Remote procedure call (RPC) systems, including Java RMI, are synchronous -- the caller must block and wait until the called method completes execution, and thus offer no potential for developing loosely coupled enterprise applications without the use of multiple threads. In other words, RPC systems require the client and the server to be available at the same time. However, such tight coupling may not be possible or desired in some applications. Message-Oriented Middleware (MOM) systems provide solutions to such problems. They are based on the asynchronous interaction model, and provide the abstraction of a message queue that can be accessed across a network. Note, however, that messaging here refers to asynchronous requests or events that are consumed by enterprise applications and not humans as in electronic mail (email). These messages contain formatted data that describe specific business actions.

The Java Message Service (JMS), which is designed by Sun Microsystems and several other companies under the Java Community Process as [JSR 914](https://www.jcp.org/en/jsr/detail?id=914), is the first enterprise messaging API that has received wide industry support. The Java Message Service (JMS) was designed to make it easy to develop business applications that asynchronously send and receive business data and events. It defines a common enterprise messaging API that is designed to be easily and efficiently supported by a wide range of enterprise messaging products. JMS supports both messaging models: point-to-point (queuing) and publish-subscribe.

JMS was defined to allow Java application to use enterprise messaging systems. More importantly, it provides a common way for Java applications to access such enterprise messaging systems. JMS falls under middleware, and specifically Message-Oriented Middleware (MOM), which is a relatively low-level of abstraction that runs underneath complementary layers such as database and application adapters, event processing, and business process automation. MOM is becoming an essential component for integrating intra-company operations as it allows separate business components to be combined into a reliable, yet flexible, system.

JMS defines a set of interfaces and semantics that allow Java applications to communicate with other messaging implementations. A JMS implementation is known as a JMS provider. JMS makes the learning curve easy by minimizing the set of concepts a Java developer must learn to use enterprise messaging products, and at the same time it maximizes the portability of messaging applications.

**Architecture**

A JMS application is composed of the following parts:

* A JMS provider: A messaging system that implements the JMS specification.
* JMS clients: Java applications that send and receive messages.
* Messages: Objects that are used to communicate information between JMS clients.
* Administered objects: Preconfigured JMS objects that are created by an administrator for the use of JMS clients.

**Message Delivery Models**

JMS supports two different message delivery models:

1. Point-to-Point (Queue destination): In this model, a message is delivered from a producer to one consumer. The messages are delivered to the destination, which is a queue, and then delivered to one of the consumers registered for the queue. While any number of producers can send messages to the queue, each message is guaranteed to be delivered, and consumed by one consumer. If no consumers are registered to consume the messages, the queue holds them until a consumer registers to consume them.
2. Publish/Subscribe (Topic destination): In this model, a message is delivered from a producer to any number of consumers. Messages are delivered to the topic destination, and then to all active consumers who have *subscribed* to the topic. In addition, any number of producers can send messages to a topic destination, and each message can be delivered to any number of subscribers. If there are no consumers registered, the topic destination doesn't hold messages unless it has durable subscription for inactive consumers. A durable subscription represents a consumer registered with the topic destination that can be inactive at the time the messages are sent to the topic.

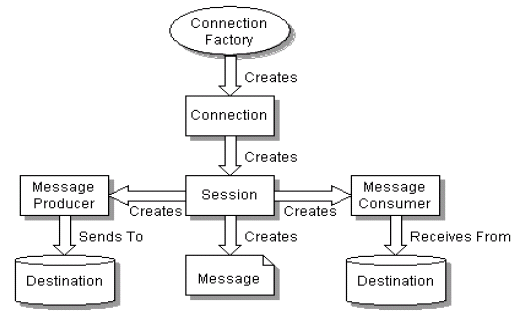
**The JMS Programming Model**

A JMS application consists of a set of application-defined messages and a set of clients that exchange them. JMS clients interact by sending and receiving messages using the JMS API. A message is composed of three parts: *header*, *properties*, and a *body*.

* The header, which is required for every message, contains information that is used for routing and identifying messages. Some of these fields are set automatically, by the JMS provider, during producing and delivering a message, and others are set by the client on a message by message basis.
* Properties, which are optional, provide values that clients can use to filter messages. They provide additional information about the data, such as which process created it, the time it was created. Properties can be considered as an extension to the header, and consist of property name/value pairs. Using properties, clients can fine-tune their selection of messages by specifying certain values that act as selection criteria.
* The body, which is also optional, contains the actual data to be exchanged. The JMS specification defined six type or classes of messages that a JMS provider must support:
  + Message: This represents a message without a message body.
  + StreamMessage: A message whose body contains a stream of Java primitive types. It is written and read sequentially.
  + MapMessage: A message whose body contains a set of name/value pairs. The order of entries is not defined.
  + TextMessage: A message whose body contains a Java string...such as an XML message.
  + ObjectMessage: A message whose body contains a serialized Java object.
  + BytesMessage: A message whose body contains a stream of uninterpreted bytes.

