Java Message Service™

JMS is an API for accessing enterprise messaging systems from Java programs .

Version 1.0.1 October 5, 1998

Please send technical comments on this specification to:
jets-jms@eng.sun.com

Please send product and business questions to:
jets-jms-business@eng.sun.com

Mark Hapner, Senior Staff Engineer
Rich Burridge, Staff Engineer
Rahul Sharma, Staff Engineer
Sun Microsystems, JavaSoft ware

Sun microsystems
JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
408-343-1400

Contents

1. Introduction 12

1.1 Abstract 12
1.2 Overview 12
1.2.1 Is This a Mail API? 13
1.2.2 Existing Messaging Systems 13
1.2.3 JMS Objectives 14
1.2.3.1 JMS Provider 14
1.2.3.2 JMS Messages 14
1.2.3.3 JMS Domains 14
1.2.3.4 Portability 15
1.2.4 What JMS Does Not Include 15
1.3 What Is Required by JMS 16
1.4 Relationship to Other JavaSoft Enterprise APIs 16
1.4.1 JDBC. 16
1.4.2 JavaBeans 16
1.4.3 Enterprise Java Beans 16
1.4.4 Java Transaction (JTA) 17
1.4.5 Java Transaction Service (JTS). 17
1.4.6 Java Naming and Directory Service (JNDI) 17

2. Architecture. 18

2.1 Overview. 18
2.2 What is a JMS Application? 18
2.3 Administration 19

2.4 Two Messaging Styles 19
2.5 JMS Interfaces 19
2.6 Developing a JMS Application 21
2.6.1 Developing a JMS Client 21
2.7 Security 21
2.8 Multi-Threading 21
2.9 Triggering Clients 22
2.10 Request/Reply 23

3. JMS Message Model 24

3.1 Background 24
3.2 Goals 24
3.3 JMS Messages 25
3.4 Message Header Fields 25
3.4.1 JMSDestination 25
3.4.2 JMSDeliveryMode 26
3.4.3 JMSMessageID 26
3.4.4 JMSTimestamp 26
3.4.5 JMSCorrelationID 27
3.4.6 JMSReplyTo 28
3.4.7 JMSRedelivered 28
3.4.8 JMSType 28
3.4.9 JMSExpiration 29
3.4.10 JMSPriority 29
3.4.11 How Message Header Values Are Set 30
3.4.12 Overriding Message Header Fields 30
3.5 Message Properties 30
3.5.1 Property Names 31
3.5.2 Property Values 31
3.5.3 Using Properties 31
3.5.4 Property Value Conversion 31
3.5.5 Property Values as Objects 32
3.5.6 Property Iteration 32
3.5.7 Deleting a Message's Property Values 33
3.5.8 Non-existent Properties 33
3.5.9 JMS Defined Properties 33
3.5.10 Provider-Specific Properties 35
3.6 Message Acknowledgment 35

3.7 The Message Interface 35
3.8 Message Selection 35
3.8.1 Message Selector 36
3.8.1.1 Message Selector Syntax 36
3.8.1.2 Null Values 39
3.8.1.3 Special Notes 40
3.9 Access to Sent Messages 41
3.10 JMS Message Body 41
3.10.1 Conversions Provided by StreamMessage and MapMessage 42
3.10.2 Messages for Non-JMS Clients 43
3.11 Provider Implementations of JMS Message Interfaces 43

4. JMS Common Facilities 44

4.1 Overview 44
4.2 Administered Objects 44
4.2.1 Destination 45
4.2.2 ConnectionFactory 45
4.3 Connection 46
4.3.1 Authentication 46
4.3.2 Client Identifier 46
4.3.3 Connection Setup 47
4.3.4 Pausing Delivery of Incoming Messages. 47
4.3.5 Closing a Connection. 48
4.3.6 Sessions 48
4.3.7 ConnectionMetaData 48
4.3.8 ExceptionListener. 48
4.4 Session 49
4.4.1 Closing a Session 49
4.4.2 MessageProducer and MessageConsumer Creation . 49
4.4.3 Creating TemporaryDestinations 50
4.4.4 Creating Destinations 50
4.4.5 Optimized Message Implementations. 50
4.4.6 Conventions for Using a Session 51
4.4.7 Transactions 51
4.4.8 Distributed Transactions 52
4.4.9 Multiple Sessions 52
4.4.10 Message Order 53

4.4.10.1 Order of Message Receipt 53
4.4.10.2 Order of Message Sends 53
4.4.11 Message Acknowledgment 54
4.4.12 Duplicate Delivery of Messages 55
4.4.13 Duplicate Production of Messages 55
4.4.14 Serial Execution of Client Code 56
4.4.15 Concurrent Message Delivery 56
4.5 MessageConsumer 56
4.5.1 Synchronous Delivery 57
4.5.2 Asynchronous Delivery 57
4.6 MessageProducer 57
4.7 Message Delivery Mode 58
4.8 Message Time-To-Live 58
4.9 Exceptions 59
4.10 Reliability 59

5. JMS Point-to-Point Model 62

5.1 Overview 62
5.2 Queue Management 62
5.3 Queue 63
5.4 TemporaryQueue 63
5.5 QueueConnectionFactory 63
5.6 QueueConnection 63
5.7 QueueSession 63
5.8 QueueReceiver 64
5.9 QueueSender 64
5.10 QueueBrowser 64
5.11 QueueRequestor 65
5.12 Reliability 65

6. JMS Publish/Subscribe Model 66

6.1 Overview 66
6.2 Pub/Sub Latency 66
6.3 Durable Subscription 67
6.4 Topic Management 67
6.5 Topic 67
6.6 TemporaryTopic 68
6.7 TopicConnectionFactory 68

6.8 TopicConnection 68
6.9 TopicSession 68
6.10 TopicPublisher 69
6.11 TopicSubscriber 69
6.11.1 Durable TopicSubscriber 69
6.12 Recovery and Redelivery 70
6.13 Administering Subscriptions 70
6.14 TopicRequestor 70
6.15 Reliability 70

7. JMS Exceptions 72

7.1 Overview 72
7.2 The JMSException 72
7.3 Standard Exceptions 73

8. JMS Application Server Facilities 76

8.1 Overview 76
8.2 Concurrent Processing of a Subscription's Messages 76
8.2.1 Session 77
8.2.2 ServerSession 77
8.2.3 ServerSessionPool 77
8.2.4 ConnectionConsumer 78
8.2.5 How a ConnectionConsumer Uses a ServerSession . 78
8.2.6 How an App Server Implements a ServerSession . . . 79
8.2.7 The Result 79
8.3 XAConnectionFactory 82
8.4 XAConnection 82
8.5 XASession 82
8.6 JMS Application Server Interfaces 83

9. JMS Sample Code 84

9.1 Sample JMS Code 84
9.1.1 Background 84
9.1.2 Point-to-Point Setup 84
9.1.2.1 Getting a QueueConnectionFactory 85
9.1.2.2 Getting a Message Queue 85
9.1.2.3 Getting a QueueConnection 85
9.1.2.4 Getting a QueueSession 85

9.1.2.5 Getting a QueueSender 86
9.1.2.6 Getting a QueueReceiver 86
9.1.3 Publish/Subscribe Messaging Domain Setup 86
9.1.3.1 Getting a TopicConnectionFactory 86
9.1.3.2 Getting a Message Topic 87
9.1.3.3 Getting a TopicConnection 87
9.1.3.4 Getting a TopicSession 87
9.1.3.5 Getting a TopicSubscriber 87
9.1.3.6 Getting a TopicPublisher 88
9.1.4 JMS Message Types 89
9.1.4.1 Using a BytesMessage 89
9.1.4.2 Using a TextMessage 89
9.1.4.3 Using a MapMessage 89
9.1.4.4 Using a StreamMessage 90
9.1.4.5 Using an ObjectMessage 91
9.1.5 Point-to-Point Sending and Receiving............. 92
9.1.5.1 Sending a Message 92
9.1.5.2 Receiving a Message 92
9.1.6 Publish/Subscribe Sending and Receiving......... 92
9.1.6.1 Sending a Message 92
9.1.6.2 Receiving a Message 92
9.1.7 Unpacking messages 93
9.1.7.1 Unpacking a BytesMessage 93
9.1.7.2 Unpacking a TextMessage 93
9.1.7.3 Unpacking a MapMessage 93
9.1.7.4 Unpacking a StreamMessage 93
9.1.7.5 Unpacking an ObjectMessage 94
9.1.8 Message Selection.............................. 94
9.1.8.1 Point-To-Point QueueReceiver Setup 94
9.1.8.2 Publish/Subscribe TopicSubscriber Setup 95

10. Issues 96

10.1 Resolved Issues................................ 96
10.1.1 JDK 1.1.x Compatibility....................... 96
10.1.2 Distributed Java Event Model 96
10.1.3 Should the Two JMS Domains, PTP and Pub/Sub, be merged? 96
10.1.4 Should JMS Specify a Set of JMS Java Beans?....... 97

10.1.5 Alignment with the CORBA Notification Service . . . 97
10.1.6 Should JMS Provide End-to-end Synchronous Message Delivery and Notification of Delivery? 97
10.1.7 Should JMS Provide a Send-to-List Mechanism? 98
10.1.8 Should JMS Provide Subscription Notification? 98

11. Change History 100

11.1 Version 1.0.1 100
11.1.1 JMS Exceptions 100

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.

1.2.4 JMS Provider

As noted earlier, a JMS provider is the entity which 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.

1.2.5 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 which is independent of JMS provider.

1.2.6 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 client interfaces tailored for each domain.

1.2.7 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).

1.2.8 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.

1.4 Relationship to Other JavaSoft Enterprise APIs

1.4.1 JDBC

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

1.4.2 JavaBeans

JavaBeans 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 Java beans.

1.4.3 Enterprise Java Beans

JMS will be an important resource available to EJB component developers. It can be used in conjunction with other resources like JDBC to implement enterprise services.

The current EJB specification defines beans that are invoked synchronously via method calls from EJB clients. A future release of EJB will add a form of asynchronous bean that is invoked when a JMS client sends it a message.

1.4.4 Java Transaction (JTA)

The java 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 per se.

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. This 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 this explicitly.

1.4.6 Java Naming and Directory Service (JNDI)

JMS clients look up configured JMS objects using JNDI. 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.

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 systems 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 each JMS provider 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.

2.4 Two Messaging Styles

A JMS application either uses the point-to-point (PTP) or publish-and-subscribe (Pub/Sub) style of messaging. Nothing prevents these styles from being combined in a single application; however, JMS focuses on applications that use one or the other.

JMS defines these two styles because they represent the two dominant approaches to messaging currently in use. Since many messaging systems only support one of these styles, JMS provides a separate domain for each and defines compliance for each domain.

2.5 JMS Interfaces

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

Table 0-1 Relationship of PTP and Pub/Sub interfaces

JMS Parent	**PTP Domain**	**Pub/Sub Domain**
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

The following provides a brief definition of these JMS concepts. See the PTP and Pub/Sub chapters for more information.

- ConnectionFactory - an administered object used by a client to create a Connection
- Connection - An active connection to a JMS provider
- Destination - 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

The term *consume* is used in this document to mean the receipt of a message by a JMS client, i.e. 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
- Use the Connection to create one or more JMS Sessions
- Use a Session and the Destinations to create the MessageProducers and MessageConsumers needed

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 Multi-Threading

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 multi-threaded client. The remainder are designed to be accessed by one logical thread of control at a time.

Table 0-2 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 multi-threaded. 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.

There are many styles of request/reply supported by enterprise messaging products. From 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 Message Model 3

3.1 Background

Enterprise messaging products treat messages as lightweight entities that consist of a header and a body. The header contains fields used for message routing and identification; the body contains the application data being sent.

Within this general form, the definition of a message varies significantly across products. There are major differences in the content and semantics of headers. Some products use a self describing, canonical encoding of message data; others treat data as completely opaque. Some products provide a repository for storing message descriptions that can be used to identify and interpret message content; others don't.

