**http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html**

**The JMS API Programming Model**

The basic building blocks of a JMS application consist of

* [Administered Objects](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp78884): connection factories and destinations

* [Connections](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp78986)

* [Sessions](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79001)

* [Message Producers](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79085)

* [Message Consumers](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79145)

* [Messages](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79302)

[Figure 33-5](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp78861) shows how all these objects fit together in a JMS client application.

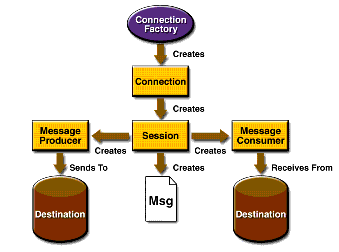


Figure 33-5 The JMS API Programming Model

This section describes all these objects briefly and provides sample commands and code snippets that show how to create and use the objects. The last subsection briefly describes JMS API exception handling.

Examples that show how to combine all these objects in applications appear in later sections. For more details, see the JMS API documentation, which is part of the J2EE API documentation.

Administered Objects

Two parts of a JMS application--destinations and connection factories--are best maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access these objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then look them up by using the JNDI API. J2EE applications always use the JNDI API.

With the Application Server, you use the Admin Console to create JMS administered objects in the form of resources. You can also use the asadmin command.

**Connection Factories**

A *connection factory* is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory interface.

To learn how to use the Admin Console to create connection factories, see [Creating JMS Administered Objects](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS5.html#wp80290).

At the beginning of a JMS client program, you usually perform a JNDI lookup of a connection factory, then cast and assign it to a ConnectionFactory object.

For example, the following code fragment obtains an InitialContext object and uses it to look up a ConnectionFactory by name. Then it assigns it to a ConnectionFactory object:

Context ctx = new InitialContext();

ConnectionFactory connectionFactory = (ConnectionFactory)

  ctx.lookup("jms/ConnectionFactory");

In a J2EE application, JMS administered objects are normally placed in the jms naming subcontext.

**Destinations**

A *destination* is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging domain, destinations are called queues. In the pub/sub messaging domain, destinations are called topics.

Creating destinations using the Application Server is a two-step process. You create a JMS destination resource that specifies the JNDI name of the destination. You also create a physical destination to which the JNDI name refers.

To learn how to use the Admin Console to create physical destinations and destination resources, see [Creating JMS Administered Objects](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS5.html#wp80290).

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

In addition to looking up a connection factory in a client program, you usually look up a destination. Unlike connection factories, destinations are specific to one domain or the other. To create an application that allows you to use the same code for both topics and queues, you cast and assign the destination to a Destination object. To preserve the semantics of queues and topics, however, you cast and assign the object to a destination of the appropriate type.

For example, the following line of code performs a JNDI lookup of the previously created topic jms/MyTopic and casts and assigns it to a Destination object:

Destination myDest = (Destination) ctx.lookup("jms/MyTopic");

The following line of code looks up a queue named jms/MyQueue and casts and assigns it to a Queue object:

Queue myQueue = (Queue) ctx.lookup("jms/MyQueue");

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the ConnectionFactory interface, you can look up a QueueConnectionFactory and use it with a Topic, and you can look up a TopicConnectionFactory and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

Connections

A *connection* encapsulates a virtual connection with a JMS provider. A connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Connections implement the Connection interface. When you have a ConnectionFactory object, you can use it to create a Connection:

Connection connection = connectionFactory.createConnection();

Before an application completes, you must close any connections that you have created. Failure to close a connection can cause resources not to be released by the JMS provider. Closing a connection also closes its sessions and their message producers and message consumers.

connection.close();

Before your application can consume messages, you must call the connection's start method; for details, see [Message Consumers](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79145). If you want to stop message delivery temporarily without closing the connection, you call thestop method.

Sessions

A *session* is a single-threaded context for producing and consuming messages. You use sessions to create message producers, message consumers, and messages. Sessions serialize the execution of message listeners; for details, see [Message Listeners](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79175).

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work. For details, see [Using JMS API Local Transactions](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS6.html#wp92878).

Sessions implement the Session interface. After you create a Connection object, you use it to create a Session:

Session session = connection.createSession(false,

  Session.AUTO\_ACKNOWLEDGE);

The first argument means that the session is not transacted; the second means that the session automatically acknowledges messages when they have been received successfully. (For more information, see [Controlling Message Acknowledgment](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS6.html#wp81785).)

To create a transacted session, use the following code:

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

Here, the first argument means that the session is transacted; the second indicates that message acknowledgment is not specified for transacted sessions. For more information on transactions, see [Using JMS API Local Transactions](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS6.html#wp92878). For information about the way JMS transactions work in J2EE applications, see [Using the JMS API in a J2EE Application](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS7.html#wp82114).

Message Producers

A *message producer* is an object that is created by a session and used for sending messages to a destination. It implements the MessageProducer interface.