**Producing and Consuming Messages**

Here are the necessary steps to develop clients to produce and consumer messages. Note that there are some common steps that shouldn't be duplicated if the client is both producing and consuming messages. Figure 1 depicts the high-level view of the steps:



*Figure 1: Producing and consuming messages*

**Client to Produce Messages**

1. Use the Java Naming and Directory Interface (JNDI) to find a ConnectionFactory object, or instantiate a ConnectionFactory object directly and set its attributes.

A client uses a connection factory, which is an instance of either QueueConnectionFactory (point-to-point) or TopicConnectionFactory (publish/subscribe), to create a connection to a provider. The following snippet of code demonstrates how to use JNDI to find a connection factory object:

[Copy](https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html#copy)

Copied to Clipboard

Error: Could not Copy

Context ctx = new InitialContext();

ConnectionFactory cf1 = (ConnectionFactory) ctx.lookup("jms/QueueConnectionFactory");

ConnectionFactory cf2 = (ConnectionFactory) ctx.lookup("/jms/TopicConnectionFactory");



Alternatively, you can directly instantiate a connection factory as follows:

ConnectionFactory connFactory = new com.sun.messaging.ConnectionFactory();

1. Use the ConnectionFactory object to create a Connection object. This can be done as follows:

Connection connection = connFactory.createConnection();

Note that you must close all connections you have created. This is done using the Connection.close() method.

1. Use the Connection object to create one or more Session objects, which provide transactional context with which to group a set of sends and receives into an atomic unit of work. A session can be created as follows:

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE)

The createSession() method takes two arguments: the first (false in this case) means that the session is not transacted, and the second means that the session will automatically acknowledge messages when they have been received successfully.

1. Use JNDI to find Destination object(s), or instantiate one directly and configure it by setting its attributes.

A destination object is used by the client to specify the source of messages it consumes and the target of messages it produces. In the point-to-point messaging, destinations are known as queues, and in the publish/subscribe model of messaging, they are known as topics. The following snippet of code demonstrates how to perform a JNDI lookup of a queue named jms/SomeQueue:

Destination dest = (Queue) ctx.lookup("jms/SomeQueue");

Or, you can directly instantiate and configure a destination object as follows:

Queue q = new com.sun.messaging.Queue("world");

1. Use a Session and a Destination object to create the needed MessageProducer object, which are used for sending messages to a destination. The code below shows how that is done. Note that you can create a MessageProducer object without specifying a Destination object, but in that case a Destination object must be specified for each message produced.

MessageProducer producer = session.createProducer(SomeQueue OR SomeTopic);

Once a producer has been created, it can be used to send messages as follows:

producer.send(message);

**Client to Consume Messages**

The first four steps are the same as above.

5. Use a Session object and a Destination object to create any needed MessageConsumer objects that are used for receiving messages. This can be done as follows:

MessageConsumer consumer = session.createConsumer(SomeQueue or SomeTopic);

Note that if topic is being used, then you can use the Session.createDurableSubscriber() to create a durable topic subscriber.

Once the consumer has been created, it can be used to receive messages. Message delivery, however, doesn't begin until you start the connection created earlier, which can be done by calling the start() method:

[Copy](https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html#copy)

Copied to Clipboard

Error: Could not Copy

connection.start();

Message msg = consumer.receive();



A long parameter can be passed to the receive() method to specify a time-out (for example, 3000L for 3 seconds).

It is important to note that the receive() method is used to consume a message synchronously. In order to consume a message asynchronously, a message listener is used.

6. If asynchronous communication is desired, instantiate a MessageListener object and register it with a MessageConsumer object.

A MessageListener object acts as an asynchronous event handler for messages. The MessageListener interface contains one method, onMessage(), which you implement to receive and process the messages.

In the following snippet of code, the class MyListener implements the MessageListener interface. The message listener is registered with a specific consumer using the setMessageListener():

[Copy](https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html#copy)

Copied to Clipboard

Error: Could not Copy

MessageListener listener = new MyListener();

consumer.setMessageListener(listener);



In order to avoid missing messages, the start() method should be called on the connection after the listener has been registered. When message delivery begins, the JMS provides automatically invokes the message listener's onMessage() whenever a message is delivered.

7. Instruct the Connection object to start delivery of messages by calling the start() method.

**Sun Java System Message Queue 3.5**

Sun is one of the principal designers of JMS, and therefore they have been shipping a production implementation of JMS since 1999.