It would be quite difficult for JMS to capture the breadth of this, sometimes conflicting, union of message models.

3.2 Goals

The JMS message model has the following goals:

- Provide a single, unified message API
- Provide an API suitable for creating messages that match the format used by existing, non-JMS applications
- Support the development of heterogeneous applications that span operating systems, machine architectures, and computer languages
- Support messages containing Java objects

- Support messages containing Extensible Markup Language pages (see http://www.w3.org/XML).

3.3 JMS Messages

JMS messages are composed of the following parts:

- Header - All messages support the same set of header fields. Header fields contain values used by both clients and providers to identify and route messages.
- Properties - In addition to the standard header fields, messages provide a built-in facility for adding optional header fields to a message.
  - Application-specific properties - In effect, this provides a mechanism for adding application specific header fields to a message.
  - Standard properties - JMS defines some standard properties that are, in effect, optional header fields.
  - Provider-specific properties - Integrating a JMS client with a JMS provider native client may require the use of provider-specific properties. JMS defines a naming convention for these.
- Body - JMS defines several types of message body which cover the majority of messaging styles currently in use.

3.4 Message Header Fields

The following subsections describe each JMS message header field. A message's complete header is transmitted to all JMS clients that receive the message. JMS does not define the header fields transmitted to non-JMS clients.

3.4.1 JMSDestination

The *JMSDestination* header field contains the destination to which the message is being sent.

When a message is sent this value is ignored. After completion of the send it holds the destination object specified by the sending method.

When a message is received, its destination value must be equivalent to the value assigned when it was sent.

3.4.2 *JMSDeliveryMode*

The *JMSDeliveryMode* header field contains the delivery mode specified when the message was sent.

When a message is sent this value is ignored. After completion of the send it holds the delivery mode specified by the sending method.

See Section 4.7, "Message Delivery Mode" for more information.

3.4.3 *JMSMessageID*

The *JMSMessageID* header field contains a value that uniquely identifies each message sent by a provider.

When a message is sent, *JMSMessageID* is ignored. When the send method returns it contains a provider-assigned value.

A *JMSMessageID* is a *String* value which should function as a unique key for identifying messages in a historical repository. The exact scope of uniqueness is provider defined. It should at least cover all messages for a specific installation of a provider where an installation is some connected set of message routers.

All *JMSMessageID* values must start with the prefix 'ID:'. Uniqueness of message ID values across different providers is not required.

Since message ID's take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that message ID is not used by an application. JMS *MessageProducer* provides a hint to disable message ID. When a client sets a producer to disable message ID it is saying that it does not depend on the value of message ID for the messages it produces. These messages must either have message ID set to null or, if the hint is ignored, message ID must be set to its normal unique value.

3.4.4 *JMSTimestamp*

The *JMSTimestamp* header field contains the time a message was handed off to a provider to be sent. It is not the time the message was actually transmitted because the actual send may occur later due to transactions or other client side queueing of messages.

When a message is sent, *JMSTimestamp* is ignored. When the send method returns it contains a a time value somewhere in the interval between the call and the return. It is in the format of a normal Java millis time value.

Since timestamps take some effort to create and increase a message's size, some JMS providers may be able to optimize message overhead if they are given a hint that timestamp is not used by an application. JMS *MessageProducer* provides a hint to disable timestamps. When a client sets a producer to disable timestamps it is saying that it does not depend on the value of timestamp for the messages it produces. These messages must either have timestamp set to null or, if the hint is ignored, timestamp must be set to its normal value.

3.4.5 JMSCorrelationID

A client can use the *JMSCorrelationID* header field to link one message with another. A typically use is to link a response message with its request message.

JMSCorrelationID can hold one of the following:

- A provider-specific message ID
- An application-specific *String*
- A provider-native *byte[]* value.

Since each message sent by a JMS provider is assigned a message ID value it is convenient to link messages via message ID. All message ID values must start with the 'ID:' prefix.

In some cases, an application (made up of several clients) needs to use an application specific value for linking messages. For instance, an application may use *JMSCorrelationID* to hold a value referencing some external information. Application specified values must not start with the 'ID:' prefix; this is reserved for provider-generated message ID values.

If a provider supports the native concept of correlation ID, a JMS client may need to assign specific *JMSCorrelationID* values to match those expected by non-JMS clients. A *byte[]* value is used for this purpose. JMS providers without native correlation ID values are not required to support *byte[]* values*. The use of a *byte[]* value for *JMSCorrelationID* is non-portable.

---

* Their implementation of *setJMSCorrelationIDAsBytes()* and *getJMSCorrelationIDAsBytes()* may throw *java.lang.UnsupportedOperationException*.

3.4.6 JMSReplyTo

The *JMSReplyTo* header field contains a Destination supplied by a client when a message is sent. It is the destination where a reply to the message should be sent.

Messages sent with a null *JMSReplyTo* value may be a notification of some event or they may just be some data the sender thinks is of interest.

Messages sent with a *JMSReplyTo* value are typically expecting a response. A response may be optional, it is up to the client to decide.

3.4.7 JMSRedelivered

If a client receives a message with the *JMSRedelivered* indicator set, it is likely, but not guaranteed, that this message was delivered to the client earlier but the client did not acknowledge its receipt at that earlier time. See Section 4.4.13, "Message Acknowledgment" for more information.

This header field has no meaning on send and is left unassigned by the sending method.

3.4.8 JMSType

The *JMSType* header field contains a message type identifier supplied by a client when a message is sent.

Some JMS providers use a message repository that contains the definition of messages sent by applications. The *type* header field may reference a message's definition in the provider's repository.

JMS does not define a standard message definition repository nor does it define a naming policy for the definitions it contains.

Some messaging systems require that a message type definition for each application message be created and that each message specify its type. In order to work with such JMS providers, JMS clients should assign a value to *JMSType* whether the application makes use of it or not. This insures that it is properly set for those providers that require it.

To insure portability, JMS clients should use symbolic values for *JMSType* that can be configured at installation time to the values defined in the current providers message repository. If string literals are use they may not be valid type names for some JMS providers.

3.4.9 JMSExpiration

When a message is sent, its expiration time is calculated as the sum of the time-to-live value specified on the send method and the current GMT. On return from the send method, the message's *JMSExpiration* header field contains this value. When a message is received its *JMSExpiration* header field contains this same value.

If the time-to-live is specified as zero, expiration is set to zero which indicates the message does not expire.

When GMT is later than an undelivered message's expiration time, the message should be destroyed. JMS does not define a notification of message expiration.

Clients should not receive messages that have expired; however, JMS does not guarantee that this will not happen.

3.4.10 JMSPriority

The *JMSPriority* header field contains the message's priority.

When a message is sent this value is ignored. After completion of the send it holds the value specified by the method sending the message.

JMS defines a ten level priority value with 0 as the lowest priority and 9 as the highest. In addition, clients should consider priorities 0-4 as gradations of *normal* priority and priorities 5-9 as gradations of *expedited* priority.

JMS does not require that a provider strictly implement priority ordering of messages; however, it should do its best to deliver expedited messages ahead of normal messages.

3.4.11 How Message Header Values Are Set

Table 0-3 Message Header Field Value Sent

Header Fields	**Set By**
JMSDestination	Send Method
JMSDeliveryMode	Send Method
JMSExpiration	Send Method
JMSPriority	Send Method
JMSMessageID	Send Method
JMSTimestamp	Send Method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

3.4.12 Overriding Message Header Fields

JMS permits an administrator to configure JMS to override the client specified values for *JMSDeliveryMode*, *JMSExpiration* and *JMSPriority*. If this is done, the header field value must reflect the administratively specified value.

JMS does not define specifically how an administrator overrides these header field values. A JMS provider is not required to support this administrative option.

3.5 Message Properties

In addition to the header fields defined here, *Message* contains a built-in facility for supporting property values. In effect, this provides a mechanism for adding optional header fields to a message.

Properties allow a client, via message selectors (see Section 3.8, ”Message Selection”), to have a JMS provider select messages on its behalf using application-specific criteria.

3.5.1 Property Names

Property names must obey the rules for a message selector identifier. See Section 3.8.1.1, "Message Selector Syntax" for more information.

3.5.2 Property Values

Property values can be *boolean*, *byte*, *short*, *int*, *long*, *float*, *double*, and *String*.

3.5.3 Using Properties

Property values are set prior to sending a message. When a client receives a message, its properties are in read-only mode. If a client attempts to set properties at this point, a *MessageNotWriteableException* is thrown. If *clearProperties()* is called, the properties can now be both read from and written to.

A property value may duplicate a value in a message's body or it may not. Although JMS does not define a policy for what should or should not be made a property, application developers should note that JMS providers will likely handle data in a message's body more efficiently than data in a message's properties. For best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection.

See Section 3.8, "Message Selection" for more information about JMS message properties.

3.5.4 Property Value Conversion

Properties support the following conversion table. The marked cases must be supported. The unmarked cases must throw the JMS *MessageFormatException*. The *String* to numeric conversions must throw the *java.lang.NumberFormatException* if the numeric's *valueOf()* method does not accept the *String* value as a valid representation.

Table 0-4 A value set as the row type can be read as the column type

	boolean	byte	short	int	long	float	double	String
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
String	X	X	X	X	X	X	X	X

3.5.5 *Property Values as Objects*

In addition to the type-specific set/get methods for properties, JMS provides the *setObjectProperty/getObjectProperty* methods. These support the same set of property types using the objectified primitive values. Their purpose is to allow the decision of property type to made at execution time rather than at compile time. They support the same property value conversions.

The *setObjectProperty* method accepts values of *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double* and *String*. An attempt to use any other class must throw a JMS *MessageFormatException*.

The *getObjectProperty* method only returns values of *null*, *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double* and *String*. A *null* value is returned if a property by the specified name does not exist.

3.5.6 *Property Iteration*

The order of property values is not defined. To iterate through a message's property values, use *getPropertyNames* to retrieve a property name enumeration and then use the various property get methods to retrieve their values.

3.5.7 Deleting a Message's Property Values

A message's properties are deleted by the *clearProperties* method. This leaves the message with an empty set of properties.

3.5.8 Non-existent Properties

Getting a property value for a name which has not been set returns a *null* value. Only the *getStringProperty* and *getObjectProperty* methods can return a *null* value. The other property get methods must throw a *java.lang.NullPointerException* if they are used to get a non-existent property.

3.5.9 JMS Defined Properties

JMS reserves the 'JMSX' property name prefix for JMS defined properties. The full set of these properties is provided below. New JMS defined properties may be added in later versions of JMS.

Unless noted otherwise, support for these properties is optional. The *String[] ConnectionMetaData.getJMSXPropertyNames()* method returns the names of the JMSX properties supported by a connection.

JMSX properties may be referenced in message selectors whether or not they are supported by a connection. They are treated like any other absent property.

Table 0-5

Name	**Type**	**Set By**	**Use**
JMSXUserID	String	Provider on Send	The identity of the user sending the message
JMSXAppID	String	Provider on Send	The identity of the application sending the message
JMSXDeliveryCount	int	Provider on Receive	The number of message delivery attempts; the first is 1, the second 2,...
JMSXGroupID	String	Client	The identity of the message group this message is part of

Table 0-5

Name	Type	Set By	Use
JMSXGroupSeq	int	Client	The sequence number of this message within the group; the first message is 1, the second 2,...
JMSXProducerTXID	String	Provider on Send	The transaction identifier of the transaction within which this message was produced
JMSXConsumerTXID	String	Provider on Receive	The transaction identifier of the transaction within which this message was consumed
JSMXRcvTimestamp	long	Provider on Receive	The time JMS delivered the message to the consumer
JSMXState	int	Provider	Assume there exists a message warehouse that contains a separate copy of each message sent to each consumer and that these copies exist from the time the original message was sent. Each copy's state is one of: 1(wating), 2(ready), 3(expired) or 4(retained) Since state is of no interest to producers and consumers it is not provided to either. It is only of relevance to messages looked up in a warehouse and JMS provides no API for this.

