Java Message Service

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

Version 2.0 (Early Draft)

Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kate Stout
Sun Microsystems
(Version 1.1)

Nigel Deakin
Oracle
(Version 2.0)

16 February 2012

License

Specification: JSR-343 Java Message Service 2.0 ("Specification")
Version: 2.0
Status: Early Draft Review
Release: 10 February 2012

NOTICE

The Specification is protected by copyright and the information described therein may be protected by one or more U.S. patents, foreign patents, or pending applications. Except as provided under the following license, no part of the Specification may be reproduced in any form by any means without the prior written authorization of Oracle America, Inc. ("Oracle") and its licensors, if any. Any use of the Specification and the information described therein will be governed by the terms and conditions of this Agreement.

Subject to the terms and conditions of this license, including your compliance with Paragraphs 1 and 2 below, Oracle hereby grants you a fully-paid, non-exclusive, non-transferable, limited license (without the right to sublicense) under Oracle's intellectual property rights to:

1. Review the Specification for the purposes of evaluation. This includes: (i) developing implementations of the Specification for your internal, non-commercial use; (ii) discussing the Specification with any third party; and (iii) excerpting brief portions of the Specification in oral or written communications which discuss the Specification provided that such excerpts do not in the aggregate constitute a significant portion of the Technology.

2. Distribute implementations of the Specification to third parties for their testing and evaluation use, provided that any such implementation:

(i) does not modify, subset, superset or otherwise extend the Licensor Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the Licensor Name Space other than those required/authorized by the Specification or Specifications being implemented;

(ii) is clearly and prominently marked with the word "UNTESTED" or "EARLY ACCESS" or "INCOMPATIBLE" or "UNSTABLE" or "BETA" in any list of available builds and in proximity to every link initiating its download, where the list or link is under Licensee's control; and

(iii) Includes the following notice:

"This is an implementation of an early-draft specification developed under the Java Community Process (JCP) and is made available for testing and evaluation purposes only. The code is not compatible with any specification of the JCP."

The grant set forth above concerning your distribution of implementations of the specification is contingent upon your agreement to terminate development and distribution of your "early draft" implementation as soon as feasible following final completion of the specification. If you fail to do so, the foregoing grant shall be considered null and void.

No provision of this Agreement shall be understood to restrict your ability to make and distribute to third parties applications written to the Specification.

Other than this limited license, you acquire no right, title or interest in or to the Specification or any other Oracle intellectual property, and the Specification may only be used in accordance with the license terms set forth herein. This license will expire on the earlier of: (a) two (2) years from the date of Release listed above; (b) the date on which the final version of the Specification is publicly released; or (c) the date on which the Java Specification Request (JSR) to which the Specification corresponds is withdrawn. In addition, this license will terminate immediately without notice from Oracle if you fail to comply with any provision of this license. Upon termination, you must cease use of or destroy the Specification.

"Licensor Name Space" means the public class or interface declarations whose names begin with "java", "javax", "com.oracle" or their equivalents in any subsequent naming convention adopted by Oracle through the Java Community Process, or any recognized successors or replacements thereof

TRADEMARKS

No right, title, or interest in or to any trademarks, service marks, or trade names of Oracle or Oracle's licensors is granted hereunder. Oracle, the Oracle logo, Java are trademarks or registered trademarks of Oracle USA, Inc. in the U.S. and other countries.

DISCLAIMER OF WARRANTIES

THE SPECIFICATION IS PROVIDED "AS IS" AND IS EXPERIMENTAL AND MAY CONTAIN DEFECTS OR DEFICIENCIES WHICH CANNOT OR WILL NOT BE CORRECTED BY ORACLE. ORACLE MAKES NO REPRESENTATIONS OR WARRANTIES, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT THAT THE CONTENTS OF THE SPECIFICATION ARE

SUITABLE FOR ANY PURPOSE OR THAT ANY PRACTICE OR IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADE SECRETS OR OTHER RIGHTS. This document does not represent any commitment to release or implement any portion of the Specification in any product.

THE SPECIFICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION THEREIN; THESE CHANGES WILL BE INCORPORATED INTO NEW VERSIONS OF THE SPECIFICATION, IF ANY. ORACLE MAY MAKE IMPROVEMENTS AND/OR CHANGES TO THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THE SPECIFICATION AT ANY TIME. Any use of such changes in the Specification will be governed by the then-current license for the applicable version of the Specification.

LIMITATION OF LIABILITY

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ORACLE OR ITS LICENSORS BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION, LOST REVENUE, PROFITS OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO ANY FURNISHING, PRACTICING, MODIFYING OR ANY USE OF THE

SPECIFICATION, EVEN IF ORACLE AND/OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You will hold Oracle (and its licensors) harmless from any claims based on your use of the Specification for any purposes other than the limited right of evaluation as described above, and from any claims that later versions or releases of any Specification furnished to you are incompatible with the Specification provided to you under this license.

