Introduction 1

1.1 Abstract

This specification describes the objectives and functionality of the Java™ Message Service (JMS).

JMS provides a common way for Java programs to create, send, receive and read an enterprise messaging system's messages.

1.2 Overview

Enterprise messaging products (or as they are sometimes called, Message Oriented Middleware products) are becoming an essential component for integrating intra-company operations. They allow separate business components to be combined into a reliable, yet flexible, system.

In addition to the traditional MOM vendors, enterprise messaging products are also provided by several database vendors and a number of internet related companies.

Java language clients and Java language middle tier services must be capable of using these messaging systems. JMS provides a common way for Java language programs to access these systems.

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

Since messaging is peer-to-peer, all users of JMS are referred to generically as *clients*. A JMS *application* is made up of a set of application defined messages and a set of clients that exchange them.

Products that implement JMS do this by supplying a *provider* that implements the JMS interfaces.

1.2.1 Is This a Mail API?

The term *messaging* is quite broadly defined in computing. It is used for describing various operating system concepts; it is used to describe email and fax systems; and here, it is used to describe asynchronous communication between enterprise applications.

Messages, as described here, are asynchronous requests, reports or events that are consumed by enterprise applications, not humans. They contain vital information needed to coordinate these systems. They contain precisely formatted data that describe specific business actions. Through the exchange of these messages each application tracks the progress of the enterprise.

1.2.2 Existing Messaging Systems

Messaging systems are peer-to-peer facilities. In general, each client can send messages to, and receive messages from any client. Each client connects to a messaging agent which provides facilities for creating, sending and receiving messages.

Each system provides a way of addressing messages. Each provides a way to create a message and fill it with data.

Some systems are capable of broadcasting a message to many destinations. Others only support sending a message to a single destination.

Some systems provide facilities for asynchronous receipt of messages (messages are delivered to a client as they arrive). Others support only synchronous receipt (a client must request each message).

Each messaging system typically provides a range of service that can be selected on a per message basis. One important attribute is the lengths to which the system will go to insure delivery. This varies from simple best effort to guaranteed, only once delivery. Other important attributes are message time-to-live, priority and whether a response is required.

1.2.3 JMS Objectives

If JMS provided a union of all the existing features of messaging systems it would be much too complicated for its intended users. On the other hand, JMS is more than an intersection of the messaging features common to all products. It is crucial that JMS include the functionality needed to implement sophisticated enterprise applications.

JMS defines a common set of enterprise messaging concepts and facilities. It attempts to minimize the set of concepts a Java language programmer must learn to use enterprise messaging products. It strives to maximize the portability of messaging applications.

1.2.3.1 JMS Provider

As noted earlier, a JMS provider is the entity that implements JMS for a messaging product.

Ideally, JMS providers will be written in 100% Pure Java so they can run in applets; simplify installation; and, work across architectures and OS's.

An important goal of JMS is to minimize the work needed to implement a provider.

1.2.3.2 JMS Messages

JMS defines a set of message interfaces.

Clients use the message implementations supplied by their JMS provider.

A major goal of JMS is that clients have a consistent API for creating and working with messages that is independent of the JMS provider.

1.2.3.3 JMS Domains

Messaging products can be broadly classified as either *point-to-point* or *publish-subscribe* systems.

Point-to-point (PTP) products are built around the concept of message queues. Each message is addressed to a specific queue; clients extract messages from the queue(s) established to hold their messages.

Publish and subscribe (Pub/Sub) clients address messages to some node in a content hierarchy. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a node's multiple publishers to its multiple subscribers.

JMS provides a set of interfaces that allow the client to send and receive messages in both domains, while supporting the semantics of each domain. JMS also provides client interfaces tailored for each domain. Prior to version 1.1 of the JMS specification, only the client interfaces that were tailored to each domain were available. These interfaces continue to be supported to provide backward compatibility for those who have already implemented JMS clients using them. The preferred approach for implementing clients is to use the domain-independent interfaces. These interfaces, referred to as the "common interfaces", are parents of the domain-specific interfaces.

1.2.3.4 Portability

The primary portability objective is that new, JMS only, applications are portable across products within the same messaging domain.

This is in addition to the expected portability of a JMS client across machine architectures and operating systems (when using the same JMS provider).

Although JMS is designed to allow clients to work with existing message formats used in a mixed language application, portability of such clients is not generally achievable (porting a mixed language application from one product to another is beyond the scope of JMS).

1.2.4 What JMS Does Not Include

JMS does not address the following functionality:

- Load Balancing/Fault Tolerance - Many products provide support for multiple, cooperating clients implementing a critical service. The JMS API does not specify how such clients cooperate to appear to be a single, unified service.
- Error/Advisory Notification - Most messaging products define system messages that provide asynchronous notification of problems or system events to clients. JMS does not attempt to standardize these messages. By following the guidelines defined by JMS, clients can avoid using these messages and thus prevent the portability problems their use introduces.

- Administration - JMS does not define an API for administering messaging products.
- Security - JMS does not specify an API for controlling the privacy and integrity of messages. It also does not specify how digital signatures or keys are distributed to clients. Security is considered to be a JMS provider-specific feature that is configured by an administrator rather than controlled via the JMS API by clients.
- Wire Protocol - JMS does not define a wire protocol for messaging.
- Message Type Repository - JMS does not define a repository for storing message type definitions and it does not define a language for creating message type definitions.

1.3 What Is Required by JMS

The functionality discussed in the specification is required of all JMS providers unless it is explicitly noted otherwise.

Providers of JMS point-to-point functionality are not required to provide publish/subscribe functionality and vice versa.

JMS is also used within the Java 2, Enterprise Edition (J2EE™) platform. See Section 1.4, “Relationship to Other Java APIs” for additional requirements for JMS when it is integrated in that software environment.

1.4 Relationship to Other Java APIs

1.4.1 Java DataBase Connectivity (JDBC™) Software

JMS clients may also use the JDBC API. They may desire to include the use of both the JDBC API and the JMS API in the same transaction. In most cases, this will be achieved automatically by implementing these clients as Enterprise JavaBeans™ components. It is also possible to do this directly with the Java Transaction API (JTA).

1.4.2 JavaBeans™ Components

JavaBeans components can use a JMS session to send/receive messages. JMS itself is an API and the interfaces it defines are not designed to be used directly as JavaBeans components.

1.4.3 Enterprise JavaBeans™ Component Model

The JMS API is an important resource available to Enterprise Java Beans (EJB™) component developers. It can be used in conjunction with other resources like JDBC to implement enterprise services.