JSMX properties 'set by provider on send' are available to both the producer and the consumers of the message. JSMX properties 'set by provider on receive' are only available to the consumers.

JMSXGroupID and *JMSXGroupSeq* are standard properties clients should use if they want to group messages. All providers must support them.

Unless specifically noted, the values and semantics of the JMSX properties are undefined.

3.5.10 Provider-Specific Properties

JMS reserves the 'JMS_<vendor_name>' property name prefix for provider-specific properties. Each provider defines their own value of <vendor_name>. This is the mechanism a JMS provider uses to make its special per message services available to a JMS client.

The purpose of provider-specific properties is to provide special features needed to support JMS use with provider-native clients. They should not be used for JMS to JMS messaging.

3.6 Message Acknowledgment

All JMS messages support the *acknowledge* method for use when a client has specified that a JMS consumer's messages are to be explicitly acknowledged.

If a client uses automatic acknowledgment, calls to acknowledge are ignored.

See Section 4.4.13, "Message Acknowledgment" for more information.

3.7 The Message Interface

The *Message* interface is the root interface for all JMS messages. It defines the JMS message header fields, property facility and the acknowledge method used for all messages.

3.8 Message Selection

Many messaging applications need to filter and categorize the messages they produce.

In the case where a message is sent to a single receiver, this can be done with reasonable efficiency by putting the criteria in the message and having the receiving client discard the ones it's not interested in.

When a message is broadcast to many clients, it becomes useful to place the criteria into the message header so that it is visible to the JMS provider. This allows the provider to handle much of the filtering and routing work that would otherwise need to be done by the application.

JMS provides a facility that allows clients to delegate message selection to their JMS provider. This simplifies the work of the client and allows JMS providers to eliminate the time and bandwidth they would otherwise waste sending messages to clients that don't need them.

Clients attach application-specific selection criteria to messages using message properties. Clients specify message selection criteria using JMS *message selector* expressions.

3.8.1 Message Selector

A JMS message selector allows a client to specify, by message header, the messages it's interested in. Only messages whose headers match the selector are delivered. The semantics of *not delivered* differ a bit depending on the MessageConsumer being used. See Section 5.8, "QueueReceiver" and Section 6.11, "TopicSubscriber" for more details.

Message selectors cannot reference message body values.

A message selector matches a message when the selector evaluates to true when the message's header field and property values are substituted for their corresponding identifiers in the selector.

3.8.1.1 Message Selector Syntax

A message selector is a *String* whose syntax is based on a subset of the SQL92* conditional expression syntax.

The order of evaluation of a message selector is from left to right within precedence level. Parentheses can be used to change this order.

Predefined selector literals and operator names are written here in upper case; however, they are case insensitive.

A selector can contain:

- Literals:
  - A string literal is enclosed in single quotes with an included single quote represented by doubled single quote such as 'literal' and 'literal''s'; like Java *String* literals these use the unicode character encoding.

---

* See X/Open CAE Specification Data Management: Structured Query Language (SQL), Version 2, ISBN: 1-85912-151-9 March 1996.

- An exact numeric literal is a numeric value without a decimal point such as 57, -957, +62; numbers in the range of Java *long* are supported.
- An approximate numeric literal is a numeric value in scientific notation such as 7E3, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +6.2; numbers in the range of Java *double* are supported.
- The boolean literals *TRUE* and *FALSE*.

• Identifiers:

- An identifier is an unlimited length character sequence that must begin with a Java identifier start character and all following characters must be Java identifier part characters. An identifier start character is any character for which the method *Character.isJavaIdentifierStart* returns true. This includes '_' and '$'. An identifer part character is any character for which the method *Character.isJavaIdentifierPart* returns true.
- Identifiers cannot be the names *NULL*, *TRUE*, or *FALSE*.
- Identifiers cannot be *NOT*, *AND*, *OR*, *BETWEEN*, *LIKE*, *IN*, and *IS*.
- Identifiers are either header field references or property references.
- Identifiers are case sensitive.
- Message header field references are restricted to *JMSDeliveryMode*, *JMSPriority*, *JMSMessageID*, *JMSTimestamp*, *JMSCorrelationID*, and *JMSType*. *JMSMessageID*, *JMSTimestamp*, *JMSCorrelationID*, and *JMSType* values may be *null* and if so are treated as a NULL value.
- Any name beginning with 'JMSX' is a JMS defined property name.
- Any name beginning with 'JMS_' is a provider-specific property name.
- Any name that does not begin with 'JMS' is an application-specific property name. If a property is referenced that does not exist in a message its value is NULL. If it does exist, its value is the corresponding property value.

• Whitespace is the same as that defined for Java: space, horizontal tab, form feed and line terminator.

• Expressions:

- A selector is a conditional expression; a selector that evaluates to true matches; a selector that evaluates to false or unknown does not match.
- Arithmetic expressions are composed of themselves, arithmetic operations, identifiers with numeric values and numeric literals.

  - Conditional expressions are composed of themselves, comparison operations, logical operations, identifiers with boolean values and boolean literals.
- Standard bracketing () for ordering expression evaluation is supported.
- Logical operators in precedence order: NOT, AND, OR
- Comparison operators: =, >, >=, <, <=, <> (not equal)
  - Only *like* type values can be compared. An exact numeric value can only be compared with an exact numeric value. The same type correspondence restriction applies to comparison of approximate numeric, String and boolean values. If the comparison of non-like type values is attempted, the selector is always false.
  - *String* comparison is restricted to = and <>. Two strings are equal if and only if they contain the same sequence of characters.
- Arithmetic operators in precedence order:
  - +, - unary
  - *, / multiplication and division
  - +, - addition and subtraction
  - Arithmetic operations must use Java numeric promotion.
- *arithmetic-expr1* [NOT] BETWEEN *arithmetic-expr2* and *arithmetic-expr3* comparison operator
  - age BETWEEN 15 and 19 is equivalent to age >= 15 AND age <= 19
  - age NOT BETWEEN 15 and 19 is equivalent to age < 15 OR age > 19
- *identifier* [NOT] IN (*string-literal1*, *string-literal2*,...) comparison operator where *identifier* has a *String* or NULL value.
  - Country IN (’ UK’, ’US’, ’France’) is true for ‘UK’ and false for ‘Peru’ it is equivalent to the expression (Country = ’ UK’) OR (Country = ’ US’) OR (Country = ’ France’)
  - Country NOT IN (’ UK’, ’US’, ’France’) is false for ‘UK’ and true for ‘Peru’ it is equivalent to the expression NOT ((Country = ’ UK’) OR (Country = ’ US’) OR (Country = ’ France’))
  - If *identifier* of an IN or NOT IN operation is NULL the value of the operation is unknown.

- *identifier* [NOT] LIKE *pattern-value* [ESCAPE *escape-character*] comparison operator, where *identifier* has a *String* value; *pattern-value* is a string literal where '_' stands for any single character; '%' stands for any sequence of characters (including the empty sequence); and all other characters stand for themselves. The optional *escape-character* is a single character string literal whose character is used to escape the special meaning of the '_' and '%' in *pattern-value.*
  - *phone LIKE '12%3'* is true for '123' '12993' and false for '1234'
  - *word LIKE 'l_se'* is true for 'lose' and false for 'loose'
  - *underscored LIKE '_%' ESCAPE '\'* is true for '_foo' and false for 'bar'
  - *phone NOT LIKE '12%3'* is false for '123' and '12993' and true for '1234'
  - If *identifier* of a LIKE or NOT LIKE operation is NULL the value of the operation is unknown.
- *identifier* IS NULL comparison operator tests for a null header field value, or a missing property value.
  - *prop_name IS NULL*
- *identifier* IS NOT NULL comparison operator tests for the existence of a non null header field value or property value.
  - *prop_name IS NOT NULL*

JMS providers are required to verify the syntactic correctness of a message selector at the time it is presented. A method providing a syntactically incorrect selector must result in a JMS *InvalidSelectorException*.

The following message selector selects messages with a message type of *car* and color of *blue* and weight greater than 2500 lbs:

```
"JMSType = 'car' AND color = 'blue' AND weight > 2500"
```

3.8.1.2 Null Values

As noted above, header fields and property values may be NULL. The evaluation of selector expressions containing NULL values is defined by SQL 92 NULL semantics. A brief description of these semantics is provided here.

SQL treats a NULL value as unknown. Comparison or arithmetic with an unknown value always yields an unknown value.

The IS NULL and IS NOT NULL operators convert an unknown header or property value into the respective TRUE and FALSE values.

The boolean operators use three valued logic as defined by the following tables:

Table 0-6 The definition of the AND operator

AND	**T**	**F**	**U**
T	T	F	U
F	F	F	F
U	U	F	U

:

Table 0-7 The definition of the OR operator

OR	**T**	**F**	**U**
T	T	T	T
F	T	F	U
U	T	U	U

:

Table 0-8 The definition of the NOT operator

NOT	
T	F
F	T
U	U

3.8.1.3 *Special Notes*

When used in a message selector *JMSDeliveryMode* is treated as having the values 'PERSISTENT' and 'NON_PERSISTENT'.

Date and time values should use the standard Java long millis value. When including a date or time literal in a message selector, it should be an integer literal for a millis value. The standard way to produce millis values is to use *java.util.Calendar.*

Although SQL supports fixed decimal comparison and arithmetic, JMS message selectors do not. This is the reason for restricting exact numeric literals to those without a decimal (and the addition of numerics with a decimal as an alternate representation for an approximate numeric values).

SQL comments are not supported.

3.9 Access to Sent Messages

After sending a message, a client may retain and modify it without affecting the message that has been sent. The same message object may be sent multiple times.

During the execution of its sending method, the message must not be changed by the client. If it is modified, the result of the send is undefined.

3.10 JMS Message Body

JMS provides five forms of message body. Each form is defined by a message interface:

- StreamMessage - a message whose body contains a stream of Java primitive values. It is filled and read sequentially.
- MapMessage - a message whose body contains a set of name-value pairs where names are *Strings* and values are Java primitive types. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
- TextMessage - a message whose body contains a *java.lang.String*. The inclusion of this message type is based on our presumption that *String* messages will be used extensively. One reason for this is that XML will likely become a popular mechanism for representing the content of JMS messages.
- ObjectMessage - a message that contains a Serializable Java object. If a collection of Java objects is needed, one of the collection classes provided in JDK 1.2 can be used.
- BytesMessage - a message that contains a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing

message format. In many cases, it will be possible to use one of the other, self-defining, message types instead. *Although JMS allows the use of message properties with byte messages it is typically not done since the inclusion of properties may affect the format.*

3.10.1 Conversions Provided by StreamMessage and MapMessage

Both StreamMessage and MapMessage support the same set of primitive data types.

The types can be read or written explicitly using methods for each type. They may also be read or written generically as objects. For instance, a call to *MapMessage.setInt("foo", 6)* is equivalent to *MapMessage.setObject("foo", new Integer(6))*. Both forms are provided because the explicit form is convenient for static programming and the object form is needed when types are not known at compile time.

Both StreamMessage and MapMessage support the following conversion table. The marked cases must be supported. The unmarked cases must throw a JMS *MessageFormatException*. The *String* to numeric conversions must throw a *java.lang.NumberFormatException*if the numeric's *valueOf()* method does not accept the *String* value as a valid representation.

Table 0-9 A value written as the row type can be read as the column type

	boolean	**byte**	**short**	**char**	**int**	**long**	**float**	**double**	**String**	**byte[]**
boolean	X								X	
byte		X	X		X	X			X	
short			X		X	X			X	
char				X					X	
int					X	X			X	
long						X			X	
float							X	X	X	
double								X	X	
String	X	X	X	X	X	X	X	X	X	
byte[]										X

3.10.2 Messages for Non-JMS Clients

A number of enterprise messaging systems support some form of self-defining stream and/or map native message type. Although clients could use BytesMessages to construct native messages of this form, JMS provides the *StreamMessage* and *MapMessage* types as a more convenient API.