RESTRICTED RIGHTS LEGEND

If this Software is being acquired by or on behalf of the U.S. Government or by a U.S. Government prime contractor or subcontractor (at any tier), then the Government's rights in the Software and accompanying documentation shall be only as set forth in this license; this is in accordance with 48 C.F.R. 227.7201 through 227.7202-4 (for Department of Defense (DoD) acquisitions) and with 48 C.F.R. 2.101 and 12.212 (for non-DoD acquisitions).

REPORT

You may wish to report any ambiguities, inconsistencies or inaccuracies you may find in connection with your evaluation of the Specification ("Feedback"). To the extent that you provide Oracle with any Feedback, you hereby: (i) agree that such Feedback is provided on a non-proprietary and non-confidential basis, and (ii) grant Oracle a perpetual, non-exclusive, worldwide, fully paid-up, irrevocable license, with the right to sublicense through multiple levels of sublicensees, to incorporate, disclose, and use without limitation the Feedback for any purpose related to the Specification and future versions, implementations, and test suites thereof.

GENERAL TERMS

Any action related to this Agreement will be governed by California law and controlling U.S. federal law. The U.N. Convention for the International Sale of Goods and the choice of law rules of any jurisdiction will not apply.

The Specification is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Licensee agrees to comply strictly with all such laws and regulations and acknowledges that it has the responsibility to obtain such licenses to export, re-export or import as may be required after delivery to Licensee.

This Agreement is the parties' entire agreement relating to its subject matter. It supersedes all prior or contemporaneous oral or written communications, proposals, conditions, representations and warranties and prevails over any conflicting or additional terms of any quote, order, acknowledgment, or other communication between the parties relating to its subject matter during the term of this Agreement. No modification to this Agreement will be binding, unless in writing and signed by an authorized representative of each party.

Contents

1. Introduction

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 ensure delivery. This varies from simple best effort to guaranteed, only once delivery. Other important attributes are message time-to-live, priority and whether a response is required.

1.2.3. JMS Objectives

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

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

1.2.3.1. JMS Provider

As noted earlier, a JMS provider is the entity 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.

1.2.3.2. JMS Messages

JMS defines a set of message interfaces.

Clients use the message implementations supplied by their JMS provider.

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

1.2.3.3. JMS Domains

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

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

Publish and subscribe (Pub/Sub) clients address messages to some node in a content hierarchy. Publishers and subscribers are generally anonymous and may dynamically publish or subscribe to the content hierarchy. The

system takes care of distributing the messages arriving from a node's multiple publishers to its multiple subscribers.

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

1.2.3.4. Portability

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

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

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

1.2.4. What JMS Does Not Include

JMS does not address the following functionality:

- Load Balancing/Fault Tolerance - Many products provide support for multiple, cooperating clients implementing a critical service. The JMS API does not specify how such clients cooperate to appear to be a single, unified service.
- Error/Advisory Notification - Most messaging products define system messages that provide asynchronous notification of problems or system events to clients. JMS does not attempt to standardize these messages. By following the guidelines defined by JMS, clients can avoid using these messages and thus prevent the portability problems their use introduces.
- Administration - JMS does not define an API for administering messaging products.
- Security - JMS does not specify an API for controlling the privacy and integrity of messages. It also does not specify how digital signatures or keys are distributed to clients. Security is considered to be a JMS provider-specific feature that is configured by an administrator rather than controlled via the JMS API by clients.
- Wire Protocol - JMS does not define a wire protocol for messaging.
- Message Type Repository - JMS does not define a repository for storing message type definitions and it does not define a language for creating message type definitions.

1.3. What Is Required by JMS

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

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

1.4. Relationship to Other Java APIs

1.4.1. Java DataBase Connectivity (JDBC™) Software

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

1.4.2. JavaBeans™ Components

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

1.4.3. Enterprise JavaBeans™ Component Model

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

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

1.4.4. Java Transaction API (JTA)

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

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

A JMS provider can optionally support distributed transactions via JTA.

1.4.5. Java Transaction Service (JTS)

JMS can be used in conjunction with JTS to form distributed transactions that combine message sends and receives with database updates and other JTS aware services. 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 Interface™ (JNDI) API

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

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

1.4.7. Java Platform, Enterprise Edition (Java EE)

The Java™ Platform, Enterprise Edition (Java EE) Specification, v7 requires support for the JMS API as part of the Java EE platform. The Java EE platform specification places certain additional requirements on the implementation and use of the JMS API. The most important requirements are described in chapter 12 "Use of JMS API in Java EE applications".