You use a Session to create a MessageProducer for a destination. Here, the first example creates a producer for the destination myQueue, and the second for the destination myTopic:

MessageProducer producer = session.createProducer(myQueue);

MessageProducer producer = session.createProducer(myTopic);

You can create an unidentified producer by specifying null as the argument to createProducer. With an unidentified producer, you do not specify a destination until you send a message.

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

producer.send(message);

You must first create the messages; see [Messages](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79302).

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

MessageProducer anon\_prod = session.createProducer(null);

anon\_prod.send(myQueue, message);

Message Consumers

A *message consumer* is an object that is created by a session and used for receiving messages sent to a destination. It implements the MessageConsumer interface.

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

For example, you use a Session to create a MessageConsumer for either a queue or a topic:

MessageConsumer consumer = session.createConsumer(myQueue);

MessageConsumer consumer = session.createConsumer(myTopic);

You use the Session.createDurableSubscriber method to create a durable topic subscriber. This method is valid only if you are using a topic. For details, see [Creating Durable Subscriptions](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS6.html#wp81941).

After you have created a message consumer, it becomes active, and you can use it to receive messages. You can use the close method for a MessageConsumer to make the message consumer inactive. Message delivery does not begin until you start the connection you created by calling its start method. (Remember always to call the start method; forgetting to start the connection is one of the most common JMS programming errors.)

You use the receive method to consume a message synchronously. You can use this method at any time after you call the start method:

connection.start();

Message m = consumer.receive();

connection.start();

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

To consume a message asynchronously, you use a message listener, described in [Message Listeners](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79175).

**Message Listeners**

A *message listener* is an object that acts as an asynchronous event handler for messages. This object implements the MessageListener interface, which contains one method, onMessage. In the onMessage method, you define the actions to be taken when a message arrives.

You register the message listener with a specific MessageConsumer by using the setMessageListener method. For example, if you define a class named Listener that implements the MessageListener interface, you can register the message listener as follows:

Listener myListener = new Listener();

consumer.setMessageListener(myListener);

After you register the message listener, you call the start method on the Connection to begin message delivery. (If you call start before you register the message listener, you are likely to miss messages.)

When message delivery begins, the JMS provider automatically calls the message listener's onMessage method whenever a message is delivered. The onMessage method takes one argument of type Message, which your implementation of the method can cast to any of the other message types (see [Message Bodies](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79620)).

A message listener is not specific to a particular destination type. The same listener can obtain messages from either a queue or a topic, depending on the type of destination for which the message consumer was created. A message listener does, however, usually expect a specific message type and format. Moreover, if it needs to reply to messages, a message listener must either assume a particular destination type or obtain the destination type of the message and create a producer for that destination type.

Your onMessage method should handle all exceptions. It must not throw checked exceptions, and throwing a RuntimeException is considered a programming error.

The session used to create the message consumer serializes the execution of all message listeners registered with the session. At any time, only one of the session's message listeners is running.

In the J2EE platform, a message-driven bean is a special kind of message listener. For details, see [Using Message-Driven Beans](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS7.html#wp82163).

**Message Selectors**

If your messaging application needs to filter the messages it receives, you can use a JMS API message selector, which allows a message consumer to specify the messages it is interested in. Message selectors assign the work of filtering messages to the JMS provider rather than to the application. For an example of an application that uses a message selector, see [A J2EE Application That Uses the JMS API with a Session Bean](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMSJ2EEex2.html#wp95588).

A message selector is a String that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has aNewsType property that is set to the value 'Sports' or 'Opinion':

NewsType = 'Sports' OR NewsType = 'Opinion'

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

The message consumer then receives only messages whose headers and properties match the selector. (See [Message Headers](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79367), and [Message Properties](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79540).) A message selector cannot select messages on the basis of the content of the message body.

  前面的例子中创建一个消息消费者使用的是:  
          sesssion.createConsumer(destination)  
  另外，还提供了另一种方式：  
    sesssion.createConsumer(destination, selector)  
  这里selector是一个字符串，用来过滤消息。也就是说，这种方式可以创建一个可以只接收特定消息的一个消费者。Selector的格式是类似于SQL-92的一种语法。可以用来比较消息头信息和属性。  
  
  下面的例子中，创建两个消费者，共同监听同一个Queue，但是它们的Selector不同，然后创建一个消息生产者，来发送多个消息。  
  
import javax.jms.Connection;  
import javax.jms.JMSException;  
import javax.jms.Message;  
import javax.jms.MessageConsumer;  
import javax.jms.MessageListener;  
import javax.jms.MessageProducer;  
import javax.jms.Queue;  
import javax.jms.Session;  
import javax.jms.TextMessage;  
  
import org.apache.activemq.ActiveMQConnectionFactory;  
import org.apache.activemq.command.ActiveMQQueue;  
  
public class JMSSelectorTest {  
  
    public static void main(String[] args) throws Exception {  
        ActiveMQConnectionFactory factory = new ActiveMQConnectionFactory("vm://localhost");  
     
        Connection connection = factory.createConnection();  
        connection.start();  
         
        Queue queue = new ActiveMQQueue("testQueue");  
         
        Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);  
         