For instance, when a client is using a JMS provider that supports a native map message; and, it wishes to send a map message that can be read by both JMS and native clients, it uses a *MapMessage*. When the message is sent, the provider translates it into its native form. Native clients can then receive it. If a JMS provider receives it, the provider translates it back into a *MapMessage*.

Even when a new JMS application with newly defined messages is written, the application may choose to use *StreamMessage* and *MapMessage* to insure that later, non-JMS clients will be able to read them.

If a JMS client sends a *StreamMessage* or *MapMessage*, it must be translated by a receiving JMS provider into an equivalent *StreamMessage* or *MapMessage*. When passed between JMS clients, a message must always retain its full form. For instance, a message sent as *MapMessage* must not arrive at a JMS client as a *BytesMessage*.

If a JMS provider receives a message created by a native client, the provider should do its best to transform it into the 'best' JMS message type. For instance, if it is a native stream message it should be transformed into a *StreamMessage*. If this is not possible, the provider is always able to transform it into a *BytesMessage*.

3.11 Provider Implementations of JMS Message Interfaces

JMS provides a set of message interfaces that define the JMS message model. It does not provide implementations of these interfaces.

Each JMS provider provides its own implementation of its Session's message creation methods. This allows a provider to use message implementations that are tailored to its needs.

A provider must be prepared to accept, from a client, a message whose implementation is *not* one of its own. A message with a 'foreign' implementation may not be handled as efficiently as a provider's own implementation; however, they must be handled.

JMS Common Facilities 4

4.1 Overview

This chapter describes the JMS facilities that are shared by both the PTP and Pub/Sub domains.

4.2 Administered Objects

JMS administered objects are objects containing JMS configuration information that are created by a JMS administrator and later used by JMS clients. They make it practical to administer JMS applications in the enterprise.

Although the interfaces for administered objects do not explicitly depend on JNDI, JMS establishes the convention that JMS clients find them by looking them up in a namespace using JNDI.

An administrator can place an administered object anywhere in a namespace. JMS does not define a naming policy.

This strategy of partitioning JMS and administration provides several benefits:

- It hides provider-specific configuration details from JMS clients.
- It abstracts JMS administrative information into Java objects that are easily organized and administered from a common management console.
- Since there will be JNDI providers for all popular naming services, this means JMS providers can deliver one implementation of administered objects that will run everywhere.

An administered object should not hold on to any remote resources. Its lookup should not use remote resources other than those used by JNDI itself.

Clients should think of administered objects as local Java objects. Looking them up should not have any hidden side affects or use surprising amounts of local resources.

JMS defines two administered objects, *Destination* and *ConnectionFactory*.

It is expected that JMS providers will provide the tools an administrator needs to create and configure administered objects in a JNDI namespace. JMS provider implementations of administered objects should be both *javax.naming.Referenceable* and *java.io.Serializable* so that they can be stored in all JNDI naming contexts. In addition, it is recommended that these implementations follow the JavaBeans™ design patterns.

4.2.1 Destination

JMS does not define a standard address syntax. Although this was considered, it was decided that the differences in address semantics between existing enterprise messaging products was too wide to bridge with a single syntax. Instead, JMS defines the *Destination* object which encapsulates provider-specific addresses.

Since *Destination* is an administered object it may also contain provider-specific configuration information in addition to its address.

JMS also supports a client's use of provider-specific address names. See Section 4.4.4, "Creating Destinations" for more information.

Destination objects support concurrent use.

4.2.2 ConnectionFactory

A *ConnectionFactory* encapsulates a set of connection configuration parameters that has been defined by an administrator. A client uses it to create a Connection with a JMS provider.

ConnectionFactory objects support concurrent use.

4.3 Connection

A JMS *Connection* is a client's active connection to its JMS provider. It will typically allocate provider resources outside the Java virtual machine.

Connections support concurrent use.

A *Connection* serves several purposes:

- It encapsulates an open connection with a JMS provider. It typically represents an open TCP/IP socket between a client and a provider's service daemon.
- Its creation is where client authenticating takes place.
- It can specify a unique client identifier.
- It creates *Sessions*.
- It provides *ConnectionMetaData*.
- It supports an optional *ExceptionListener*.

Due to the authentication and communication setup done when a *Connection* is created, a *Connection* is a relatively heavy-weight JMS object. Most clients will do all their messaging with a single Connection. Other more advanced applications may use several *Connections*. JMS does not architect a reason for using multiple connections (other than when a client acts as a gateway between two different providers); however, there may be operational reasons for doing so.

4.3.1 Authentication

When creating a connection, a client may specify their credentials as name/password.

If no credentials are specified, the current thread's credentials are used. At this point, the JDK does not define the concept of a thread's default credentials; however, it is likely this will be defined in the near future. For now, the identity of the user under which the JMS client is running should be used.

4.3.2 Client Identifier

The preferred way to assign a client's client identifier is for it to be configured in a client-specific *ConnectionFactory* and transparently assigned to the connection it creates. Alternatively, a client can set a connections's client identifier using a provider-specific value.

The purpose of client identifier is to associate a connection and its objects with a state maintained on behalf of the client by a provider. The only such state identified by JMS is that required to support durable subscriptions, see Section 6.3, "Durable Subscription" for more information.

4.3.3 Connection Setup

A JMS client typically creates a *Connection*; one or more *Sessions*; and a number of *MessageProducers* and *MessageConsumers*. When a *Connection* is created it is in *stopped* mode. That means that no messages are being delivered to it.

It is typical to leave the *Connection* in stopped mode until setup is complete. At that point the *Connection's start()* method is called and messages begin arriving at the *Connection's* consumers. This setup convention minimizes any client confusion that may result from asynchronous message delivery while the client is still in the process of setting itself up.

A *Connection* can immediately be started and the setup can be done afterwards. Clients that do this must be prepared to handle asynchronous message delivery while they are still in the process of setting up.

A *MessageProducer* can send messages while a *Connection* is stopped.

4.3.4 Pausing Delivery of Incoming Messages

A connection's delivery of incoming messages can be temporarily stopped using its *stop()* method. It can be restarted using its *start()* method. When stopped, delivery to all the connection's *MessageConsumers* is inhibited: synchronous receive's block and messages are not delivered to *MessageListeners*.

After *stop* is called there may still be some messages delivered.

Stopping a connection has no affect on its ability to send messages. Stopping a stopped connection and starting a started connection is ignored.

4.3.5 Closing a Connection

Since a provider typically allocates significant resources outside the JVM on behalf of a connection, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough.

4.3.6 Sessions

A *Connection* is a factory for *Sessions* that use its underlying connection to a JMS provider for producing and consuming messages.

4.3.7 ConnectionMetaData

A *Connection* provides a *ConnectionMetaData* object. This object provides the latest version of JMS supported by the provider as well as the providers product name and version.

It also provides a list of the JMS defined property names supported by the connection.

4.3.8 ExceptionListener

If a JMS provider detects a problem with a connection it will inform the connection's *ExceptionListener* if one has been registered. It does this by calling the listener's *onException()* method passing it a *JMSException* describing the problem.

This allows a client to be asynchronously notified of a problem. Some connections only consume messages so they would have no other way to learn their connection has failed.

A Connection serializes execution of its *ExceptionListener*.

A JMS provider should attempt to resolve connection problems itself prior to notifying the client of them.

4.4 Session

A JMS *Session* is a single threaded context* for producing and consuming messages. Although it may allocate provider resources outside the Java virtual machine, it is considered a light-weight JMS object.

A *Session* serves several purposes:

- It is a factory for its *MessageProducers* and *MessageConsumers*.
- It is a factory for temporary destinations.
- It provides a way to create *Destination* objects for those clients that need to dynamically manipulate provider-specific destination names.
- It supplies provider-optimized message factories.
- It supports a single series of transactions that combine work spanning this session's producers and consumers into atomic units.
- A *Session* defines a serial order for the messages it consumes and the messages it produces.
- A *Session* retains messages it consumes until they have been acknowledged.
- A *Session* serializes execution of *MessageListeners* registered with it.

4.4.1 Closing a Session

Since a provider may allocate some resources on behalf of a session outside the JVM, clients should close them when they are not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough. The same is true for the *MessageProducers* and *MessageConsumers* created by a session.

4.4.2 MessageProducer and MessageConsumer Creation

A session can create and service multiple *MessageProducers* and *MessageConsumers*. See Section 4.5, "MessageConsumer" and Section 4.6, ~~"MessageProducer" for informa~~tion on their creation and use.

* There are no restrictions on the number of threads that can use a Session object or those it creates. The restriction is that the resources of a Session should not be used concurrently by multiple threads. It is up to the user to insure that this concurrency restriction is met. The simplest way to do this is to use one thread. In the case of asynchronous delivery, use one thread for setup in stopped mode and then start asynchronous delivery. In more complex cases the user must provide explicit synchronization.

Although a session may create multiple producers and consumers, they are restricted to serial use. In effect, only a single logical thread of control can use them. This is explained in more detail later.

4.4.3 Creating TemporaryDestinations

Although sessions are used to create temporary destinations, this is only for convenience. Their scope is actually the entire connection. Their lifetime is that of their connection and any of the connection's sessions are allowed to create a *MessageConsumer* for them.

TemporaryDestinations are destinations that are system generated uniquely for their connection. Only their own connection is allowed to create *MessageConsumers* for them.

One typical use for a *TemporaryDestination* is as the *JMSReplyTo* destination for service requests.

Each *TemporaryDestination* object is unique. It cannot be copied.

Since *TemporaryDestinations* may allocate resources outside the JVM they should be deleted if they are no longer needed. They will be automatically deleted when they are garbage collected or when their connection is closed.

4.4.4 Creating Destinations

Most clients will use *Destinations* that are JMS administered objects that they have looked up in via JNDI. This is the most portable approach.

Some specialized clients may need to create *Destinations* by dynamically manufacturing one using a provider specific destination name. Sessions provide a JMS domain-specific method for doing this.

4.4.5 Optimized Message Implementations

A session provides message create methods that use provider-optimized implementations. This allows a provider to minimize its overhead for handling messages.

Sessions are capable of sending all JMS messages regardless of how they may be implemented.

4.4.6 Conventions for Using a Session

Sessions are designed for serial use by one thread at a time.

One typical use is to have a thread block on a synchronous *MessageConsumer* until a message arrives. The thread may then use one or more of the session's *MessageProducers*.

Another typical use is to have one thread set up a session by creating its producers and one or more asynchronous consumers. Since the session serializes execution of its consumer's *MessageListeners*, they can safely share the resources of their session.

If a connection is left in stopped mode while its sessions are being set up, a client does not have to deal with messages arriving before the client is fully prepared to handle them. This is the preferred strategy because it eliminates the possibility of unanticipated conflicts between setup and message processing. It is possible to create and set up a session while a connection is receiving messages. In this case, more care is required to insure that a session's *MessageProducers*, *MessageConsumers* and *MessageListeners* are created in the right order. For instance, a bad order may cause a *MessageListener* to use a *MessageProducer* that has yet to be created; or, messages may arrive in the wrong order due to the order *MessageListeners* are registered.

If a client desires to have one thread producing messages while others consume them, the client should use a separate session for its producing thread.

It should be natural for most clients to partition their work into sessions. This model allows clients to start simply and incrementally add message processing complexity as their need for concurrency grows.