1.5. What is New in JMS 1.1?

In previous versions of JMS, client programming for the Point-to-Point and Pub/Sub domains was done using similar but separate class hierarchies. In JMS 1.1, there is now a domain-independent approach to programming the client application. This provides several benefits:

- For the client programmer, a simpler programming model
- The ability to engage queues and topics in the same transaction, now that they can be created in the same session
- For the JMS provider, increased opportunity to optimize implementations by pooling thread management
- To take advantage of these features, the developer of JMS clients needs to use the domain-independent or “common” APIs. In the future, some of the domain-specific APIs may be deprecated.

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

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

1.6. What is new in JMS 2.0?

A full list of the new features, changes and clarifications introduced in JMS 2.0 is given in section B.5 "Version 2.0" of the "Change History" chapter. Here is a summary:

The JMS 2.0 specification now requires JMS providers to implement both P2P and Pub-Sub.

The following new messaging features have been added in JMS 2.0:

- Batch delivery: new API has been added to allow a JMS provider to deliver messages to an asynchsonous consumer in batches.
- Delivery delay: a message producer can now specify that a message must not be delivered until after a specified time interval.
- New send methods have been added to allow an application to send messages asynchronously.
- JMS providers must now set the `JMSXDeliveryCount` message property.

Several changes have been made to the JMS API to make it simpler and easier to use:

- `Connection`, `Session` and other objects with a `close()` method now implement the `java.jang.AutoCloseable` interface to allow them to be used in a Java SE 7 try-with-resources starement.
- A new "simplified API" has been added which offers a simpler alternative to the standard API, especially in Java EE applications.
- New methods have been added to create a session without the need to supply redundant arguments.
- Client ID is now optional when creating a durable subscription

A new chapter has been added which describes some additional restrictions and behaviour which apply when using the JMS API in the Java EE web or EJB container. This information was previously only available in the EJB and Java EE platform specifications.

New methods have been added to `Session` which return a `MessageConsumer` on a durable topic subscription. Applications could previously only obtain a domain-specific `TopicSubscriber`, even though its use was discouraged.

The specification has been clarified in various places.

2. Architecture

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.

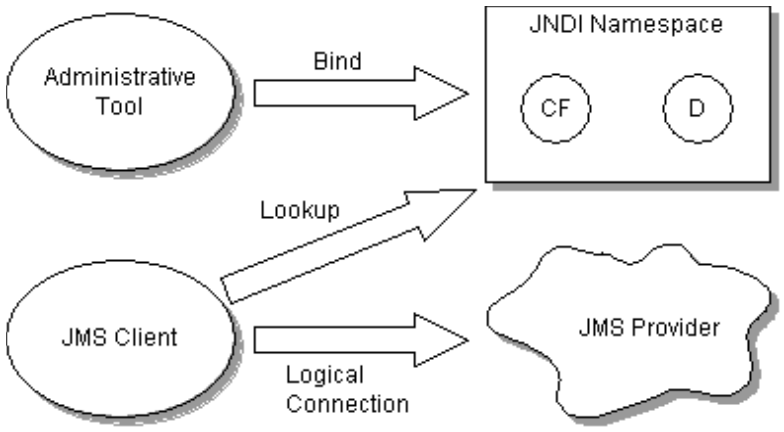
If JMS clients are to be portable, they must be isolated from these proprietary aspects of a provider. This is done by defining JMS administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. 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. Figure 2.1 illustrates how JMS administration ordinarily works.

Figure 2.1 JMS Administration



2.4. Two Messaging Styles

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

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

2.5. JMS Interfaces

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

Table 2.1 Relationship of PTP and Pub/Sub Interfaces

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

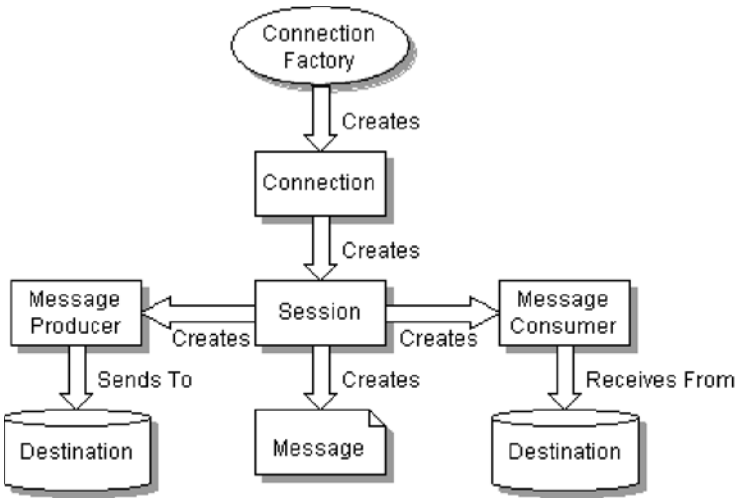
The JMS common interfaces provide a domain-independent view of the PTP and Pub/Sub messaging domains. JMS client programmers are encouraged to use these interfaces to create their client programs.