The EJB 2.0 specification defines beans that are invoked synchronously via method calls from EJB clients. It also defines a form of asynchronous bean that is invoked when a JMS client sends it a message, called a message-driven bean. The EJB specification supports both synchronous and asynchronous message consumption. In addition, EJB 2.0 specifies how the JMS API participates in bean-managed or container-managed transactions. The EJB 2.0 specification restricts how to use JMS interfaces when implementing EJB clients. Refer to the EJB 2.0 specification for the details.

1.4.4 Java Transaction API (JTA)

The *javax.transaction* package provides a client API for delimiting distributed transactions and an API for accessing a resource's ability to participate in a distributed transaction.

A JMS client may use JTA to delimit distributed transactions; however, this is a function of the transaction environment the client is running in. It is not a feature of JMS.

A JMS provider can optionally support distributed transactions via JTA.

1.4.5 Java Transaction Service (JTS)

JMS can be used in conjunction with JTS to form distributed transactions that combine message sends and receives with database updates and other JTS aware services. Distributed transactions should be handled automatically when a JMS client is run from within an application server such as an Enterprise JavaBeans server; however, it is also possible for JMS clients to program them explicitly.

1.4.6 Java Naming and Directory Interface™ (JNDI) API

JMS clients look up configured JMS objects using the JNDI API. JMS administrators use provider-specific facilities for creating and configuring these objects.

This division of work maximizes the portability of clients by delegating provider-specific work to the administrator. It also leads to more administrable applications because clients do not need to embed administrative values in their code.

1.4.7 Java 2, Enterprise Edition (J2EE) Platform

The J2EE platform specification (version 1.3) requires support for the JMS API as part of the J2EE platform. The J2EE platform specification places certain additional requirements on the implementation of JMS beyond those described in the JMS specification, including the support of both Point-to-Point and Publish/Subscribe domains.

1.4.8 Integration of JMS with the EJB Components

The J2EE platform and EJB specifications describe additional requirements for a JMS provider that is integrated into the J2EE platform. One of the key set of requirements is how JMS message production and JMS message consumption interact with the transactional requirements of container-managed transactions in enterprise beans. Refer to these two specification for full requirements for JMS integration.

This JMS API specification does not address a model for implementing these requirements for integration. Therefore, different JMS provider implementations may implement integration with the J2EE platform and support EJB requirements in different ways.

In the future, an integration point for JMS integration into J2EE platforms will be provided using the J2EE Connector Architecture.

1.5 What is New in JMS 1.1?

In previous versions of JMS, client programming for the Point-to-Point and Pub/Sub domains was done using similar but separate class hierarchies. In

JMS 1.1, there is now a domain-independent approach to programming the client application. This provides several benefits:

- For the client programmer, a simpler programming model
- The ability to engage queues and topics in the same transaction, now that they can be created in the same session
- For the JMS provider, increased opportunity to optimize implementations by pooling thread management

To take advantage of these features, the developer of JMS clients needs to use the domain-independent or "common" APIs. In the future, some of the domain-specific APIs may be deprecated.

In JMS 1.1, all of the classes and methods from JMS 1.0.2b are retained to provide backward compatibility. The semantics of the two messaging domains are retained; the expected behavior of a Point-to-Point domain and a Pub/Sub domain remain the same, as described in Chapter 5, "JMS Point-to-Point Model," and Chapter 6, "JMS Publish/Subscribe Model."

To see details of the changes made to this specification, see Chapter 11, "Change History."

Architecture 2

2.1 Overview

This chapter describes the environment of message-based applications and the role JMS plays in this environment.

2.2 What is a JMS Application?

A JMS application is composed of the following parts:

- JMS Clients - These are the Java language programs that send and receive messages.
- Non-JMS Clients - These are clients that use a message system's native client API instead of JMS. If the application predated the availability of JMS it is likely that it will include both JMS and non-JMS clients.
- Messages - Each application defines a set of messages that are used to communicate information between its clients.
- JMS Provider - This is a messaging system that implements JMS in addition to the other administrative and control functionality required of a full-featured messaging product.
- Administered Objects - Administered objects are preconfigured JMS objects created by an administrator for the use of clients.

2.3 Administration

It is expected that JMS providers will differ significantly in their underlying messaging technology. It is also expected there will be major differences in how a provider's system is installed and administered.

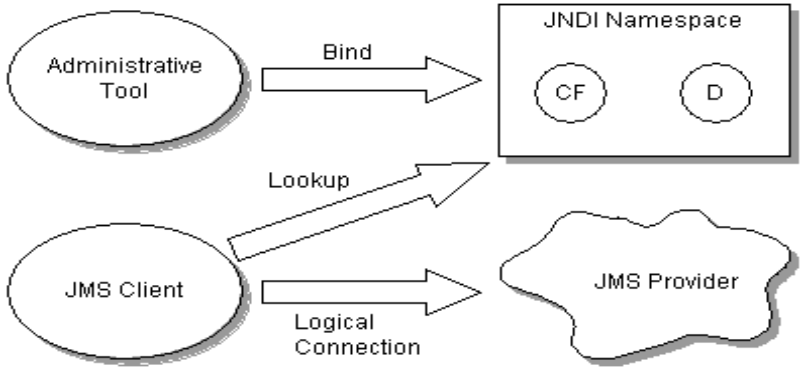
If JMS clients are to be portable, they must be isolated from these proprietary aspects of a provider. This is done by defining JMS administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. The administrator creates them using provider-specific facilities.

There are two types of JMS administered objects:

- ConnectionFactory - This is the object a client uses to create a connection with a provider.
- Destination - This is the object a client uses to specify the destination of messages it is sending and the source of messages it receives.

Administered objects are placed in a JNDI namespace by an administrator. A JMS client typically notes in its documentation the JMS administered objects it requires and how the JNDI names of these objects should be provided to it. Figure 2-1 illustrates how JMS administration ordinarily works.

Figure 2-1 JMS Administration



2.4 Two Messaging Styles

A JMS application can use either the point-to-point (PTP) and the publish-and-subscribe (Pub/Sub) style of messaging, which are described in more detail later in this specification. An application can also combine both styles of messaging in one application. These two styles of messaging are often referred to as messaging domains. JMS provides these two messaging domains because they represent two common models for messaging.