4.4.7 Transactions

A *Session* may be optionally specified as *transacted*. Each transacted session supports a single series of transactions. Each transaction groups a set of produced messages and a set of consumed messages into an atomic unit of work. In effect, transactions organize a session's input message stream and output message stream into series of atomic units. When a transaction commits, its atomic unit of input is acknowledged and its associated atomic unit of output is sent. If a transaction rollback is done, its produced messages are destroyed and its consumed messages are automatically recovered. For more information on session recovery see Section 4.4.13, “Message Acknowledgment”

A transaction is completed using either its session’s *commit()* or *rollback()* method. The completion of a session’s current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

JTS or some other transaction monitor facility may be used to combine a session’s transaction with transactions on other resources (databases, other JMS Sessions, etc.). Since Java distributed transactions are controlled via the JTA transaction demarcation API, use of the session’s *commit* and *rollback* methods in this context throws a JMS *TransactionInProgressException*.

4.4.8 Distributed Transactions

JMS does not require that a provider support distributed transactions; however, it does define that if a provider supplies this support it should be done via the JTA *XAResource* API.

A JMS provider may also be a distributed transaction monitor. If it is, it should provide control of the transaction via the JTA API.

Although it is possible for a JMS client to handle distributed transactions directly, it is unlikely that many JMS clients will do this. Support for JTA in JMS is targeted at systems vendors who will be integrating JMS into their application server products. See Chapter 8, “JMS Application Server Facilities“ for more information.

4.4.9 Multiple Sessions

A client may create multiple sessions. Each session is an independent producer and consumer of messages.

For Pub/Sub, if two sessions each have a subscriber that subscribes to the same Topic, each Subscriber is given each message. Delivery to one Subscriber does not block if the other gets behind.

For PTP, JMS does not specify the semantics of concurrent QueueReceivers for the same queue; however, JMS does not prohibit a provider from supporting this.

4.4.10 Message Order

JMS clients need to understand when they can depend on message order and when they cannot.

4.4.11 Order of Message Receipt

Messages consumed by a session define a serial order. This order is important because it defines the effect of message acknowledgment. See Section 4.4.13, “Message Acknowledgment” for more details. The messages for each of a session’s consumers are interleaved in a session’s input message stream.

JMS defines that messages sent by a session to a destination must be received in the order they were sent (see below for a few qualifications). This defines a partial ordering constraint on a session’s input message stream.

If a Session is servicing two *MessageConsumers* for the same destination, each message for the destination appears only once in the session’s input message stream; and, each message is delivered to only one of the consumers. See Section 4.5, “MessageConsumer” for more information.

JMS does not define order of message receipt across destinations or across a destination’s messages sent from multiple sessions. This aspect of a session’s input message stream order is timing dependent. It is not under application control.

4.4.12 Order of Message Sends

Although clients loosely view the messages they produce within a session as forming a serial stream of sent messages, the total ordering of this stream is not significant. The only ordering that is visible to receiving clients is the order of messages a session sends to a particular destination. Several things can affect this order:

- Messages of higher priority may jump ahead of prior, lower priority messages.
- There is also the possibility that a client may not receive a NON_PERSISTENT message due to a JMS provider failure.
- If both PERSISTENT and NON_PERSISTENT messages are sent to a destination, order is only guaranteed within delivery mode. That is, a later NON_PERSISTENT message may arrive ahead of an earlier PERSISTENT

message; however, it will never arrive ahead of an earlier NON_PERSISTENT message.

- A client may use a transacted session to group its sent messages into atomic units (the producer component of a JMS transaction). A transaction's order of messages to a particular destination is significant. The order of sent messages across destinations is not significant. See Section 4.4.7, "Transactions" for more information.

4.4.13 Message Acknowledgment

If a session is transacted, message acknowledgment is handled automatically by *commit* and recovery is handled automatically by *rollback*.

If a session is not transacted, there are three acknowledgment options and recovery is handled manually:

- DUPS_OK_ACKNOWLEDGE - This option instructs the session to lazily acknowledge the delivery of messages. This is likely to result in the delivery of some duplicate messages if JMS fails, it should only be used by consumers that are tolerant of duplicate messages. Its benefit is the reduction of session overhead achieved by minimizing the work the session does to prevent duplicates.
- AUTO_ACKNOWLEDGE - With this option, the session automatically acknowledges a client's receipt of a message when it has either successfully returned from a call to receive or the *MessageListener* it has called to process the message successfully returns.
- CLIENT_ACKNOWLEDGE - With this option, a client acknowledges a message by calling the message's *acknowledge* method. Acknowledging a consumed message automatically acknowledges the receipt of all messages that have been delivered by its session.

When CLIENT_ACKNOWLEDGE mode is used, a client may build up a large number of unacknowledged messages while attempting to process them. A JMS provider should provide administrators with a way to limit client over-run so that clients are not driven to resource exhaustion and ensuing failure when some resource they are using is temporarily blocked.

A session's *recover* method is used to stop a session and restart it with its first unacknowledged message. In effect, the session's series of delivered messages is reset to the point after its last acknowledged message. The messages it now delivers may be different from that which were originally delivered due to message expiration and the arrival of higher priority messages.

A session must set the *redelivered* flag of messages it redelivers due to a recovery.

4.4.14 Duplicate Delivery of Messages

A JMS provider must never deliver a second copy of an acknowledged message.

When a client uses the AUTO_ACKNOWLEDGE mode it is not in direct control of message acknowledgment. Since such clients cannot know for certain if a particular message has been acknowledged, they must be prepared for re-delivery of the last consumed message. This can be caused by the client completing its work just prior to a failure that prevents the message acknowledgment from occurring. Only a session's last consumed message is subject to this ambiguity.

4.4.15 Duplicate Production of Messages

JMS providers must never produce duplicate messages. This means that a client that produces a message can rely on its JMS provider to insure the consumers of the message will only receive it once. No client error can cause a provider to duplicate a message.

If a failure occurs between the time a client commits its work on a Session and the commit method returns, the client cannot determine if the transaction was committed or rolled back. The same ambiguity exists when a failure occurs between the non-transactional send of a PERSISTENT message and the return from the sending method.

It is up to a JMS application to deal with this ambiguity. In some cases, this may cause a client to produce functionally duplicate messages.

A message that is redelivered due to session recovery is not considered a duplicate message.

4.4.16 Serial Execution of Client Code

Even though the Java language provides built-in support for multi-threading, writing multi-threaded programs is still more difficult than writing single threaded ones.

For this reason, JMS does not cause concurrent execution of client code unless a client explicitly requests it. One way this is done is to define that a session serializes all asynchronous delivery of messages.

To receive messages asynchronously, a client registers an object that implements the JMS *MessageListener* interface with a *MessageConsumer. In effect, a Session uses a single thread to run all its MessageListeners*. While the thread is busy executing one listener, all other messages to be asynchronously delivered to the session must wait.

4.4.17 Concurrent Message Delivery

Clients which desire concurrent delivery can use multiple sessions. In effect, each session's listener thread runs concurrently. While a listener on one session is executing, a listener on another session may also be executing.

4.5 MessageConsumer

A client uses a *MessageConsumer* to receive messages from a destination. It is created by passing a destination to a session's create *MessageConsumer* method.

A consumer can be created with a message selector. This allows the client to restrict the messages delivered to the consumer to those that match the selector. See Section 3.8.1, "Message Selector" for more information.

Although a session allows the creation of multiple *MessageConsumers* per destination, it will only deliver each message for a destination to one consumer. If more than one consumer could receive it, the session randomly selects one to deliver it to. See Section 4.4.9, "Multiple Sessions" for information on consuming the same destination from multiple sessions.

A client may either synchronously receive a consumer's messages or have the provider asynchronously deliver them as they arrive.

4.5.1 Synchronous Delivery

A client can request the next message from a *MessageConsumer* using one of its *receive* methods. There are several variations of receive that allow a client to poll or wait for the next message.

4.5.2 Asynchronous Delivery

A client can register an object that implements the JMS *MessageListener* interface with a *MessageConsumer*. As messages arrive for the consumer, the provider delivers them by calling the listener's *onMessage* method.

It is possible for a listener to throw a *RuntimeException*; however, this is considered a client programming error. Well behaved listeners should catch such exceptions and attempt to divert messages causing them to some form of application-specific 'unprocessable message' destination.

The result of a listener throwing a *RuntimeException* depends on the session's acknowledgment mode.

- AUTO_ACKNOWLEDGE - the message will be immediately redelivered. The number of times a JMS provider will redeliver the same message before giving up is provider dependent.
- CLIENT_ACKNOWLEDGE - the next message for the listener is delivered. If a client wishes to have the previous unacknowledged message redelivered it must manually recover the session.
- Transactional Session - the next message for the listener is delivered. The client can either commit or rollback the Session (i.e. a *RuntimeException* does not automatically rollback the session).

JMS providers should flag clients with message listeners that are throwing *RuntimeExceptions* as possibly malfunctioning.

See Section 4.4.16, "Serial Execution of Client Code" for information about how *onMessage* calls are serialized by a session.

4.6 MessageProducer

A client uses a *MessageProducer* to send messages to a *Destination*. It is created by passing a destination to a session's create *MessageProducer* method.

A client also has the option of creating a producer without supplying a destination. In this case, a destination must be input on every send operation. A typical use for this style of producer is to send replies to requests using the request's *JMSReplyTo* destination.

A client can specify a default delivery mode, priority, and time-to-live for messages sent by a producer. It can also specify delivery mode, priority, and time-to-live per message.

Each time a client creates a *MessageProducer*, it defines a new sequence of messages that have no ordering relationship with the messages it has previously sent.

See Section 3.4.9, "JMSExpiration" for more information on time-to-live. See Section 3.4.10, "JMSPriority" for more information on priority.

4.7 Message Delivery Mode

JMS supports two modes of message delivery.

- The NON_PERSISTENT mode is the lowest overhead delivery mode because it does not require that the message be logged to stable storage. A JMS provider failure can cause a NON_PERSISTENT message to be lost.
- The PERSISTENT mode instructs the JMS provider to log the message to stable storage as part of the client's send operation. This insures the message will survive a JMS provider failure.

A JMS provider must deliver a NON_PERSISTENT message with an *at-most-once* guarantee. This means it may lose the message but it must not deliver it twice.

A JMS provider must deliver a PERSISTENT message with a *once-and-only-once* guarantee. It must not lose it and it must not deliver it twice.

4.8 Message Time-To-Live

A client can specify a time-to-live value in milliseconds for each message it sends. This value defines a message expiration time which is the sum of the message's time-to-live and the GMT it is sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed).

A JMS provider should do its best to accurately expire messages; however, JMS does not define the accuracy provided. It is not acceptable to simply ignore time-to-live.

For more information on message expiration, see Section 3.4.9, “JMSExpiration”.

4.9 Exceptions

JMSException is the base class for all JMS exceptions. See Chapter 7, “JMS Exceptions“ for more information.

4.10 Reliability

Most clients should use producers that produce PERSISTENT messages. This insures once-and-only-once message delivery.

In some cases, an application may only require at-most-once message delivery for some of its messages. This is accomplished by publishing NON_PERSISTENT messages. These messages have lower overhead because they are not required to be logged to stable store; however, they may be lost if the JMS fails. Both PERSISTENT and NON_PERSISTENT messages can be published to the same destination.

Normally, a consumer fully processes each message before acknowledging its receipt to JMS. This insures that JMS does not discard a partially processed message due to machine failure, etc. A consumer accomplishes this by using either a transacted or CLIENT_ACKNOWLEDGE’ed session.

It is expected that important messages will be produced with a PERSISTENT delivery mode within a transaction; and, consumed within a transaction.

When this is done, applications have the highest level of assurance that a message has been properly produced; reliably delivered; and, accurately consumed. Non-transactional production and consumption can also achieve the same level of assurance; however, this requires careful programming.

A JMS provider may have resource restrictions that limit the number of messages that can be held for high volume or non-responsive clients. If messages are dropped due to resource limits this is usually a serious administrative issue that needs attention. Correct functioning of JMS requires that clients are responsive and adequate resources to service them are available.

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.