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

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

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

Figure 2.2 Overview of JMS object relationships



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

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

2.6. Developing a JMS Application

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

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

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

2.6.1. Developing a JMS Client

A typical JMS client executes the following JMS setup procedure:

- Use JNDI to find a ConnectionFactory object
- Use JNDI to find one or more Destination objects
- Use the ConnectionFactory to create a JMS Connection with message delivery inhibited
- Use the Connection to create one or more JMS Sessions
- Use a Session and the Destinations to create the MessageProducers and MessageConsumers needed
- Tell the Connection to start delivery of messages

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

2.7. Security

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

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

2.8. 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 2.2 JMS objects that support concurrent use

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

JMS defines some specific rules that restrict the concurrent use of Sessions. Since they require more knowledge of JMS specifics than we have presented at this point, they will be described later. Here we will describe the rationale for imposing them.

There are two reasons for restricting concurrent access to Sessions. First, Sessions are the JMS entity that supports transactions. It is very difficult to implement transactions that are 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.

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

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

3. JMS Message Model

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 IDs 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. If the JMS provider accepts this hint, these messages must have the message ID set to null; if the provider ignores the hint, the 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, the field 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. If the JMS provider accepts this hint, these messages must have the timestamp set to zero; if the provider ignores the hint, the 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 typical 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[1] The use of a *byte[]* value for *JMSCorrelationID* is non-portable.

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.

---

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

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 but not acknowledged in the past. In general, a provider must set the *JMSRedelivered* message header field of a message whenever it is redelivering a message. If the field is set to true, it is an indication to the consuming application that the message may have been delivered in the past and that the application should take extra precautions to prevent duplicate processing. See Section 4.4.11"Message Acknowledgment" for more information.

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

The JMS-defined message property JMSXDeliveryCount will be set to the number of times a particular message has been delivered. See section 3.5.11 "JMSXDeliveryCount" for more information.

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 definitions 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 the field is properly set for those providers that require it.

To ensure portability, JMS clients should use symbolic values for *JMSType* that can be configured at installation time to the values defined in the current provider's message repository. If string literals are used 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 value. 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 to indicate that 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 3.1 Message header field values

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

Message header fields that are defined as being set by "client" in the above table may be set by the client application before the message is sent using an appropriate setter method.

Message header fields that are defined as being set by "send method" or "provider" will be set by the JMS provider, either after it has been sent or before it has been delivered. It does this using appropriate setter methods. These methods are public to allow one JMS provider to set these fields when handling a message whose implementation is not its own. They are not for use by clients. Any values set by the client will be ignored and overwritten.

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.

3.4.13. JMSDeliveryTime

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

A message's delivery time is the earliest time when a provider may make the message visible on the target destination and available for delivery to consumers.

Clients must not receive messages before the delivery time has been reached.

3.5. Message Properties

In addition to the header fields defined here, the *Message* interface 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 ""), 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 "Message Selection" 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.

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. Attempting to read a null value as a Java primitive type must be treated as calling the primitive's corresponding *valueOf(String)* conversion method with a null value.

A value set as the row type can be read as the column type.

Table 3.2 Property value conversion

	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.

The *getPropertyNames* method does not return the names of the JMS standard header fields.

3.5.7. Clearing 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. New property entries can then be both created and read.

Clearing a message's property entries does not clear the value of its body.

JMS does not provide a way to remove an individual property entry once it has been added to a message.

3.5.8. Non-existent Properties

Getting a property value for a name which has not been set is handled as if the property exists with a null value.

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 in Table 3.3. This table defines:

- The name of the property
- The type of the property (integer or string)
- Whether support for the property is mandatory or optional.
- Whether the property is set by the sending client, by the provider when the message is sent, or by the provider when the message is received.
- The purpose of the property

New JMS defined properties may be added in later versions of JMS.

The *Enumeration 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. If they are not present in a message, they are treated like any other absent property. The effect of setting a message selector on a property which is set by the provider on receive is undefined.

The existence, in a particular message, of optional JMS defined properties that are set by a JMS Provider depends on how a particular provider controls use of the property. It may choose to include them in some messages and omit them in others depending on administrative or other criteria.

Table 3.3 JMS defined properties

Name	**Type**	**Optional or mandatory**	**Set By**	**Use**
JMSXUserID	String	Optional	Provider on Send	The identity of the user sending the message
JMSXAppID	String	Optional	Provider on Send	The identity of the application sending the message
JMSXDeliveryCount	int	Mandatory	Provider on Receive	The number of message delivery attempts. See section 3.5.11 "JMSXDeliveryCount".
JMSXGroupID	String	Optional	Client	The identity of the message group this message is part of
JMSXGroupSeq	int	Optional	Client	The sequence number of this message within the group; the first message is 1, the second 2,...
JMSXProducerTXID	String	Optional	Provider on Send	The transaction identifier of the transaction within which this message was produced
JMSXConsumerTXID	String	Optional	Provider on Receive	The transaction identifier of the transaction within which this message was consumed

Name	Type	Optional or mandatory	Set By	Use
JMSXRcvTimestamp	long	Optional	Provider on Receive	The time JMS delivered the message to the consumer
JMSXState	int	Optional	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(waiting), 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.