When using the JMS API, a developer can use interfaces and methods that support both models of messaging. When using these interfaces, the behavior of the messaging system may be somewhat different, because the two messaging domains have different semantics. These semantic differences are described in Chapter 5, "JMS Point-to-Point Model," and Chapter 6, "JMS Publish/Subscribe Model."

2.5 JMS Interfaces

JMS is based on a set of common messaging concepts. Each JMS messaging domain - PTP and Pub/Sub - also defines a customized set of interfaces for these concepts.

Table 2-1 Relationship of PTP and Pub/Sub Interfaces

JMS Common Interfaces	**PTP-specific Interfaces**	**Pub/Sub-specific interfaces**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

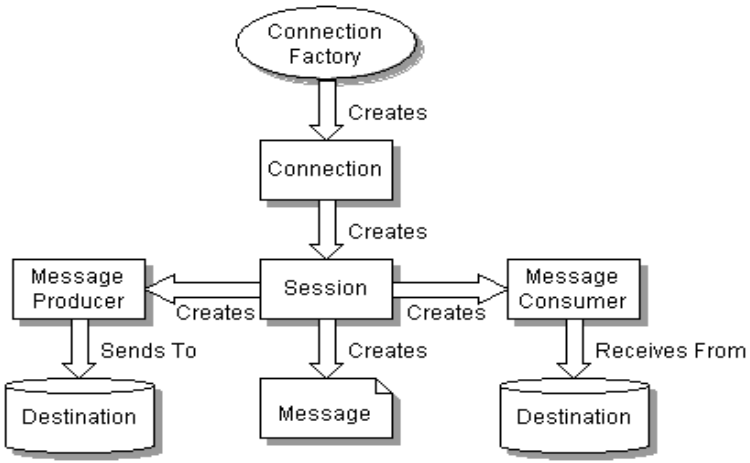
The JMS common interfaces provide a domain-independent view of the PTP and Pub/Sub messaging domains. JMS client programmers are encouraged to use these interfaces to create their client programs.

The following list provides a brief definition of these JMS concepts. See Chapter 4, "JMS Common Facilities," for more details about these concepts.

For the details about the differences in the two messaging domains, see Chapter 5, "JMS Point-to-Point Model," and Chapter 6, "JMS Publish/Subscribe Model."

- ConnectionFactory - an administered object used by a client to create a Connection
- Connection - an active connection to a JMS provider
- Destination - an administered object that encapsulates the identity of a message destination
- Session - a single-threaded context for sending and receiving messages
- MessageProducer - an object created by a Session that is used for sending messages to a destination
- MessageConsumer - an object created by a Session that is used for receiving messages sent to a destination

Figure 2-1 Overview of JMS object relationships



The term *consume* is used in this document to mean the receipt of a message by a JMS client; that is, a JMS provider has received a message and has given it to its client. Since JMS supports both synchronous and asynchronous receipt of messages, the term *consume* is used when there is no need to make a distinction between them.

The term *produce* is used as the most general term for sending a message. It means giving a message to a JMS provider for delivery to a destination.

2.6 Developing a JMS Application

Broadly speaking, a JMS application is one or more JMS clients that exchange messages. The application may also involve non-JMS clients; however, these clients use the JMS provider's native API in place of JMS.

A JMS application can be architected and deployed as a unit. In many cases, JMS clients are added incrementally to an existing application.

The message definitions used by an application may originate with JMS, or they may have been defined by the non-JMS part of the application.

2.6.1 Developing a JMS Client

A typical JMS client executes the following JMS setup procedure:

- Use JNDI to find a ConnectionFactory object
- Use JNDI to find one or more Destination objects
- Use the ConnectionFactory to create a JMS Connection with message delivery inhibited

- Use the Connection to create one or more JMS Sessions
- Use a Session and the Destinations to create the MessageProducers and MessageConsumers needed
- Tell the Connection to start delivery of messages

At this point a client has the basic JMS setup needed to produce and consume messages.

2.7 Security

JMS does not provide features for controlling or configuring message integrity or message privacy.

It is expected that many JMS providers will provide such features. It is also expected that configuration of these services will be handled by provider-specific administration tools. Clients will get the proper security configuration as part of the administered objects they use.

2.8 Multithreading

JMS could have required that all its objects support concurrent use. Since support for concurrent access typically adds some overhead and complexity, the JMS design restricts its requirement for concurrent access to those objects that would naturally be shared by a multithreaded client. The remainder are designed to be accessed by one logical thread of control at a time.

Table 2-2 JMS Objects that Support Concurrent Use

JMS Object	Supports Concurrent Use
Destination	YES
ConnectionFactory	YES
Connection	YES
Session	NO
MessageProducer	NO
MessageConsumer	NO

JMS defines some specific rules that restrict the concurrent use of Sessions. Since they require more knowledge of JMS specifics than we have presented at

this point, they will be described later. Here we will describe the rationale for imposing them.

There are two reasons for restricting concurrent access to Sessions. First, Sessions are the JMS entity that supports transactions. It is very difficult to implement transactions that are multithreaded. Second, Sessions support asynchronous message consumption. It is important that JMS *not* require that client code used for asynchronous message consumption be capable of handling multiple, concurrent messages. In addition, if a Session has been set up with multiple, asynchronous consumers, it is important that the client is not forced to handle the case where these separate consumers are concurrently executing. These restrictions make JMS easier to use for typical clients. More sophisticated clients can get the concurrency they desire by using multiple sessions.

2.9 Triggering Clients

Some clients are designed to periodically wake up and process messages waiting for them. A message-based application triggering mechanism is often used with this style of client. The trigger is typically a threshold of waiting messages, etc.

JMS does not provide a mechanism for triggering the execution of a client. Some providers may supply such a triggering mechanism via their administrative facilities.

2.10 Request/Reply

JMS provides the *JMSReplyTo* message header field for specifying the Destination where a reply to a message should be sent. The *JMSCorrelationID* header field of the reply can be used to reference the original request. See Section 3.4, “Message Header Fields,” for more information.

In addition, JMS provides a facility for creating temporary queues and topics that can be used as a unique destination for replies.

Enterprise messaging products support many styles of request/reply, from the simple “one message request yields a one message reply” to “one message request yields streams of messages from multiple respondents.” Rather than architect a specific JMS request/reply abstraction, JMS provides the basic facilities on which many can be built.

For convenience, JMS defines request/reply helper classes (classes written using JMS) for both the PTP and Pub/Sub domains that implement a basic form of request/reply. JMS providers and clients may provide more specialized implementations.