5.2 Queue Management

JMS does not define facilities for creating, administering, or deleting long lived queues (it does provide such a mechanism for *TemporaryQueues*). 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 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 *QueueConnection*. It is a system defined queue that can only be consumed by the QueueConnection that created it.

See Section 4.4.3, “Creating TemporaryDestinations“ for more information.

5.5 QueueConnectionFactory

A client uses a *QueueConnectionFactory* to create *QueueConnections* 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 *QueueSessions* 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 *TemporaryQueues*.

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

If a *QueueReceiver* specifies a message selector, the messages that are not selected remain on the queue.

For more information see Section 4.5, "MessageConsumer".

5.9 QueueSender

A client uses a *QueueSender* to send messages to a queue.

For more information see Section 4.6, "MessageProducer".

5.10 QueueBrowser

A client uses a *QueueBrowser* to look at messages on a queue without removing them.

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 only contain 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.

5.11 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.

5.12 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 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.

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

Non-durable 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 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.

6.4 Topic Management

Some products require that topics be statically defined with associated authorization control lists, etc.; 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 restriction 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 *TopicConnection*. It is a system defined *Topic* that can only be consumed by the *TopicConnection* that created it.

See Section 4.4.3, "Creating TemporaryDestinations" for more information.

6.7 TopicConnectionFactory

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

See Section 4.2, "Administered Objects" for more information about JMS ConnectionFactory's.

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 TopicSessions 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 TemporaryTopics. It also provides the *unsubscribe* method for deleting its client's durable subscriptions.

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*. 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”.

Regular *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.

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*. JMS retains a record of this durable subscription and insures that all messages from the topic's publishers are retained until they are either 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 which 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. Changing a durable subscription is equivalent to deleting and recreating it.

TopicSessions 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.

6.12 Recovery and Redelivery

Due to the temporary nature of non-durable subscriptions, attempting to recover one's unacknowledged messages may drop messages. When a non-durable 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.

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 administrators viewpoint, there may be other tasks need to support their creation.

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.

Non-durable subscribers should only be used when missed messages are tolerable.

Table 0-10 Pub/Sub Reliability

How Published	**Non-Durable 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 Exceptions 7

7.1 Overview

This chapter provides an overview of JMS exception handling and defines the standard JMS exceptions.

7.2 The JMSException

JMS defines *JMSException* as the root interface for exceptions thrown by JMS methods. *JMSException* is a checked exception and catching it provides a generic way of handling all JMS related exceptions. *JMSException* provides the following information:

- A provider-specific string describing the error - This string is the standard Java exception message, and is available via *getMessage()*.
- A provider-specific, string error code
- A reference to another exception - Often a JMS exception will be the result of a lower level problem. If appropriate, this lower level exception can be linked to the JMS exception.

JMS methods only include *JMSException* in their signatures. JMS methods can throw any JMS standard exception as well as any JMS provider specific exception. **The javadoc for JMS methods documents only the mandatory exception cases.**

7.3 Standard Exceptions

In addition to *JMSException*, JMS defines several additional exceptions that standardize the reporting of basic error conditions.

There are only a few cases where JMS mandates that a specific JMS exception must be thrown. These cases are indicated by the words **must be** in the exception description. **These cases are the only ones on which client logic should depend on a specific problem resulting in a specific JMS exception being thrown.**

In the remainder of cases, it is strongly suggested that JMS providers use one of the standard exceptions where possible. JMS providers may also derive provider specific exceptions from these if needed.

JMS defines the following standard exceptions:

- *IllegalStateException*: This exception is thrown when a method is invoked at an illegal or inappropriate time or if the provider is not in an appropriate state for the requested operation. For example, this exception should be thrown if *Session.commit()* is called on a non-transacted session.
- *JMSSecurityException*: This exception **must be** thrown when a provider rejects a user name/password submitted by a client. It may also be thrown for any case where a security restriction prevents a method from completing.
- *InvalidClientIDException*: This exception **must be** thrown when a client attempts to set a connection's client id to a value that is rejected by a provider.
- *InvalidDestinationException*: This exception **must be** thrown when a destination is either not understood by a provider or is no longer valid.
- *InvalidSelectorException*: This exception **must be** thrown when a JMS client attempts to give a provider a message selector with invalid syntax.
- *MessageEOFException*: This exception **must be** thrown when an unexpected end of stream has been reached when a *StreamMessage* or *BytesMessage* is being read.
- *MessageFormatException*: This exception **must be** thrown when a JMS client attempts to use a data type not supported by a message or attempts to read data in a message as the wrong type. It must also be thrown when equivalent type errors are made with message property values. For example, this exception must be thrown if *StreamMessage.setObject()* is given an

unsupported class or if *StreamMessage.getShort()* is used to read a boolean value. This exception also **must be** thrown if a provider is given a type of message it cannot accept. Note that the special case of a failure caused by attempting to read improperly formatted *String* data as numeric values should throw the *java.lang.NumberFormatException*.

- *MessageNotReadableException*: This exception **must be** thrown when a JMS client attempts to read a write-only message.
- *MessageNotWriteableException*: This exception **must be** thrown when a JMS client attempts to write to a read-only message.
- *ResourceAllocationException*: This exception is thrown when a provider is unable to allocate the resources required by a method. For example, this exception should be throw when a call to *createTopicConnection* fails due to lack of JMS Provider resources.
- *TransactionInProgressException*: This exception is thrown when an operation is invalid because a transaction is in progress. For instance, attempting to call Session.commit() when a session is part of a distributed transaction should throw a *TransactionInProgressException.*
- *TransactionRolledBackException*: This exception **must be** thrown when a call to Session.commit results in a rollback of the current transaction.

JMS Application Server Facilities 8

8.1 Overview

This chapter describes JMS facilities for concurrent processing of a subscription's messages. It also defines how a JMS provider supplies JTS aware sessions. These facilities can also be used by expert JMS clients.

These facilities are a special category of JMS. They will only be supported by the more sophisticated JMS providers.

8.2 Concurrent Processing of a Subscription's Messages

JMS provides a special facility for creating a *MessageConsumer* that can concurrently consume messages.

This facility partitions the work into three roles:

- JMS provider - its role is to deliver the messages.
- Application Server - its role is to create the consumer and manage the threads used by the concurrent *MessageListener* objects.
- Application - its role is to define a subscription with a destination and optionally a message selector and provide a single threaded *MessageListener* class to consume its messages. An application server will construct multiple objects of this class to concurrently consume messages.

8.2.1 Session

Sessions provide three methods for use by application servers:

- *setMessageListener()* and *getMessageListener()* - a session's *MessageListener* consumes messages that have been assigned to the session by a *ConnectionConsumer*, see below.
- *run()* - causes the messages assigned to its session by a *ConnectionConsumer* to be serially processed by the session's *MessageListener*. When the listener returns from processing the last message, *run*() returns.

An application server would typically be given a *MessageListener* class which contains the single threaded code written by an application programmer to process messages. It would also be given the destination and message selector that specified the messages the listener was to consume.

An application server would take care of creating the JMS *Connection*, *ConnectionConsumer*, and *Sessions* it needs to handle message processing. It would create as many *MessageListener* instances as it needed and register each with its own session.

Since many listeners will need to use the services of its session, the listener is likely to require that its session be passed to it as a constructor parameter.

8.2.2 ServerSession

A *ServerSession* is an object implemented by an application server. It is used by an application server to associate a thread with a JMS session.

A *ServerSession* implements two methods:

- *getSession()* - returns the *ServerSession's* JMS *Session*.
- *start()* - starts the execution of the *ServerSession* thread and results in the execution of the associated JMS *Session's run* method.

8.2.3 ServerSessionPool

A *ServerSessionPool* is an object implemented by an application server to provide a pool of *ServerSessions* for processing the messages of a *ConnectionConsumer.*

Its only method is *getServerSession()*. This removes a *ServerSession* from the pool and gives it to the caller (which is assumed to be a *ConnectionConsumer*) to use for consuming one or more messages.

JMS does not architect how the pool is implemented. It could be a static pool of *ServerSessions* or it could use a sophisticated algorithm to dynamically create *ServerSessions* as needed.

If the *ServerSessionPool* is out of *ServerSessions*, the *getServerSession()* method may block. If a *ConnectionConsumer* is blocked, it cannot deliver new messages until a *ServerSession* is eventually returned.

8.2.4 ConnectionConsumer

For application servers, connections provide a special facility for creating a *ConnectionConsumer*. The messages it is to consume are specified by a destination and a message selector. In addition, a *ConnectionConsumer* must be given a *ServerSessionPool* to use for processing its messages. A *maxMessages* value is specified to limit the number of messages a *ConnectionConsumer* may load at one time into a *ServerSession's Session*.

Normally, when traffic is light, a *ConnectionConsumer* gets a *ServerSession* from its pool; loads its *Session* with a single message; and, starts it. As traffic picks up, messages can back up. If this happens, a *ConnectionConsumer* can load each *Session* with more than one message. This reduces the thread context switches and minimizes resource use at the expense of some serialization of a message processing.

8.2.5 How a ConnectionConsumer Uses a ServerSession

A *ConnectionConsumer* implemented by a JMS provider uses a *ServerSession* to process one or more messages that have arrived. It does this as follows:

- It gets a *ServerSession* from the its *ServerSessionPool*
- It gets the *ServerSession's Session*
- It loads the *Session* with one or more messages
- It then starts the *ServerSession* to consume these messages

A *ConnectionConsumer* for a *QueueConnection* will expect to load its messages into a *QueueSession*; as would one for a *TopicConnection* expect to load a *TopicSession*.

Note that JMS does not architect how the *ConnectionConsumer* loads the *Session* with messages. Since both the *ConnectionConsumer* and *Session* are implemented by the same JMS provider, they can accomplish the load using a private mechanism.

8.2.6 How an App Server Implements a ServerSession

JMS does not architect the implementation of a *ServerSession*. A typical implementation is presented here to illustrate the concept:

- An app server creates a *Thread* for a *ServerSession* registering the *ServerSession's runObject*. The implementation of this *runObject* is private to the app server.
- The *ServerSession's start* method calls its *Thread's start* method. Like all Java threads, a call to start initiates execution of the started thread and calls the thread's *runObject*. The caller to *ServerSession.start* (the *ConnectionConsumer*) and the *ServerSession runObject* are now running in different threads.
- The *runObject* will do some housekeeping and then call its *Session's run()* method. On return, the *runObject* puts its *ServerSession* back into its *ServerSessionPool* and returns. This terminates execution of the *ServerSession's* thread and the cycle starts again.