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

The case of these JMSX property names must be as defined in the table above.

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.

3.5.11. JMSXDeliveryCount

When a client receives a message the mandatory JMS-defined message property JMSXDeliveryCount will be set to the number of times the message has been delivered. The first time a message is received it will be set to 1, so a value of 2 or more means the message has been redelivered.

If the JMSRedelivered message header value is set then the JMSXDeliveryCount property must always be 2 or more. See section 3.4.7 "JMSRedelivered" for more information about the JMSRedelivered message header,

The purpose of the JMSXDeliveryCount property is to allow consuming applications to identify whether a particular message is being repeatedly redelivered and take appropriate action.

The value of the JMSXDeliveryCount property is not guaranteed to be exactly correct. The JMS provider is not expected to persist this value to ensure that its value is not lost in the event of a failure.

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 4.4.11 "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 and properties 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[2] conditional expression syntax.

If the value of a message selector is an empty string, the value is treated as a null and indicates that there is no message selector for the message consumer.

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; for example, 'literal' and 'literal''s'. Like Java String literals, these use the Unicode character encoding.
  - 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. Exact numeric literals use the Java integer literal syntax.
  - An approximate numeric literal is a numeric value in scientific notation, such as 7E3 and -57.9E2, or a numeric value with a decimal, such as 7., -95.7, and +6.2; numbers in the range of Java double are supported. Approximate literals use the Java floating-point literal syntax.
  - The boolean literals TRUE and FALSE.
- Identifiers:
  - An identifier is an unlimited-length character sequence that must begin with a Java identifier start character; 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 identifier 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*, *IS*, or *ESCAPE*.
  - Identifiers are either header field references or property references. The type of a property value in a message selector corresponds to the type used to set the property. If a property that does not exist in a message is referenced, its value is NULL. The semantics of evaluating NULL values in a selector are described in Section 3.8.1.2 "Null Values".

---

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

  - The conversions that apply to the get methods for properties do not apply when a property is used in a message selector expression. For example, suppose you set a property as a string value, as in the following:

    ```
    myMessage.setStringProperty("NumberOfOrders", "2");
    ```

    The following expression in a message selector would evaluate to false, because a string cannot be used in an arithmetic expression:

    ```
    "NumberOfOrders > 1"
    ```

  - Identifiers are case sensitive.
  - Message header field references are restricted to JMSDeliveryMode, JMSPriority, JMSMessageID, JMSTimestamp, JMSCorrelationID, and JMSType. JMSMessageID, 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.
- 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, arithmeticoperations, 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. One exception is that it is valid to compare exact numeric values and approximate numeric values (the type conversion required is defined by the rules of Java numeric promotion). If the comparison of non-like type values is attempted, the value of the operation is false. If either of the type values evaluates to NULL, the value of the expression is unknown.

  - `String` and `Boolean` 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’ or ‘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 that tests for a null header field value or a missing property value)
  - “prop_name IS NULL”

- *identifier* IS NOT NULL (comparison operator that 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*. JMS providers may also optionally provide some semantic checking at the time the selector is presented. Not all semantic checking can be performed at the time a message selector is presented, because property types are not known.

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 3.4 The definition of the AND operator

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

Table 3.5 The definition of the OR operator

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

Table 3.6 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 is included 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. Changing the Value of a Received Message

When a message is received, its header field values can be changed; however, its property entries and its body are read-only, as specified in this chapter.

The rationale for the read-only restriction is that it gives JMS Providers more freedom in how they implement the management of received messages. For instance, they may return a message object that references property entries and body values that reside in an internal message buffer rather than being forced to make a copy.

A consumer can modify a received message after calling either the *clearBody* or *clearProperties* method to make the body or properties writable. If the consumer modifies a received message, and the message is subsequently redelivered, the redelivered message must be the original, unmodified message (except for headers and properties modified by the JMS provider as a result of the redelivery, such as the *JMSRedelivered* header and the *JMSXDeliveryCount* property).

3.11. 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.*

3.11.1. Clearing a Message Body

The *clearBody* method of *Message* resets the value of the message body to the 'empty' initial message value as set by the message type's create method provided by *Session*. Clearing a message's body does not clear its property entries.

3.11.2. Read-Only Message Body

When a message is received, its body is read only. If an attempt is made to change the body a *MessageNotWriteableException* must be thrown. If its body is subsequently cleared, the body is in the same state as an empty body in a newly created message.

3.11.3. 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.