JMS Point-to-Point Model 5

5.1 Overview

Point-to-point systems are about working with queues of messages. They are point-to-point in that a client sends a message to a specific queue. Some PTP systems blur the distinction between PTP and Pub/Sub by providing system clients that automatically distribute messages.

It is common for a client to have all its messages delivered to a single queue.

Like any generic mailbox, a queue can contain a mixture of messages. And, like real mailboxes, creating and maintaining each queue is somewhat costly. Most queues are created administratively and are treated as static resources by their clients.

The JMS PTP model defines how a client works with queues: how it finds them, how it sends messages to them, and how it receives messages from them.

This chapter describes the semantics of the Point-to-Point model. A JMS provider that supports the Point-to-Point model must deliver the semantics described here.

Whether a JMS client program uses the PTP domain-specific interfaces, or the common interfaces that are described in Chapter 4, “JMS Common Facilities,” the client program *must* be guaranteed the same behavior.

Table 5-1 shows the interfaces that are specific to the PTP domain and the JMS common interfaces. The common interfaces are preferred for creating JMS application programs, because they are domain-independent.

Table 5-1 PTP Domain Interfaces and JMS Common Interfaces

PTP Domain Interfaces	**JMS Common Interfaces** ***Preferred***
QueueConnectionFactory	ConnectionFactory
QueueConnection	Connection
Queue	Destination
QueueSession	Session
QueueSender	MessageProducer
QueueReceiver	MessageConsumer

5.2 Queue Management

JMS does not define facilities for creating, administering, or deleting long-lived queues (it does provide such a mechanism for *TemporaryQueue*s). Since most clients use statically defined queues, this is not a problem.

5.3 Queue

A *Queue* object encapsulates a provider-specific queue name. It is the way a client specifies the identity of a queue to JMS methods.

The actual length of time messages are held by a queue and the consequences of resource overflow are not defined by JMS.

See Section 4.2, “Administered Objects,” for more information about JMS *Destination* objects.

5.4 TemporaryQueue

A *TemporaryQueue* is a unique *Queue* object created for the duration of a *Connection* or *QueueConnection*. It is a system-defined queue that can be consumed only by the *Connection* or *QueueConnection* that created it.

See Section 4.4.3, “Creating Temporary Destinations,” for more information.

5.5 QueueConnectionFactory

A client uses a *QueueConnectionFactory* to create *QueueConnection*s with a JMS PTP provider.

See Section 4.2, “Administered Objects,” for more information about JMS *ConnectionFactory* objects.

5.6 QueueConnection

A *QueueConnection* is an active connection to a JMS PTP provider. A client uses a *QueueConnection* to create one or more *QueueSession*s for producing and consuming messages.

See Section 4.3, “Connection,” for more information.

5.7 QueueSession

A *QueueSession* provides methods for creating *QueueReceiver*s, *QueueSender*s, *QueueBrowser*s, and *TemporaryQueue*s.

If there are messages that have been received but not acknowledged when a *QueueSession* terminates, these messages must be retained and redelivered when a consumer next accesses the queue.

See Section 4.4, “Session,” for more information.

5.8 QueueReceiver

A client uses a *QueueReceiver* for receiving messages that have been delivered to a queue.

Although it is possible to have two sessions with a *QueueReceiver* for the same queue, JMS does not define how messages are distributed between the *QueueReceiver*s.

If a *QueueReceiver* specifies a message selector, the messages that are not selected remain on the queue. By definition, a message selector allows a *QueueReceiver* to skip messages. This means that when the skipped messages are eventually read, the total ordering of the reads does not retain the partial order defined by each message producer. Only *QueueReceiver*s without a message selector will read messages in message producer order.

For more information, see Section 4.5, “MessageConsumer.” If *MessageConsumer* is consuming messages from a *Queue*, then it must behave as described here in Section 5.8, “QueueReceiver.”

A client uses a *MessageProducer* or *QueueSender* to send messages to a *Queue*.

For more information, see Section 4.6, “MessageProducer.”

5.9 QueueBrowser

A client uses a *QueueBrowser* to look at messages on a queue without removing them. A *QueueBrowser* can be created from a *Session* or a *QueueSession*.

The browse methods return a *java.util.Enumeration* that is used to scan the queue’s messages. It may be an enumeration of the entire content of a queue, or it may contain only the messages matching a message selector.

Messages may be arriving and expiring while the scan is done. JMS does not require the content of an enumeration to be a static snapshot of queue content. Whether these changes are visible or not depends on the JMS provider.

5.10 QueueRequestor

JMS provides a *QueueRequestor* helper class to simplify making service requests.

The *QueueRequestor* constructor is given a *QueueSession* and a destination queue. It creates a *TemporaryQueue* for the responses and provides a *request* method that sends the request message and waits for its reply.

This is a basic request/reply abstraction that should be sufficient for most uses. JMS providers and clients can create more sophisticated versions.

5.11 Reliability

A queue is typically created by an administrator and exists for a long time. It is always available to hold messages sent to it, whether or not the client that consumes its messages is active. For this reason, a client does not have to take any special precautions to insure that it does not miss messages.

JMS Publish/Subscribe Model 6

6.1 Overview

The JMS Pub/Sub model defines how JMS clients publish messages to, and subscribe to messages from, a well-known node in a content-based hierarchy. JMS calls these nodes *topics*.

In this section, the terms *publish* and *subscribe* are used in place of the more generic terms *produce* and *consume* used previously.

A topic can be thought of as a mini message broker that gathers and distributes messages addressed to it. By relying on the topic as an intermediary, message publishers are kept independent of subscribers and vice versa. The topic automatically adapts as both publishers and subscribers come and go.

Publishers and subscribers are *active* when the Java objects that represent them exist. JMS also supports the optional *durability* of subscribers that 'remembers' the existence of them while they are inactive.

This chapter describes the semantics of the Publish/Subscribe model. A JMS provider that supports the Publish/Subscribe model must deliver the semantics described here.

Whether a JMS client program uses the Pub/Sub domain-specific interfaces, or the common interfaces that are described in Chapter 4, "JMS Common Facilities," the clien program *must* be guaranteed the same behavior.

Table 6-1 shows the interfaces that are specific to the PTP domain and the JMS common interfaces. The common interfaces are preferred for creating JMS application programs, because they are domain-independent.