The Sun Java System Message Queue is a currently shipping product, which has been formerly known by the names: Java Message Queue, iPlanet Message Queue for Java, and Sun ONE Message Queue. This product acts as one system for all business messaging needs through support for point-to-point and publish/subscribe messaging models as well as support for synchronous and asynchronous messaging.

Using this product, processes running on different platforms and operating systems can connect to a common Message Queue service to send and receive information. Application developers can focus on the business logic of their applications rather than the low-level details of how to achieve or implement reliable communication across the network.

This product is a leading business integration enterprise message server that provides a standards-based messaging solution. It is available in two edition: Platform Edition and Enterprise Edition.

* The Platform Edition is both the reference implementation of JMS 1.1 specification as well as a product. It is designed for small-scale deployments and development environments. The Platform Edition is included in the J2EE 1.4 reference implementation, and the Sun Java System Application Server Platform Edition 8. It is also available as a free download for use as a standalone product.
* The Enterprise Edition is a high performance messaging system designed for large-scale enterprise deployments for integrating disparate applications. It includes key enterprise features such as scalability, reliability, and advanced security.

**Note**: JMS doesn't define functionality that addresses load balancing, fault tolerance, security, and administration. Such mechanisms must be provided by JMS providers.

Note, however, that if you download the free Platform Edition, it comes with a 90 day trial of the Enterprise Edition

It is worth noting that the product includes a J2EE 1.4 compliant Resource Adapter that enables you to use Sun Java System Message Queue as a JMS provider for any J2EE 1.4 technology-compliant application server. This enables businesses to maximize their investments in current IT assets by being able to continue using their existing application server, but leverage the benefits of Sun Java System Message Queue.

Download and install Java System Message Queue. Once installed, the \bin directory contains a utility to install and uninstall the broker as a Window Service (imqsvcadmin). In addition, it contains the executable for the broker (imqbrokerd) and the following administration tools:

* imqadmin: Administration Console
* imqcmd: Command Utility
* imqobjmgr: Object Manager Utility
* imqusermgr: User Manager Utility
* imqdbmgr: Database Manager Utility
* imqkeytool: Key Tool Utility

**Test Your Installation**

To test your installation, do the following:

1. Run the broker. To do that, go to the bin directory of your installation and issue the following command:

prompt> imqbrokerd -tty

The -tty option causes all logged messages to be displayed to the console...in addition to the log file.

The broker will start and display a few messages before displaying "imqbroker@hostname:7676 ready"

1. Test the broker by issuing the following command in a separate window:

prompt> imqcmd query bkr -u admin -p admin

If everything is fine, it will display information as follows:

Querying the broker specified by:

-------------------------

Host Primary Port

-------------------------

localhost 7676

Version 3.5 SP1

Instance Name imqbroker

Primary Port 7676

Current Number of Messages in System 0

Current Total Message Bytes in System 0

Max Number of Messages in System unlimited (-1)

Max Total Message Bytes in System unlimited (-1)

Max Message Size 70m

Auto Create Queues true

Auto Create Topics true

Auto Created Queue Max Number of Active Consumers 1

Auto Created Queue Max Number of Backup Consumers 0

Cluster Broker List (active)

Cluster Broker List (configured)

Cluster Master Broker

Cluster URL

Log Level INFO

Log Rollover Interval (seconds) 604800

Log Rollover Size (bytes) unlimited (-1)

Successfully queried the broker.



**Sample Application**

To give you a flavor of the effort involved in developing JMS applications, here is a sample application that uses point-to-point communication to send and receive a text message. Note that in this example, the sample application is acting as a sender (sends the message to the queue) and receiver (receives the same message from the queue). As you can see, the application is straightforward and simple to code.

[Copy](https://www.oracle.com/technical-resources/articles/java/intro-java-message-service.html#copy)

Copied to Clipboard

Error: Could not Copy

import javax.jms.\*;

public class HelloMsg {

public static void main(String argv[]) throws Exception {

// The producer and consumer need to get a connection factory and use it to set up

// a connection and a session

QueueConnectionFactory connFactory = new com.sun.messaging.QueueConnectionFactory();

QueueConnection conn = connFactory.createQueueConnection();

// This session is not transacted, and it uses automatic message acknowledgement

QueueSession session = conn.createQueueSession(false, Session.AUTO\_ACKNOWLEDGE);

Queue q = new com.sun.messaging.Queue("world");

// Sender

QueueSender sender = session.createSender(q);

// Text message

TextMessage msg = session.createTextMessage();

msg.setText("Hello there!");

System.out.println("Sending the message: "+msg.getText());

sender.send(msg);

// Receiver

QueueReceiver receiver = session.createReceiver(q);

conn.start();