StreamMessage and *MapMessage* must implement the *String* to boolean conversion as specified by the *valueOf(String)* method of *Boolean* as defined by the Java language.

Attempting to read a null value as a Java primitive type must be treated as calling the primitive's corresponding *valueOf(String)* conversion method with a null value. Since *char* does not support a *String* conversion, attempting to read a null value as a *char* must throw *NullPointerException*.

Getting a *MapMessage* field for a field name that has not been set is handled as if the field exists with a null value.

If a read method of *StreamMessage* or *BytesMessage* throws a *MessageFormatException* or *NumberFormatException*, the current position of the read pointer must not be incremented. A subsequent read must be capable of recovering from the exception by rereading the data as a different type.

A value written as the row type can be read as the column type

Table 3.7 Conversions for StreamMessage and MapMessage

	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	
byte[]										X

3.11.4. 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 ensure 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*.

3.12. 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, it must be handled.

The JMS message interfaces provide write/set methods for setting object values in a message body and message properties. All of these methods must be implemented to copy their input objects into the message. The value of an input object is allowed to be null and will return null when accessed. One exception to this is that *BytesMessage* does not support the concept of a null stream and attempting to write a null into it must throw *java.lang.NullPointerException*.

The JMS message interfaces provide read/get methods for accessing objects in a message body and message properties. All of these methods must be implemented to return a copy of the accessed message objects.

4. JMS Common Facilities

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 effects 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 Destination Objects" 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 authentication takes place.
- It can specify a unique client identifier.
- It creates *Session* objects.
- 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 heavyweight 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 its 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 facility to explicitly set a connection's client identifier is not a mechanism for overriding the identifier that has been administratively configured. It is provided for the case where no administratively specified identifier exists. If one does exist, an attempt to change it by setting it must throw a *IllegalStateException*.

An application may explicitly set a connection's client identifier by calling the `setClientID` method on the `Connection` object or, if the simplified API is being used, on the `MessagingContext`.

If a client explicitly sets a connection's client identifier it must do so immediately after creating the connection or messaging context and before any other action on the connection or messaging context is taken. After this point, setting the client identifier is a programming error that should throw an *IllegalStateException*.

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. By definition, the client state identified by a client identifier can only be 'in use' by only one client at a time. A JMS provider must prevent concurrently executing clients from using it.

This prevention may take the form of JMSExceptions thrown when such use is attempted; it may result in the offending client being blocked; or some other solution. A JMS provider must insure that such attempted 'sharing' of an individual client state does not result in messages being lost or doubly processed.

The only use of a client identifier defined by JMS is its optional use in identifying a durable subscription.

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 be started immediately 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.

It is important to note that clients rely on the fact that no messages are delivered by a connection until it has been started. JMS Providers must ensure that this is the case.

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 receives block, and messages are not delivered to any message listeners (*MessageListeners* or `BatchMessageListeners`).

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

A *stop* method call must not return until delivery of messages has paused. This means a client can rely on the fact that none of its message listeners will be called and all threads of control waiting for receive to return will not return with a message until the connection is restarted. The receive timers for a stopped connection continue to advance so receives may time out and return a null message while the connection is stopped.

If any message listeners are running when stop is invoked, stop must wait until all of them have returned before it may return. While these message listeners are completing, they must have the full services of the connection available to them.

A message listener must not attempt to stop its own connection as this would lead to deadlock. The JMS provider must detect this and throw a `javax.jms.IllegalStateException`.

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.

A close terminates all pending message receives on the connection's session's consumers. The receives may return with a message or null depending on whether there was a message or not available at the time of the close.

Note that in this case, the message consumer will likely get an exception if it is attempting to use the facilities of the now closed connection while processing its last message. A developer must take this 'last message' case into account when writing a message consumer. It bears repeating that the message consumer cannot rely on a null return value to indicate this 'last message' case.

If one or more of the connection's session's message listeners is processing a message at the point when connection close is invoked, all the facilities of the connection and its sessions must remain available to those listeners until they return control to the JMS provider.

When connection close is invoked it should not return until message processing has been shut down in an orderly fashion. This means that all message listeners that may have been running have returned, and that all pending receives have returned.

A message listener must not attempt to close its own connection as this would lead to deadlock. The JMS provider must detect this and throw a `javax.jms.IllegalStateException`.

If a connection is closed, there is no need to close its constituent sessions, message producers, messages consumers or queue browsers. The connection close is sufficient to signal the JMS provider that all resources for the connection should be released.

The `Connection` interface extends the `java.lang.AutoCloseable` interface. This means that applications which create the connection in a `try-with-resources` statement do not need to call the `close` method when the connection is no longer needed. Instead the connection will be closed automatically at the end of the statement. The use of a `try-with-resources` statement also simplifies the handling of any exceptions thrown by the `close` method. See the Java Tutorials[3] for more information about the `try-with-resources` statement.