Table 6-1 Pub/Sub Domain Interfaces and JMS Common Interfaces

Pub/Sub Domain interfaces	**JMS Common interfaces** ***Preferred***
TopicConnectionFactory	ConnectionFactory
TopicConnection	Connection
Topic	Destination
TopicSession	Session
TopicPublisher	MessageProducer
TopicSubscriber	MessageConsumer

6.2 Pub/Sub Latency

Since there is typically some latency in all pub/sub systems, the exact messages seen by a subscriber may vary depending on how quickly a JMS provider propagates the existence of a new subscriber and the length of time a provider retains messages in transit.

For instance, some messages from a distant publisher may be missed because it may take a second for the existence of a new subscriber to be propagated system-wide. When a new subscriber is created, it may receive messages sent earlier because a provider may still have them available.

JMS does not define the exact semantics that apply during the interval when a pub/sub provider is adjusting to a new client. JMS semantics only apply once the provider has reached a ‘steady state’ with respect to a new client.

6.3 Durable Subscription

Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client will only see the messages published on a topic while its

subscriber is active. If the subscriber is not active, it is missing messages published on its topic.

At the cost of higher overhead, a subscriber can be made *durable*. A *durable subscriber* registers a *durable subscription* with a unique identity that is retained by JMS. Subsequent subscriber objects with the same identity resume the subscription in the state it was left in by the prior subscriber. If there is no active subscriber for a durable subscription, JMS retains the subscription's messages until they are received by the subscription or until they expire.

All JMS providers must be able to run JMS applications that dynamically create and delete durable subscriptions. Some JMS providers may, in addition, provide facilities to administratively configure durable subscriptions. If a durable subscription has been administratively configured, it is valid for it to silently override the subscription specified by the client.

An *inactive* durable subscription is one that exists but does not currently have a message consumer subscribed to it.

6.4 Topic Management

Some products require that topics be statically defined with associated authorization control lists, and so on; others don't even have the concept of topic administration.

JMS does not define facilities for creating, administering, or deleting topics.

A special type of topic called a *TemporaryTopic* is provided for creating a *Topic* that is unique to a *TopicConnection*. See Section 6.6, "TemporaryTopic," for more details.

6.5 Topic

A *Topic* object encapsulates a provider-specific topic name. It is the way a client specifies the identity of a topic to JMS methods.

Many Pub/Sub providers group topics into hierarchies and provide various options for subscribing to parts of the hierarchy. JMS places no restrictions on what a *Topic* object represents. It might be a leaf in a topic hierarchy, or it might be a larger part of the hierarchy (for subscribing to a general class of information).

The organization of topics and the granularity of subscriptions to them is an important part of a Pub/Sub application's architecture. JMS does not specify a policy for how this should be done. If an application takes advantage of a provider-specific topic grouping mechanism, it should document this. If the application is installed using a different provider, it is the job of the administrator to construct an equivalent topic architecture and create equivalent *Topic* objects.

6.6 TemporaryTopic

A *TemporaryTopic* is a unique *Topic* object created for the duration of a *Connection* or *TopicConnection*. It is a system-defined *Topic* that can be consumed only by the *Connection* or *TopicConnection* that created it.

By definition, it does not make sense to create a durable subscription to a temporary topic. To do this is a programming error that may or may not be detected by a JMS provider.

See Section 4.4.3, "Creating Temporary Destinations," for more information.

6.7 TopicConnectionFactory

A client uses a *TopicConnectionFactory* to create *TopicConnection*s with a JMS Pub/Sub provider.

See Section 4.2, "Administered Objects," for more information about JMS *ConnectionFactory* objects.

6.8 TopicConnection

A *TopicConnection* is an active connection to a JMS Pub/Sub provider. A client uses a *TopicConnection* to create one or more *TopicSession*s for producing and consuming messages.

See Section 4.3, "Connection," for more information.

6.9 TopicSession

A *TopicSession* provides methods for creating *TopicPublisher*s, *TopicSubscriber*s, and *TemporaryTopic*s. It also provides the *unsubscribe* method for deleting its client's durable subscriptions.

If there are messages that have been received but not acknowledged when a *TopicSession* terminates, a durable *TopicSubscriber* must retain and redeliver them; a nondurable subscriber need not do so.

See Section 4.4, “Session,” for more information.

6.10 TopicPublisher

A client uses a *TopicPublisher* for publishing messages on a topic. *TopicPublisher* is the Pub/Sub variant of a JMS *MessageProducer*. Messages can also be sent to a *Topic* using a *MessageProducer*. See Section 4.6, “MessageProducer,” for a description of its common features.

6.11 TopicSubscriber

A client uses a *TopicSubscriber* for receiving messages that have been published to a topic. *TopicSubscriber* is the Pub/Sub variant of a JMS *MessageConsumer*. For more information, see Section 4.5, “MessageConsumer.”

Ordinary *TopicSubscriber*s are not durable. They only receive messages that are published while they are active.

Messages filtered out by a subscriber’s message selector will never be delivered to the subscriber. From the subscriber’s perspective, they simply don’t exist.

In some cases, a connection may both publish and subscribe to a topic. The subscriber *NoLocal* attribute allows a subscriber to inhibit the delivery of messages published by its own connection.

A *TopicSession* allows the creation of multiple *TopicSubscriber*s per destination, it will deliver each message for a destination to each *TopicSubscriber* eligible to receive it. Each copy of the message is treated as a completely separate message. Work done on one copy has no effect on any other; acknowledging one does not acknowledge any other; one message may be delivered immediately, while another waits for its consumer to process messages ahead of it.

6.11.1 Durable TopicSubscriber

If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable

TopicSubscriber. A durable *TopicSubscriber* can be created by a *Session* or by a *TopicSession*. JMS retains a record of this durable subscription and insures that all messages from the *Topic*'s publishers are retained until either they are acknowledged by this durable subscriber or they have expired.

Sessions with durable subscribers must always provide the same client identifier. In addition, each client must specify a name that uniquely identifies (within client identifier) each durable subscription it creates. Only one session at a time can have a *TopicSubscriber* for a particular durable subscription. See Section 4.3.2, "Client Identifier," for more information.

A client can change an existing durable subscription by creating a durable *TopicSubscriber* with the same name and a new topic and/or message selector, or NoLocal attribute. Changing a durable subscription is equivalent to deleting and recreating it.

Sessions and *TopicSession*s provide the *unsubscribe* method for deleting a durable subscription created by their client. This deletes the state being maintained on behalf of the subscriber by its provider. It is erroneous for a client to delete a durable subscription while it has an active *TopicSubscriber* for it or while a message received by it is part of a current transaction or has not been acknowledged in the session.