Message m = receiver.receive();

if(m instanceof TextMessage) {

TextMessage txt = (TextMessage) m;

System.out.println("Message Received: "+txt.getText());

}

session.close();

conn.close();

}

}



To experiment with this sample application, do the following:

1. Copy the HelloMsg class and save it in a new file, HelloMsg.java
2. Compile HelloMsg.java using javac -classpath <MQ\_INSTALL\_DIR>/lib/jms.jar{;|:}<MQ\_INSTALL\_DIR>/lib/img.jar HelloMsg.java

**Note**: The choice of PATH SEPARATER CHARACTER, {;|:}, is platform dependent. ':' on UNIX/Linux, and ';' on Windows

1. Assuming that the imqbrokerd is still running, run HelloMsg as follows:

java -cp <MQ\_INSTALL\_DIR>/lib/jms.jar{;|:};<MQ\_INSTALL\_DIR>/lib/img.jar HelloMsg HelloMsg

Here is the sample output:

Sending the message: Hello there!

Message Received: Hello there!



**Synchronous and Asynchronous Message Consumption**

A JMS client can consume messages either synchronously or asynchronously.

* Synchronous: In this mode, a client receives a message by invoking the receive() method of the MessageConsumer object. The application thread blocks until the method returns, and this has the consequence that if a message is not available, it blocks until a message becomes available or the receive() method times out. Also, note that in this model the client can consume one message at a time.
* Asynchronous: In this mode, the client registers a MessageListener object with a message consumer. This is like a call-back where the client consumes a message when the session invokes the onMessage() method. In other words, the application's thread doesn't block.

**Reliable Messaging**

JMS defines two delivery modes:

1. Persistent messages: Guaranteed to be successfully consumed once and only once. Messages are not lost.
2. Non-persistent messages: Guaranteed to be delivered at most once. Message loss is not a concern.

This, however, is all about performance trade-offs. The more reliable the delivery of messages, the more bandwidth and overhead required to achieve that reliability. Performance can be maximized by producing non-persistent messages, or you can maximize the reliability by producing persistent messages.

For more information on using such advanced features, please see [Chapter 33](https://docs.oracle.com/javaee/1.4/tutorial/doc/JMS.html#wp84181) of the [J2EE 1.4 Tutorial](https://docs.oracle.com/javaee/1.4/tutorial/doc/).

**Message-Driven Beans**

JMS is a mandatory API and service in J2EE platform. A good example is the message-driven bean, one of a family of EJBs specified in EJB 2.0/2.1. The other two EJBs are session beans and entity beans, which can only be called synchronously.

A JMS Message-Driven Bean (MDB) is a JMS message consumer that implements the JMS MessageListener interface. The onMessage() method is invoked when a message is received by the MDB container. Note that you do not invoke remote methods on MDBs (like with other enterprise beans) and as a result there are no home or remote interfaces associated with them. It also worth noting that with J2EE 1.4, MDBs are not limited to JMS; rather, a multiplicity of MDB interfaces can be declared and consumed by application components implementing those interfaces.

If you would like to see sample J2EE applications using the JMS API, please refer to [Chapter 34](https://docs.oracle.com/javaee/1.4/tutorial/doc/JMSJ2EEex.html#wp81128) of the [J2EE 1.4 Tutorial](https://docs.oracle.com/javaee/1.4/tutorial/doc/).

**Conclusion**

The Java Message Service (JMS) makes it easy to develop enterprise applications that asynchronously send and receive business data and events. It defines a common enterprise messaging API that is designed to be easily and efficiently supported by a wide range of enterprise messaging products. JMS supports both messaging models: point-to-point (queuing) and publish-subscribe style of messaging. This article provided a fast track introduction and tutorial to JMS and its programming model.

The JMS specification has been implemented by most application servers and other software vendors. One of these implementations is Sun Java System Message Queue Platform Edition, which is an enterprise message server designed for small-scale deployments and development environments. It implements JMS 1.1 specification and has support for SOAP. This software is available for free; however, note that it doesn't include key enterprise benefits such as scalability, reliability, and advanced security. Such features are available in the Enterprise Edition of the Java System Message Queue, but a 90-day trial version is included in the Platform Edition.

**For More Information**

* [Java Message Service (JMS)](https://www.oracle.com/java/technologies/java-message-service.html)
* [JSR 914: Java Message Service (JMS) API](https://www.jcp.org/en/jsr/detail?id=914)
* [JMS Discussion Forum](https://community.oracle.com/welcome)
* [Sun Java System Message Queue Forum](https://community.oracle.com/community/groundbreakers/java/java-ee-java-enterprise-edition/java_message_service)

**Acknowledgments**

Special thanks to George Tharakan of Sun Microsystems, whose feedback helped me improve this article.