Closing a connection must rollback the transactions in progress on its transacted sessions[4]. Closing a connection does NOT force an acknowledge of client acknowledged sessions. Invoking the *acknowledge* method of a received message from a closed connection's sessions must throw an *IllegalStateException*. These semantics insure that closing a connection does not cause messages to be lost for queues and durable subscriptions which require reliable processing by a subsequent execution of their JMS client.

Once a connection has been closed, an attempt to use it or its sessions or their message consumers and producers must throw an *IllegalStateException* (calls to the *close* method of these objects must be ignored). It is valid to continue to use message objects created or received via the connection with the exception of a received message's *acknowledge* method.

Closing a closed connection must NOT throw an exception.

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 provider's product name and version.

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

---

[3] The Java Tutorials may be found at http://docs.oracle.com/javase/tutorial/index.html.

[4] The term 'transacted session' refers to the case where a session's commit and rollback methods are used to demarcate a transaction local to the session. In the case where a session's work is coordinated by an external transaction manager, a session's commit and rollback methods are not used and the result of a closed session's work is determined later by the transaction manager.

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. To retrieve an ExceptionListener, the JMS provider calls the connection's *getExceptionListerer()* method. This method returns the *ExceptionListener* for the connection. If no *ExceptionListener* is registered, the value null is returned. The connection can then use the listener 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*.This means that if a connection encounters multiple problems and therefore needs to call its `ExceptionListener` multiple times, then it will only invoke `onMessage` from one thread at a time. However if the same `ExceptionListener` is registered with multiple connections then it is undefined whether these connections could call `onMessage` from different threads simultaneously.

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

The exceptions delivered to *ExceptionListener* are those that have no other place to be reported. If an exception is thrown on a JMS call it, by definition, must not be delivered to an *ExceptionListener* (in other words, *ExceptionListener* is not for the purpose of monitoring all exceptions thrown by a connection).

There is no restriction on the use of the JMS API by the listener's `onException` method. However since that method will only be called when there is a serious problem with the connection, any attempt to use that connection may fail and cause exceptions.

4.4. Session

A JMS *Session* is a single threaded context[5] 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 *MessageProducer*s and *MessageConsumer*s.
- It is a factory for *TemporaryTopics* and *TemporaryQueues*.

[5] 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 ensure 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.

- It provides a way to create *Queue* or *Topic* 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.
- It defines a serial order for the messages it consumes and the messages it produces.
- It retains messages it consumes until they have been acknowledged.
- It serializes execution of *MessageListener* or `BatchMessageListener` objects registered with it.
- It is a factory for *QueueBrowsers*.

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.

Session close terminates all message processing on the session. It must handle the shutdown of pending receives by the session's consumers or a running message listener as described in Closing a Connection.

Session close is the only session method that may be invoked from a thread of control separate from the one which is currently controlling the session.

When session close is invoked it should not return until its message processing has been shut down in an orderly fashion. This means that none of its message listeners are running, and that if there is a pending receive, it has returned with either null or a message.

A message listener must not attempt to close its own session as this would lead to deadlock. The JMS provider must detect this and throw a `javax.jms.IllegalStateException`.

When a session is closed, there is no need to close its constituent message producers, message consumers or queue browsers. The session close is sufficient to signal the JMS provider that all resources for the session should be released.

Note that closing a connection will cause any sessions created from it to be closed, so, although a session should be closed when no longer needed, there is no need to close a session immediately prior to closing its connection.

The `Session` interface extends the `java.lang.AutoCloseable` interface. This means that applications which create the `session` in a `try-with-resources` statement do not need to call the `close` method when the connection is no longer needed. Instead the session will be closed automatically at the end of the statement. The use of a `try-with-resources` statement also simplifies the handling of any exceptions thrown by the `close` method.

Closing a transacted session must rollback its transaction in progress. Closing a client-acknowledged session does NOT force an acknowledge.

Once a session has been closed, an attempt to use it or its message consumers and producers must throw an *IllegalStateException* (calls to the *close* method of these objects must be ignored). It is valid to continue to use message objects created or received via the session with the exception of a received message's *acknowledge* method.

Closing a closed session must NOT throw an exception.

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 information on their creation and use.

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 Temporary Destinations

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.

Temporary destinations (*TemporaryQueue* or *TemporaryTopic* objects) are destinations that are system-generated uniquely for their connection. Only their own connection is allowed to create *MessageConsumer*s for them.

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

Each *TemporaryQueue* or *TemporaryTopic* object object is unique. It cannot be copied.

Since temporary destinations 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 Destination Objects

Most clients will use *Destinations* that are JMS administered objects that they have looked up 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 provider-specific method for doing this.

4.4.5. Optimized Message Implementations

A session provides the following methods to create messages: `createMessage`, `createBytesMessage`, `createMapMessage`, `createObjectMessage`, `createStreamMessage` and `createTextMessage`.