6.12 Recovery and Redelivery

Unacknowledged messages of a nondurable subscriber should be able to be recovered for the lifetime of that nondurable subscriber. When a nondurable subscriber terminates, messages waiting for it will likely be dropped whether or not they have been acknowledged.

Only durable subscriptions are reliably able to recover unacknowledged messages.

Sending a message to a topic with a delivery mode of PERSISTENT does not alter this model of recovery and redelivery. To ensure delivery, a *TopicSubscriber* should establish a durable subscription.

6.13 Administering Subscriptions

Ideally, publishers and subscribers are dynamically registered by a provider when they are created. From the client viewpoint this is always the case. From

the administrator's viewpoint, other tasks may be needed to support the creation of publishers and subscribers.

The amount of resources allocated for message storage and the consequences of resource overflow are not defined by JMS.

6.14 TopicRequestor

JMS provides a *TopicRequestor* helper class to simplify making service requests.

The *TopicRequestor* constructor is given a *TopicSession* and a destination topic. It creates a *TemporaryTopic* for the responses and provides a *request()* method that sends the request message and waits for its reply.

This is a basic request/reply abstraction that should be sufficient for most uses. JMS providers and clients are free to create more sophisticated versions.

6.15 Reliability

When all messages for a topic must be received, a durable subscriber should be used. JMS insures that messages published while a durable subscriber is inactive are retained by JMS and delivered when the subscriber subsequently becomes active.

Nondurable subscribers should be used only when missed messages are tolerable.

Table 6-2 Pub/Sub Reliability

How Published	**Nondurable Subscriber**	**Durable Subscriber**
NON_PERSISTENT	at-most-once (missed if inactive)	at-most-once
PERSISTENT	once-and-only-once (missed if inactive)	once-and-only-once

JMS Example Code 9

This chapter gives some code examples that show how a JMS client could use the JMS API. It also demonstrates how to use several message types. The examples use methods that support a unified messaging model: these examples work with either Point-to-Point or Publish/Subscribe messaging. This is the recommended approach to working with the JMS API.

In earlier versions of the JMS Specification, only the separate interfaces for each messaging domain (Point-to-Point or Pub/Sub) were supported, and the client was programmed to use one messaging domain or the other. Now, the JMS client can be programmed using the JMS common interfaces.

In the example program, a client application sends and receives stock quote information. The messages the client application receives are from a stock quote service that sends out stock quote messages. The stock quote service is not described in the example.

To simplify the example, no exception-handling code is included.

This chapter describes the steps for creating the correct environment for sending and receiving a message.

After describing these basic functions, this chapter describes how to perform some other common functions, such as using message selectors.

9.1 Preparing to Send and Receive Messages

Here are the basic steps to establish a connection and prepare to send and receive messages.

- Get a *ConnectionFactory* and *Destination*
- Create a *Connection* and *Session*
- Create a *MessageConsumer*
- Create a *MessageProducer*

9.1.1 Getting a ConnectionFactory

Both the message producer and message consumer (the sender and receiver) need to get a *ConnectionFactory* and use it to set up both a *Connection* and a *Session*.

An administrator typically has created and configured a *ConnectionFactory* for the JMS client's use. The client program typically uses the JNDI API to look up the *ConnectionFactory*.

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

ConnectionFactory connectionFactory;

Context messaging = new InitialContext();
connectionFactory = (ConnectionFactory)
    messaging.lookup("ConnectionFactory");
```

9.1.2 Getting a Destination

An administrator has created and configured a *Queue* named "StockSource" which is where stock quote messages are sent and received. Again, the destination can be looked up using the JNDI API.

```
Queue stockQueue;

stockQueue = (Queue)messaging.lookup("StockSource");
```

9.1.3 Creating a Connection

Having obtained the *ConnectionFactory*, the client program uses it to create a *Connection*.

```
Connection connection;

connection = ConnectionFactory.createConnection();
```

9.1.4 Creating a Session

Having obtained the *Connection*, the client program uses it to create a *Session*. The *Session* is used to create a *MessageProducer* (to send messages) or a *MessageConsumer* (to receive messages).

The *Connection.createSession* method takes two parameters:

- A boolean indicating whether this session is transacted or not
- The mode of acknowledging message receipt

```
Session session;

/* Session is not transacted,
*  uses AUTO_ACKNOWLEDGE for message
*  acknowledgement
*/
session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

9.1.5 Creating a MessageProducer

Having obtained the *Session*, the client program uses the *Session* to create a *MessageProducer*. The *MessageProducer* object is used to send messages to the destination. The *MessageProducer* is created by using the *Session.createProducer* method, supplying as a parameter the destination to which the messages are sent.

```
MessageProducer sender;


/* Value in stockQueue previously looked up in the JNDI
*  createProducer takes a Destination
*/

sender = session.createProducer(stockQueue);
```

9.1.6 Creating a MessageConsumer

Messages can be consumed either synchronously or asynchronously. This example shows how to create a message consumer that consumes messages synchronously. See Section 9.3.1, “Receiving Messages Asynchronously,” to learn more about consuming messages asynchronously.

A *MessageConsumer* is used to receive messages from the destination, which in this example is the *Queue* "StockQuote." A *MessageConsumer* is created using the *Session.createConsumer* method, supplying one parameter, the destination from which messages are received.

```
MessageConsumer receiver;

/* Value in stockQueue previously looked up in the JNDI
*  createConsumer takes a Destination
*/

receiver = session.createConsumer(stockQueue);
```

9.1.7 Starting Message Delivery

Up until this point, delivery of messages has been inhibited so that the preceding setup could be done without being interrupted with asynchronously delivered messages. Now that the setup is complete, the *Connection* is told to begin the delivery of messages to its *MessageConsumer.*

```
connection.start();
```

9.1.8 Using a TextMessage

There are several JMS *Message* formats. For this example, the stock quote information is sent as a text string that is read and displayed by the client.