8.2.7 The Result

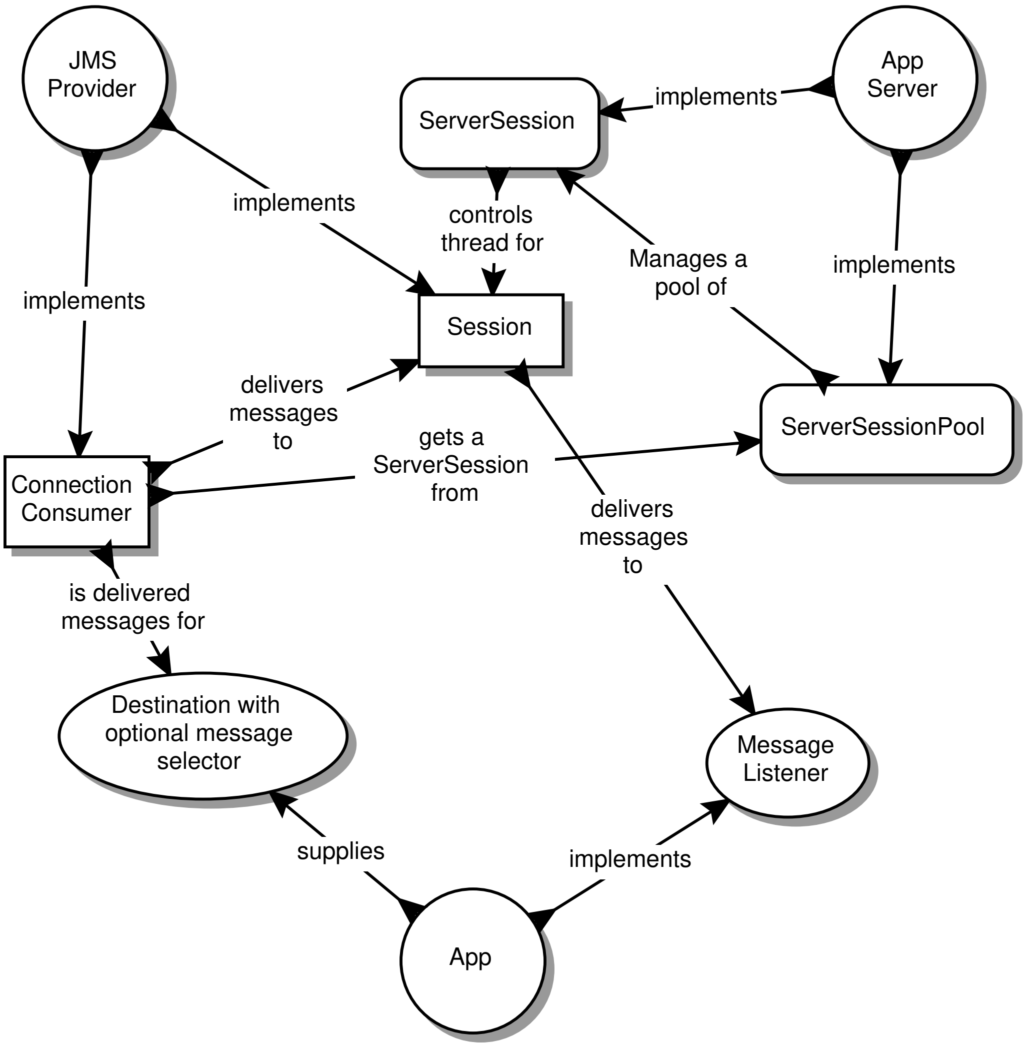
JMS has defined a flexible mechanism that partitions the job of concurrent message consumption into roles that are well suited for each participant.

The application programmer provides a simple to write, single threaded implementation of *MessageListener*.

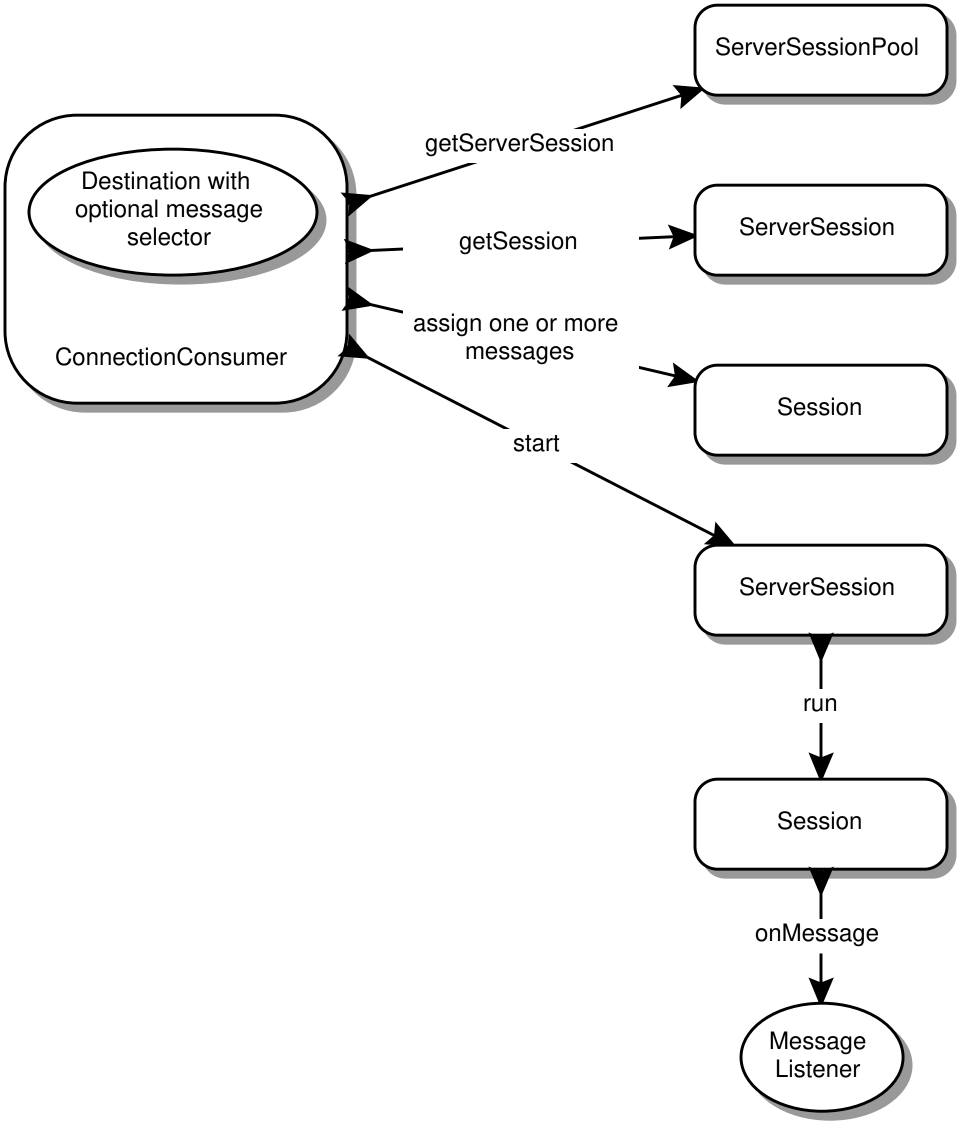
The JMS provider retains control of its messages until they are delivered to the *MessageListener*. This insures it is under direct control of message acknowledgment.

The application server is in control of setting up the *ConnectionConsumer* and managing all the threads used for executing its *MessageListeners*.

The following diagram illustrates the relationship between the three roles and the objects they implement.



The following diagram illustrates the process a ConnectionConsumer uses to deliver a message to a MessageListener.



8.3 XAConnectionFactory

Some application servers provide support for grouping resource use into a distributed transaction. To include JMS transactions in a distributed transaction, an application server requires a Java Transaction API (JTA) capable JMS provider. A JMS provider exposes its JTA support using a JMS *XAConnectionFactory* which an application server uses to create *XAConnections*.

XAConnectionFactory provides the same authentication options as *ConnectionFactory*.

XAConnectionFactory's are JMS administered objects just like *ConnectionFactory's*. It is expected that application servers will find them using JNDI.

8.4 XAConnection

XAConnection extends the capability of Connection by providing the ability to create *XASessions*.

8.5 XASession

XASession provides access to what looks like a normal *Session* object and a *javax.transaction.xa.XAResource* object which controls the session's transaction context. The functionality of *XAResource* closely resembles that defined by the standard X/Open XA Resource interface.

An application server controls the transactional assignment of an *XASession* by obtaining its *XAResource*. It uses the *XAResource* to assign the session to a distributed transaction; prepare and commit work on the transaction; etc.

An *XAResource* provides some fairly sophisticated facilities for interleaving work on multiple transactions; recovering a list of transactions in progress; etc. A JTA aware JMS provider must fully implement this functionality. This could be done by using the services of a database that supports XA or a JMS provider may choose to implement this functionality from scratch.

A client of the application server is given the *XASession's Session*. Behind the scenes, the application server controls the transaction management of the underlying *XASession*.

It is important to note that a distributed transaction context does *not* flow with a message; and, the receipt of the message cannot be part of the same transaction that produced the message. This is the fundamental difference between messaging and synchronized processing. Message producers and consumers use an alternative approach to reliability that is built upon a JMS providers ability to supply a once-and-only-once message delivery guarantee.

To reiterate, the act of producing and/or consuming messages in a Session can be transactional. The act of producing and consuming a specific message across different sessions cannot.

8.6 JMS Application Server Interfaces

Both PTP and Pub/Sub domains provide their own versions of JTS aware JMS facilities.

Table 0-11 Relationship of PTP and Pub/Sub expert interfaces

JMS Root	**PTP Interface**	**Pub/Sub Interface**
ServerSessionPool		
ServerSession		
ConnectionConsumer		
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

JMS Sample Code 9

9.1 Sample JMS Code

9.1.1 Background

The following code examples show several ways a client could use the various JMS API message types with both the Point-to-Point and Publish/Subscribe messaging, to obtain stock quote information.

Note that no exception handling code is included. This makes it easier to see what's happening.

Lets assume that there is a stock quote service which sends out the stock quote messages. This could be done in many different ways, and as will be shown below, the construction of these messages is the same for both PTP and Pub/Sub messaging.

Before we can send and receive message, the client application needs to do some initial setup.

9.1.2 Point-to-Point Setup

Here is an example that shows how to send and receive messages using Point-to-Point messaging.

9.1.2.1 Getting a QueueConnectionFactory

Both the message sender and receiver need to get a queue connection factory and use it to setup both a queue connection and a queue session.

An administrator typically has created and configured a QueueConnectionFactory for our use. We typically get it by looking it up using JNDI.

```
QueueConnectionFactory queueConnectionFactory;

Context messaging = new InitialContext();

queueConnectionFactory = (QueueConnectionFactory)
messaging.lookup(“QueueConnectionFactory”);
```

9.1.2.2 Getting a Message Queue

An administrator has created and configured "StockQueue" for our use. Again, we use JNDI to look it up.

```
Queue stockQueue;

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

9.1.2.3 Getting a QueueConnection

Having obtained the QueueConnectionFactory, we use it to create a QueueConnection.

```
QueueConnection queueConnection;

queueConnection = queueConnectionFactory.createQueueConnection();
```

9.1.2.4 Getting a QueueSession

Having obtained the QueueConnection, we use it to create a QueueSession. This will be used to create a QueueSender (if we want to send messages) or a QueueReceiver (if we want to receive messages).

We use the QueueConnection.createQueueSession method to do this, supplying two parameters:

- A boolean indicating whether this queue session will be transacted on not
- The mode of acknowledging message receipt

```
QueueSession session;
```

```
session = queueConnection.createQueueSession(false,
Session.AUTO_ACKNOWLEDGE);
```

9.1.2.5 Getting a QueueSender

Having obtained the QueueSession, we use it to create a QueueSender, if we are going to be sending messages to the queue. This is achieved with the QueueSession.createSender method. We supply one parameter; the queue we are going to be sending messages to.

```
QueueSender sender;

sender = session.createSender(queue);
```

9.1.2.6 Getting a QueueReceiver

In a similar fashion, we get a QueueReceiver if we are going to be receiving messages from the queue. This is achieved with the QueueSession.createReceiver method. We supply one parameter; the queue we are going to be receiving messages from.

```
QueueReceiver receiver;

receiver = session.createReceiver(queue);
```

9.1.3 Publish/Subscribe Messaging Domain Setup

Here is an equivalent example that shows how to send and receive messages using Pub/Sub messaging:

9.1.3.1 Getting a TopicConnectionFactory

Both the message publisher and subscriber need to get a topic connection factory and use it to setup a connection and a topic session.

An administrator typically has created and configured a TopicConnectionFactory for our use. We typically get it by looking it up using JNDI.

```
TopicConnectionFactory topicConnectionFactory;

Context messaging = new InitialContext();
```

```
topicConnectionFactory = (TopicConnectionFactory)
messaging.lookup("TopicConnectionFactory");
```

9.1.3.2 Getting a Message Topic

An administrator has created and configured "StockQueue" for our use. Again, we use JNDI to look it up.

```
Topic stockTopic;

stockTopic = (Topic) messaging.lookup("StockTopic");
```

9.1.3.3 Getting a TopicConnection

Having obtained the TopicConnectionFactory, we use it to create a TopicConnection.

