up administered objects
        InitialContext initContext = new InitialContext();
        QueueConnectionFactory factory =
        (QueueConnectionFactory) initContext.lookup(factoryName);
        Queue queue = (Queue) initContext.lookup(queueName);
        initContext.close();
        // Create JMS objects
        QueueConnection connection = factory.createQueueConnection();
        QueueSession session =
        connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueSender sender = session.createSender(queue);
        // Send messages
        String messageText = null;
        while (true) {
            System.out.println("Enter message to send or 'quit':");
            messageText = reader.readLine();
            if ("quit".equals(messageText))
                break;
            TextMessage message = session.createTextMessage(messageText);
            sender.send(message);
        }
        // Exit
        System.out.println("Exiting...");
        reader.close();
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}

Code listing for QReceiver.java

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

public class QReceiver implements MessageListener {
    private boolean stop = false;
    public static void main(String[] args) {
        new QReceiver().receive();
    }
}
public void receive() {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        // Prompt for JNDI names
        System.out.println("Enter QueueConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Queue name:");
        String queueName = reader.readLine();
        reader.close();

        // Look up administered objects
        InitialContext initContext = new InitialContext();
        QueueConnectionFactory factory = (QueueConnectionFactory) initContext.lookup(factoryName);
        Queue queue = (Queue) initContext.lookup(queueName);
        initContext.close();

        // Create JMS objects
        QueueConnection connection = factory.createQueueConnection();
        QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueReceiver receiver = session.createReceiver(queue);
        receiver.setMessageListener(this);
        connection.start();

        // Wait for stop
        while (!stop) {
            Thread.sleep(1000);
        }

        // Exit
        System.out.println("Exiting...");
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}

public void onMessage(Message message) {
    try {
        String msgText = ((TextMessage) message).getText();
        System.out.println(msgText);
        if ("stop".equals(msgText))
            stop = true;
    } catch (JMSException e) {
        e.printStackTrace();
        stop = true;
    }
}

Code listing for TPublisher.java

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

public class TPublisher {

Introducing the Java Message Service
public static void main(String[] args) {
    new TPublisher().publish();
}

public void publish() {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        //Prompt for JNDI names
        System.out.println("Enter TopicConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Topic name:");
        String topicName = reader.readLine();

        //Look up administered objects
        InitialContext initContext = new InitialContext();
        TopicConnectionFactory factory = (TopicConnectionFactory) initContext.lookup(factoryName);
        Topic topic = (Topic) initContext.lookup(topicName);
        initContext.close();

        //Create JMS objects
        TopicConnection connection = factory.createTopicConnection();
        TopicSession session = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        TopicPublisher publisher = session.createPublisher(topic);

        //Send messages
        String messageText = null;
        while (true) {
            System.out.println("Enter message to send or 'quit':");
            messageText = reader.readLine();
            if ("quit".equals(messageText))
                break;
            TextMessage message = session.createTextMessage(messageText);
            publisher.publish(message);
        }

        //Exit
        System.out.println("Exiting...");
        reader.close();
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}

Code listing for TSubscriber.java

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

public class TSubscriber implements MessageListener {
    private boolean stop = false;
    public static void main(String[] args) {
        new TSubscriber().subscribe();
    }
}
public void subscribe() {
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    try {
        // Prompt for JNDI names
        System.out.println("Enter TopicConnectionFactory name:");
        String factoryName = reader.readLine();
        System.out.println("Enter Topic name:");
        String topicName = reader.readLine();
        reader.close();

        // Look up administered objects
        InitialContext initContext = new InitialContext();
        TopicConnectionFactory factory =
            (TopicConnectionFactory) initContext.lookup(factoryName);
        Topic topic = (Topic) initContext.lookup(topicName);
        initContext.close();

        // Create JMS objects
        TopicConnection connection = factory.createTopicConnection();
        TopicSession session =
            connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        TopicSubscriber subscriber = session.createSubscriber(topic);
        subscriber.setMessageListener(this);
        connection.start();

        // Wait for stop
        while (!stop) {
            Thread.sleep(1000);
        }

        // Exit
        System.out.println("Exiting...");
        connection.close();
        System.out.println("Goodbye!");
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}  

public void onMessage(Message message) {
    try {
        String msgText = ((TextMessage) message).getText();
        System.out.println(msgText);
        if ("stop".equals(msgText))
            stop = true;
    } catch (JMSException e) {
        e.printStackTrace();
        stop = true;
    }  
}  

Related topics

- The Java Message Service specification, version 1.1 is the best source of information for understanding the finer details of JMS.
- The JMS API documentation is essential for JMS programming.
- You'll need to download the javax.jms package and javax.naming package to complete this tutorial.
- For enterprise development, you'll need the Java 2 Enterprise Edition. J2EE version 1.2 requires compliant application servers to have the JMS API present, but it does not mandate the presence of a JMS provider; J2EE version 1.3 requires application servers to supply a JMS provider. J2EE version 1.4 requires application servers to supply a JMS provider that supports both domains.
- Message-driven beans, part of the EJB 2.0 specification, add asynchronous notification abilities to Enterprise JavaBeans containers.
- For a distillation of the changes in JMS 1.1, see David Currie's "Get the message" (developerWorks, April 2004).
- Also see the Java technology zone tutorials page for a complete listing of free Java-focused tutorials from developerWorks.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)