The following demonstrates how to create such a message:

```
String        stockData;    /* Stock information as a string */
TextMessage  message;

/* Set the message's text to be the stockData string */
message = session.createTextMessage();
message.setText(stockData);
```

9.2 Sending and Receiving Messages

Now that the setup of the *Session* is complete, you can send and receive messages. This section describes how to:

- Create a message
- Send a message

- Receive a message synchronously

9.2.1 Sending a Message

To send a message, use the *MessageProducer.send* method, supplying a *Message* object for the method's parameter.

```
/* Send the message */
sender.send(message);
```

9.2.2 Receiving a Message Synchronously

To receive the next message in the *Queue*, you can use the *MessageConsumer.receive* method. This call blocks indefinitely until a message arrives on the *Queue*. The same method can be used to receive from a *Topic*.

```
TextMessage stockMessage;
stockMessage = (TextMessage)receiver.receive();
```

To limit the amount of time that the client blocks, use a timeout parameter with the *receive* method. If no messages arrive by the end of the timeout, then the *receive* method returns. The timeout parameter is expressed in milliseconds.

```
TextMessage stockMessage;

/* Wait 4 seconds for a message */
TextMessage = (TextMessage)receiver.receive(4000);
```

9.2.3 Unpacking a TextMessage

The stock quote information is sent using a *TextMessage*. To get the information from the message, use the *TextMessage.getText* method. It returns the message content as a string.

```
String newStockData;    /* Stock information as a string */

newStockData = message.getText();
```

9.3 Other Messaging Features

This section goes beyond basic messaging functions, and describes how to perform some other common messaging functions:

- Create an asynchronous *MessageListener*
- Use a message selector to filter message delivery
- Create a durable subscription to a *Topic*
- Re-connect to a *Topic using a* durable subscription

9.3.1 Receiving Messages Asynchronously

In order to receive message asynchronously as they are delivered to the message consumer, the client program needs to create a message listener that implements the *MessageListener* interface. An implementation of the *MessageListener* interface, called *StockListener.java*, might look like this:

```
import javax.jms.*;

public class StockListener implements MessageListener
{

    public void onMessage(Message message) {
       /* Unpack and handle the messages received */
       ...
    }
 }
```

The client program registers the *MessageListener* object with the *MessageConsumer* object in the following way:

```
StockListener myListener = new StockListener();

/* Receiver is MessageConsumer object */
receiver.setMessageListener(myListener);
```

The *Connection* must be started for the message delivery to begin. The *MessageListener* is asynchronously notified whenever a message has been published to the *Queue*. This is done via the *onMessage* method in the *MessageListener* interface. It is up to the client to process the message there.

```
public void onMessage(Message message)

{

    String newStockData;

    /* Unpack and handle the messages received */

    newStockData = message.getText();
    if(...)
    {
        /* Logic related to the data */

    }

}
```

9.3.2 Using Message Selection

A client program may be interested in receiving only certain stock quotes. A message selector can be used to achieve this goal. Message selectors work against properties that are assigned to the message.

In this example, the client program is only interested in technology related stocks. The sender of the messages assigns a value to a message property called *StockSector*. The values the sender assigns include “Technology”, “Financial”, “Manufacturing”, “Emerging”, and “Global”. The message sender assigns these property values by using the *Message.setStringProperty* method.

```
String stockData;    /* Stock information as a String */
TextMessage message;

/* Set the message's text to be the stockData string */

message = session.createTextMessage();
message.setText(stockData);

/* Set the message property 'StockSector' */
message.setStringProperty("StockSector", "Technology");
```

When the client program that receives the stock quote messages creates *MessageConsumer* is created, the client program can create a message selector string to determine which messages it will receive.

```
String selector;

selector = new String("(StockSector = 'Technology')");
```

This string is specified when the *MessageConsumer* is created:

```
MessageConsumer receiver;
receiver = session.createConsumer(stockQueue, selector);
```

The client program receives only messages related to the Technology sector.

9.3.3 Using Durable Subscriptions

Durable subscriptions are used to receive messages from a *Topic*. When a JMS client creates a durable subscription, the client can later disconnect from the *Topic*. When the client program re-connects, it can receive the messages that arrived while it was disconnected. In this example, the *Destination* provides information about news updates.

9.3.3.1 Creating a Durable Subscription

The following example sets up durable subscription that gets messages from a *Topic*. First, the client program must perform the usual setup steps of looking up *ConnectionFactory* and a *Destination*, and creating a *Connection* and *Session*, as described in Section 9.1, “Preparing to Send and Receive Messages.”

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

/* Look up connection factory */
ConnectionFactory connectionFactory;
Context messaging = new InitialContext();
connectionFactory =
(ConnectionFactory) Messaging.lookup("ConnectionFactory")

/* Look up destination */
Topic newsFeedTopic;
newsFeedTopic = messaging.lookup("BreakingNews");

/* Create connection and session */

Connection connection;
Session session;
connection = ConnectionFactory.createConnection();
```

```
session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

Having performed the normal setup, the client program can now create a durable subscriber to the destination. To do this, the client program creates a durable *TopicSubscriber*, using *session.CreateDurableSubscriber*. The name *mySubscription* is used as an identifier of the durable subscription.

```
session.createDurableSubscriber(newsFeedTopic,"mySubscription");
```

At this point, the client program can start the connection and start to receive messages.

9.3.3.2 Reconnecting to a Topic using a Durable Subscription

To re-connect to a *Topic* that has an existing durable subscription, the client program can simply call *session.CreateDurableSubscriber* again, using the same parameters that it previously used. A client program may be intermittently connected. Using durable subscriptions allows messages to still be available to a client program that consumes from a *Topic*, even though the client program was not continuously connected.

```
/* Reconnect to a durable subscription */
session.createDurableSubscriber(newsFeedTopic, "mySubscription");
```

This reconnects the client program to the *Topic*, and any messages that arrived while the client was disconnected are delivered. However, there are some important restrictions to be aware of:

- The client must be attached to the same C*onnection*.
- The *Destination* and subscription name must be the same.
- If a message selector was specified, it must also be the same.

If these conditions are not met, then the durable subscription is deleted, and a new subscription is created.

9.4 JMS Message Types

There are five JMS message types. This section provides an example of how to create and unpack each of these types. In each example, the data sent in the message is stock-quote-related data. In all cases, the code that creates the actual content of the messages is omitted.

9.4.1 Creating a TextMessage

In this example, the stock quote information is sent as a *TextMessage*. A *TextMessage* carries the message as a text string that can be read by the client.

The following code demonstrates how to create such a message:

```
String        stockData;    /* Stock information as a string */
TextMessage  message;

message = session.createTextMessage();

/* Set the stockData string to the message body */

message.setText(stockData);
```

9.4.2 Unpacking a TextMessage

To unpack a *TextMessage*, the client uses the *Message.getText* method.

```
String stockInfo;    /* String to hold stock info */


stockInfo = message.getText();
```

9.4.3 Creating a BytesMessage

The stock quote information could be in a binary format that the server knows how to construct and that the client program knows how to interpret and display as a stock quote. This is sent as a *BytesMessage*.