        MessageConsumer comsumerA = session.createConsumer(queue, "receiver = 'A'");  
        comsumerA.setMessageListener(new MessageListener(){  
            public void onMessage(Message m) {  
                try {  
                    System.out.println("ConsumerA get " + ((TextMessage) m).getText());  
                } catch (JMSException e1) { }  
            }  
        });  
         
        MessageConsumer comsumerB = session.createConsumer(queue, "receiver = 'B'");  
        comsumerB.setMessageListener(new MessageListener(){  
            public void onMessage(Message m) {  
                try {  
                    System.out.println("ConsumerB get " + ((TextMessage) m).getText());  
                } catch (JMSException e) { }  
            }  
        });  
         
        MessageProducer producer = session.createProducer(queue);  
        for(int i=0; i<10; i++) {  
            String receiver = (i%3 == 0 ? "A" : "B");  
            TextMessage message = session.createTextMessage("Message" + i + ", receiver:" + receiver);  
            message.setStringProperty("receiver", receiver);  
            producer.send(message );  
        }  
    }  
}  
  
结果如下：  
ConsumerA get Message0, receiver:A  
ConsumerB get Message1, receiver:B  
ConsumerB get Message2, receiver:B  
ConsumerA get Message3, receiver:A  
ConsumerB get Message4, receiver:B  
ConsumerB get Message5, receiver:B  
ConsumerA get Message6, receiver:A  
ConsumerB get Message7, receiver:B  
ConsumerB get Message8, receiver:B  
ConsumerA get Message9, receiver:A  
  
可以看出，消息消费者只会取走它自己感兴趣的消息。

### Messages

The ultimate purpose of a JMS application is to produce and to consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message has three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts:

* [Message Headers](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79367)

* [Message Properties](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79540) (optional)

* [Message Bodies](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html" \l "wp79620) (optional)

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

#### Message Headers

A JMS message header contains a number of predefined fields that contain values that both clients and providers use to identify and to route messages. [Table 33-1](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79545) lists the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field JMSMessageID. The value of another header field, JMSDestination, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the Message interface. Some header fields are intended to be set by a client, but many are set automatically by the send or the publish method, which overrides any client-set values.

|  |  |
| --- | --- |
| *Table 33-1 How JMS Message Header Field Values Are Set* | |
| **Header Field** | **Set By** |
| JMSDestination | send or publish method |
| JMSDeliveryMode | send or publish method |
| JMSExpiration | send or publish method |
| JMSPriority | send or publish method |
| JMSMessageID | send or publish method |
| JMSTimestamp | send or publish method |
| JMSCorrelationID | Client |
| JMSReplyTo | Client |
| JMSType | Client |
| JMSRedelivered | JMS provider |

#### Message Properties

You can create and set properties for messages if you need values in addition to those provided by the header fields. You can use properties to provide compatibility with other messaging systems, or you can use them to create message selectors (see [Message Selectors](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79281)). For an example of setting a property to be used as a message selector, see [A J2EE Application That Uses the JMS API with a Session Bean](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMSJ2EEex2.html#wp95588).

The JMS API provides some predefined property names that a provider can support. The use either of these predefined properties or of user-defined properties is optional.

#### Message Bodies

The JMS API defines five message body formats, also called message types, which allow you to send and to receive data in many different forms and provide compatibility with existing messaging formats. [Table 33-2](http://docs.oracle.com/javaee/1.4/tutorial/doc/JMS4.html#wp79628) describes these message types.

|  |  |
| --- | --- |
| *Table 33-2 JMS Message Types* | |
| **Message Type** | **Body Contains** |
| TextMessage | A java.lang.String object (for example, the contents of an Extensible Markup Language file). |
| MapMessage | A set of name-value pairs, with names as String objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined. |
| BytesMessage | A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format. |
| StreamMessage | A stream of primitive values in the Java programming language, filled and read sequentially. |
| ObjectMessage | A Serializable object in the Java programming language. |
| Message | Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required. |

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a TextMessage, you might use the following statements:

TextMessage message = session.createTextMessage();

message.setText(msg\_text); // msg\_text is a String

producer.send(message);

At the consuming end, a message arrives as a generic Message object and must be cast to the appropriate message type. You can use one or more getter methods to extract the message contents. The following code fragment uses the getText method:

Message m = consumer.receive();

if (m instanceof TextMessage) {

  TextMessage message = (TextMessage) m;

  System.out.println("Reading message: " + message.getText());

} else {

  // Handle error

}

### Exception Handling

The root class for exceptions thrown by JMS API methods is JMSException. Catching JMSException provides a generic way of handling all exceptions related to the JMS API. The JMSException class includes the following subclasses, which are described in the API documentation:

* IllegalStateException
* InvalidClientIDException
* InvalidDestinationException
* InvalidSelectorException
* JMSSecurityException
* MessageEOFException
* MessageFormatException
* MessageNotReadableException
* MessageNotWriteableException
* ResourceAllocationException
* TransactionInProgressException
* TransactionRolledBackException

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