```
TopicConnection topicConnection;

topicConnection = topicConnectionFactory.createTopicConnection();
```

9.1.3.4 Getting a TopicSession

Having obtained the TopicConnection, we use it to create a TopicSession. This will be used to create a TopicPublisher (if we want to publish messages) or a TopicSubscriber (if we want to receive messages).

We use the TopicConnection.createTopicSession method to do this, supplying two parameters:

- A boolean indicating whether this topic session will be transacted on not;
- The mode of acknowledging message receipt

```
TopicSession session;

session = topicConnection.createTopicSession(false,
Session.CLIENT_ACKNOWLEDGE);
```

9.1.3.5 Getting a TopicSubscriber

Having obtained the TopicSession, we use it to obtain a TopicSubscriber if we are going to subscribe to the topic in order to receive messages. This is achieved with the TopicSession.createSubscriber method. We supply the topic we wish to subscribe to.

First we create a topic subscriber:

```
TopicSubscriber subscriber;

subscriber = session.createSubscriber(stockTopic);
```

In order to asynchronously receive messages as they are delivered to our subscriber, we need to create a message listener which implements the javax.jms.MessageListener interface.

Our listener class (lets call it StockListener.java) would look something like:

```
public class StockListener implements javax.jms.MessageListener {

     void
     onMessage(Message message) {

          // unpack and handle the messages we receive.

      }
 }
```

Next we register this message listener with our subscriber:

```
StockListener myListener;
subscriber.setMessageListener(myListener);
```

9.1.3.6 Getting a TopicPublisher

Having obtained the TopicSession, we use it to create a TopicPublisher if we are going to be publishing messages to the topic. This is achieved with the TopicSession.createPublisher method. We supply one parameter; the topic we are going to be publishing messages to.

```
TopicPublisher publisher;

publisher = session.createPublisher(stockTopic);
```

9.1.4 JMS Message Types

We now need to put the stock information into one of the JMS message types. Our queue or topic session has all the methods needed for creating a message of the type we want.

9.1.4.1 Using a BytesMessage

The stock quote information could be in a binary format that the server knows how to construct, and the client knows how to interpret and display as a stock quote.

Such a message could be constructed with:

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

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

9.1.4.2 Using a TextMessage

The stock quote information could be sent as a human-readable text string which is just read and displayed by the client.

We can create such a message with:

```
StringBuffer stockData;    // stock information as a StringBuffer
TextMessage message;

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

9.1.4.3 Using a MapMessage

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

- stock quote name - String
- current value - double
- time of quote - long
- last change - double

- stock information - String

The client would receive the whole map message, but might only be interested in displaying part of this information. It could extract the required fields from the message, ignoring the rest.

Construction of a stock MapMessage would be something like:

```
String stockName;       // the name of the stock quote.
double stockValue;      // the current value of the stock.
long   stockTime;       // the time of the stock quote.
double stockDiff;       // the +/- change in the stock quote.
String stockInfo;       // information on this stock.
MapMessage message;

message = session.createMapMessage();
```

Note that the following can be set in any order.

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

9.1.4.4 *Using a StreamMessage*

In a similar fashion to the map message, the server could send out a message consisting of various fields written in sequence to the message, each in their own primitive type:

- stock quote name - String
- current value - double
- time of quote - long
- last change - double
- stock information - String.

The client might be only interested in some of the message field, but in the case of a stream message, it has to read (and potentially throw away) each field in turn.

The stock StreamMessage could be created with:

```
String stockName;      // the name of the stock quote.
double stockValue;     // the current value of the stock.
long   stockTime;      // the time of the stock quote.
double stockDiff;      // the +/- change in the stock quote.
String stockInfo;      // information on this stock.
StreamMessage message;

message = session.createStreamMessage();
```

Note that the following have to be written in the order they will be read.

```
message.writeString(stockName);
message.writeDouble(stockValue);
message.writeLong(stockTime);
message.writeDouble(stockDiff);
message.writeString(stockInfo);
```

9.1.4.5 *Using an ObjectMessage*

The stock information could be sent in the form of a special Stock Java object, which the client extracts, then uses its methods to obtain the stock information it requires.

Construction of such an object message could look like:

```
String stockName;      // the name of the stock quote.
double stockValue;     // the current value of the stock.
long   stockTime;      // the time of the stock quote.
double stockDiff;      // the +/- change in the stock quote.
String stockInfo;      // information on this stock.
StockObject stockObject = new StockObject();
ObjectMessage message;
```

These values could have been passed in when the stock object was constructed.

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

message = session.createObjectMessage();
message.setObject(stockObject);
```

9.1.5 Point-to-Point Sending and Receiving

Here's how to send and receive messages in the Point-To-Point messaging domain:

9.1.5.1 Sending a Message

Sending of all of these message types is done in the same way:

```
sender.send(message);
```

9.1.5.2 Receiving a Message

Receiving of all of these message types is done in the same way. Here is how to receive the next message in the queue. Note that this call will block indefinitely until a message arrives on the queue.

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

9.1.6 Publish/Subscribe Sending and Receiving

Here's how to send and receive messages in the Publish/Subscribe messaging domain:

9.1.6.1 Sending a Message

Sending (publishing) of all of these message types is done in the same way:

```
publisher.publish(message);
```

9.1.6.2 Receiving a Message

Receiving of all of these message types is done in the same way. When the client subscribed to the topic, it registered a message listener. This listener will be asynchronously notified whenever a message has been published to the topic. This is done via the onMessage() method in that listener class. It is up to the client to process the message there.

```
void
onMessage(Message message) {
```

```
// unpack and handle the messages we receive.

}
```

9.1.7 Unpacking messages

Unpacking of a message is different for each message type, but works the same way in both the Point-To-Point and the Publish/Subscribe messaging domains.

9.1.7.1 Unpacking a BytesMessage

```
byte[]      stockData;     // stock information as a byte array
int         length;

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

9.1.7.2 Unpacking a TextMessage

```
StringBuffer stockData;    // stock information as a StringBuffer

stockData = message.getText();
```

9.1.7.3 Unpacking a MapMessage

Note that the following can be gotten in any order.

```
String stockName;      // the name of the stock quote.
double stockValue;     // the current value of the stock.
long   stockTime;      // the time of the stock quote.
double stockDiff;      // the +/- change in the stock quote.
String stockInfo;      // information on this stock.

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

9.1.7.4 Unpacking a StreamMessage

Note that the following have to be read in the order they were written.

```
String stockName;      // the name of the stock quote.
```

```
double stockValue;     // the current value of the stock.
long   stockTime;      // the time of the stock quote.
double stockDiff;      // the +/- 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();
```

9.1.7.5 Unpacking an ObjectMessage

```
String stockName;      // the name of the stock quote.
double stockValue;     // the current value of the stock.
long   stockTime;      // the time of the stock quote.
double stockDiff;      // the +/- change in the stock quote.
String stockInfo;      // information on this stock.
StockObject stockObject = new StockObject();

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

9.1.8 Message Selection

We might be interested in only certain stock quotes. We can create a message selector to achieve this. For this example, we will assume that we only want to receive stock quotes for Sun MicroSystems (SUNW) and IBM (IBM), and that the stock name has been set in a property called *name*.

Our message selector String would look like this:

```
String selector;

selector = new String("(name = 'SUNW') OR (name = 'IBM')");
```

9.1.8.1 Point-To-Point QueueReceiver Setup

When we create our QueueReceiver, we pass in the message selector String:

```
QueueReceiver receiver;
```

```
receiver = session.createReceiver(queue, selector);
```

9.1.8.2 Publish/Subscribe TopicSubscriber Setup

When we create our TopicSubscriber, we pass in the message selector String.

```
TopicSubscriber subscriber;

subscriber = session.createSubscriber(topic, selector);
```

Now we will only receive the stock quotes we are interested in.

Issues 10

10.1 Resolved Issues

10.1.1 JDK 1.1.x Compatibility

JMS is compatible with JDK 1.1.x.

10.1.2 Distributed Java Event Model

JMS can be used, in general, as a notification service; however, it does not define a distributed version of Java Events.

One alternative for implementing distributed Java Events would be as JavaBeans that transparently, to the event producer and listener beans, distribute the events via JMS.

10.1.3 Should the Two JMS Domains, PTP and Pub/Sub, be merged?

Even though there are many similarities, providing separate domains still seem to be important.

It means that vendors aren't forced to support facilities out of their domain and client code can be a bit more portable because products more fully support a domain (as opposed to supporting less defined subsets of a merged domain).

10.1.4 Should JMS Specify a Set of JMS Java Beans?

JMS is a low level API and like other Java low level API's it doesn't lend itself to direct representation as JavaBeans.

10.1.5 Alignment with the CORBA Notification Service

The Notification service adds filtering, delivery guarantee semantics, durable connections, and the assembly of event networks to the CORBA Event Service. It gets its delivery guarantee semantics from the CORBA Messaging Service (which defines asynchronous CORBA method invocation).

Java technology is well integrated with CORBA. It provides Java IDL and COS Naming. In addition, OMG has recently defined RMI over IIOP.

It is expected that most use of IIOP from Java will be via RMI. It is expected that most use of COS Naming from Java with be via JNDI (Java Naming and Directory Service). JMS is a Java specific API designed to be layered over a wide range of existing and future MOM systems (just like JNDI is layered over existing name and directory services).

10.1.6 Should JMS Provide End-to-end Synchronous Message Delivery and Notification of Delivery?

Some messaging systems provide synchronous delivery to destinations as a mechanism for implementing reliable applications. Some systems provide clients with various forms of delivery notification so that the clients can detect dropped or ignored messages. This is not the model defined by JMS.

JMS messaging provides guaranteed delivery via the once-and-only-once delivery semantics of PERSISTENT messages. In addition, message consumers can insure reliable processing of messages by using either CLIENT_ACKNOWLEDGE mode or transactional Sessions.

This achieves reliable delivery with minimum synchronization and is the enterprise messaging model most vendors and developers prefer.

JMS does not define a schema of systems messages (such as delivery notifications). If an application requires acknowledgment of message receipt, it can define an application level acknowledgment message.

These issues are more clearly understood when they are examined in the context of Pub/Sub applications. In this context, synchronous delivery and/or system acknowledgment of receipt are not an effective mechanism for implementing reliable applications (because producers by definition are not, and don't want to be, responsible for end-to-end message delivery).

10.1.7 Should JMS Provide a Send-to-List Mechanism?

Currently JMS provides a number of message send options; however, messages can only be sent to one Destination at a time.

The benefit of send-to-list is slightly less work for the programmer and the potential for the JMS provider to optimize the fact that several destinations are being sent the same message.

The down side of a send-to-list mechanism is that the list is, in effect, a group that is implemented and maintained by the client. This would complicate the administration of JMS clients.

Instead of JMS providing a send-to-list mechanism, it is recommended that providers support configuring destinations that represent a group. This allows a client to reach all consumers with a single send while insuring that groups are properly administrable.

10.1.8 Should JMS Provide Subscription Notification?

If it were possible for a publisher to detect when subscribers for a topic existed, it could inhibit publication on unsubscribed topics.

Although there may be some benefit in providing publishers with a mechanism for inhibiting publication to unsubscribed topics, the complexity this would add to JMS and the additional provider overhead it would require is not justified by its potential benefits. Instead, JMS providers should insure that they minimize the overhead for handling messages published to an unsubscribed topic.

Change History 11

11.1 Version 1.0.1

11.1.1 JMS Exceptions

A new JMS Exception chapter was added and it contains the following new information:

- Two fields were added to *JMSException* - a vendor error code and an Exception reference.
- In version 1.0, *JMSException* was the only JMS exception specified. Version 1.0.1 adds a list of standard exceptions derived from JMSException and describes when each should be thrown by JMS providers.

JavaSoft
2550 Garcia Avenue
Mountain View, CA 94043
408-343-1400

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000