Such a message can be constructed in the following way:

```
byte[]         stockData;    /* Stock information as a byte array */
BytesMessage  message;

message = session.createBytesMessage();
message.writeBytes(stockData);
```

9.4.4 Unpacking a BytesMessage

When the *BytesMessage* is received, it can be unpacked in the following manner:

```
byte[]      stockInfo; /* Byte array to hold stock information */
int       length;

length = message.readBytes(stockData);
```

The message body is copied to the byte array. The client program can then begin reading and interpreting the data.

9.4.5 *Creating a MapMessage*

Each stock message sent by the server could be a map of various stock quote name/value pairs, using a *MapMessage*. For example, it could contain entries for:

- Stock quote name - represented as a *String*
- Current value - represented as a *double*
- Time of quote - represented as a *long*
- Last change - represented as a *double*
- Stock information - represented as a *String*

To construct the *MapMessage*, the client program uses the various set method (*setString*, *setLong*, and so forth) that are associated with *MapMessage*, and sets each named value in the *MapMessage*.

```
String stockName;     /* Name of the stock */
double stockValue;    /* Current value of the stock */
long   stockTime;      /* Time the stock quote was updated */
double stockDiff;     /* the +/- change in the stock quote*/
String stockInfo;     /* Other information on this stock */
MapMessage message;

message = session.createMapMessage();
```

Note that the following can be set in any order.

```
/* First parameter is the name of the map element,
*  second is the value
*/

message.setString("Name",  "SUNW");
message.setDouble("Value", stockValue);
message.setLong("Time",    stockTime);
message.setDouble("Diff",  stockDiff);
```

```
message.setString("Info",  "Recent server announcement causes market
                interest");
```

9.4.6 Unpacking a MapMessage

To unpack the *MapMessage*, the client program uses the various get methods associated with *MapMessage* to get the values in the named *MapMessage* elements. In the following example, the client program expects certain *MapMessage* elements.

```
String stockName;      /*Name of the stock */
double stockValue;     /* Current value of the stock */
long   stockTime;      /* Time of the stock update */
double stockDiff;      /* +/- change in the stock */
String stockInfo;      /* Information on this stock */
```

The data is retrieved from the message by using a get method and providing the name of the value desired. The elements from the *MapMessage* can be obtained in any order.

```
stockName  = message.getString("Name");
stockDiff  = message.getDouble("Diff");
stockValue = message.getDouble("Value");
stockTime  = message.getLong("Time");
```

If a client program needs to get a list of the elements in a *MapMessage*, it can use the method *MapMessage.getMapNames*.

9.4.7 Creating a StreamMessage

In a similar fashion to the *MapMessage*, an application could send a message consisting of various fields written in sequence to the message, each in their own primitive type. To do this, the would use a *StreamMessage*. Here's the primitive types assigned to each item in the stock quote message.

- Stock quote name - *String*
- Current value - *double*
- Time of quote - *long*
- Last change - *double*
- Stock information - *String*

The client program might be interested in only some of the message fields, but in the case of a *StreamMessage*, the client has to read and potentially discard each field in turn.

In the following example, the values for each of the following has already been set.:

```
String stockName;      /* Name of the stock */
double stockValue;     /* Current value of the stock */
long   stockTime;      /* Time of the stock update */
double stockDiff;      /* +/- change in the stock quote */
String stockInfo;      /* Information on this stock*/
StreamMessage message;

/* Create message */
message = session.createStreamMessage();
```

The following elements have to be written to the *StreamMessage* in the order they will be read. Notice that they are not separately named properties, as in *MapMessage*.

```
/* Set data for message */
message.writeString(stockName);
message.writeDouble(stockValue);
message.writeLong(stockTime);
message.writeDouble(stockDiff);
message.writeString(stockInfo);
```

9.4.8 Unpacking a StreamMessage

The elements of a *StreamMessage* have to be read in the order they were written.

```
String stockName;      /* Name of the stock quote */
double stockValue;     /* Current value of the stock */
long   stockTime;      /* Time of the stock update */
double stockDiff;      /* +/- change in the stock quote */
String stockInfo;      /* Information on this stock */

stockName  = message.readString();
stockValue = message.readDouble();
stockTime  = message.readLong();
stockDiff  = message.readDouble();
stockInfo  = message.readString();
```

9.4.9 Creating an ObjectMessage

The stock information could be sent in the form of a special *StockObject* Java object. This object can then be sent as the body of a *ObjectMessage*. The *ObjectMessage* can be used to sent Java objects.

These values are set using methods that are unique to the *StockObject* implementation. For example, the StockObject may have methods that set the various data values. An application using StockObject might look like this:

```
String stockName;      /* Name of the stock quote */
double stockValue;     /* Current value of the stock */
long   stockTime;      /* Time of the stock update */
double stockDiff;      /* +/- change in the stock quote */
String stockInfo;      /* Information on this stock */

/* Create a StockObject */
StockObject stockObject = new StockObject();


/* Establish the values for the StockObject */

stockObject.setName(stockName);
stockObject.setValue(stockValue);
stockObject.setTime(stockTime);
stockObject.setDiff(stockDiff);
stockObject.setInfo(stockInfo);
```

To create an *ObjectMessage*, to pass the *StockObject* in the message, you would do the following:

```
/* Create an ObjectMessage */

ObjectMessage message;
message = session.createObjectMessage();

/* Set the body of the message to the StockObject */
message.setObject(stockObject);
```

9.4.10 Unpacking an ObjectMessage

To unpack an *ObjectMessage*, use the *ObjectMessage.getObject* method to get the object. Once the object is retrieved, the client program can use methods appropriate to that object type to retrieve data from the object.

```
StockObject stockObject;

/* Retrieve the StockObject from the message */
stockObject = (StockObject)message.getObject();

/* Extract data from the StockObject by using StockObject methods */

String stockName;      /* Name of the stock quote */
double stockValue;     /* Current value of the stock */
long   stockTime;      /* Time of the stock update */
double stockDiff;      /* +/- change in the stock quote */
String stockInfo;      /* Information on this stock */

stockName =  stockObject.getName();
stockValue = stockObject.getValue();
stockTime =  stockObject.getTime();
stockDiff =  stockObject.getDiff();
stockInfo =  stockObject.getInfo();
```