These methods allow the JMS provider to create message implementations which are optimized for that particular provider and allow the provider to minimize its overhead for handling messages.

However the fact that these methods are provided on a session does not mean that messages must be sent using a message producer created from the same session. Messages may be sent using any session, not just the session used to create the message.

Furthermore, sessions must be capable of sending all JMS messages regardless of how they may be implemented. See section 3.12 "Provider Implementations of JMS Message Interfaces".

4.4.6. Conventions for Using a Session

Sessions are designed for serial use by one thread at a time. The only exception to this occurs during the orderly shutdown of the session or its connection. See Section 4.3.5 "Closing a Connection" and Section 4.4.1 "Closing a Session" for further details.

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*.

It is erroneous for a client to use a thread of control to attempt to synchronously receive a message if there is already a client thread of control waiting to receive a message in the same session.

Another typical use is to have one thread set up a session by creating its producers and one or more asynchronous consumers. In this case, the message producers are exclusively for the use of the consumer's message listeners. Since the session serializes execution of its consumers' message listeners (MessageListeners or BatchMessageListeners), 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 ensure that a session's *MessageProducers*, *MessageConsumers* and message listeners *(MessageListeners or BatchMessageListeners)* 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 in which *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.

Once a connection has been started, all its sessions with a registered message listener are dedicated to the thread of control that delivers messages to them. It is erroneous for client code to use such a session from another thread of control. The only exception to this is the use of the session or connection close method.

One consequence of the session's single-thread-of-control restriction is that a session with message listeners cannot also be used to synchronously receive messages. Either the session is dedicated to the thread of control

used for delivery to message listeners, or it is dedicated to a thread of control initiated by client code. It is erroneous to attempt to combine both in the same session.

Another consequence is that a connection must be in stopped mode to set up a session with more than one message listener. The reason is that when a connection is actively delivering messages, once the first message listener for a session has been registered, the session is now controlled by the thread of control that delivers messages to it. At this point a client thread of control cannot be used to further configure the session.

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.

Since a MessagingContext incorporates a Session it is subject to the same threading restrictions as a Session. For more information, and an exception to this, see section 13.2.8 "Threading restrictions on a MessagingContext".

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 4.4.11 "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 recommended that JMS clients avoid doing this. JMS clients that use the XA-based interfaces described in Chapter 8 "JMS Application Server Facilities" may not be portable across different JMS implementations, because these interfaces are optional. 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 *TopicSubscriber* 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. Therefore, message delivery to multiple *QueueReceivers* will depend on the JMS provider's implementation. Applications that depend on delivery to multiple *QueueReceivers* are not portable

4.4.10. Message Order

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

4.4.10.1. 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 4.4.11 "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 in which they were sent (see Section 4.4.10.2 "Order of Message Sends" for a few qualifications). This defines a partial ordering constraint on a session's input message stream.

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.

4.4.10.2. 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 previous lower-priority messages.
- Messages with a later delivery time may be delivered after messages with an earlier delivery time.
- 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 with the same priority.

- 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.

4.4.11. 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 those that were originally delivered due to message expiration, the arrival of higher-priority messages, or the delivery of messages which could not previously be delivered as they had not reached their specified delivery time.

A session must set the *JMSRedelivered* header and increment the `JMSXDeliveryCount` property of messages it redelivers due to a recovery

4.4.12. 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 redelivery 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. The *JMSRedelivered* message header field will be set for a message redelivered under these circumstances, and the JMSXDeliveryCount property will be incremented.

4.4.13. 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 ensure that consumers of the message will receive it only 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.

4.4.14. Serial Execution of Client Code

Even though the Java language provides built-in support for multithreading, writing multithreaded programs is still more difficult than writing singlethreaded 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 or BatchMessageListener* interface with a *MessageConsumer*. *In effect, a Session uses a single thread to run all its message listeners*. While the thread is busy executing one listener, all other messages to be asynchronously delivered to the session must wait.

4.4.15. Concurrent Message Delivery

Clients that 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.

Note that JMS itself does not provide the facilities for concurrently processing a topic's message set (the messages delivered to a single consumer). A client could use a single consumer and implement all the multithreading logic needed to concurrently process the messages; however, it is not possible to do this reliably, because JMS does not have the transaction facilities needed to handle the concurrent transactions this would require.

4.5. MessageConsumer

A client uses a *MessageConsumer* to receive messages from a destination. A *MessageConsumer* is created by passing a *Queue* or *Topic* to a *Session's createConsumer* 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.

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.

A client may alternatively register an object that implements the JMS `BatchMessageListener` interface with a `MessageConsumer`. Messages that arrive for the consumer will be delivered in batches by calling the listener's `onMessages` method. See section 4.12 "Batch delivery" for more information.

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 or DUPS_OK_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. The *JMSRedelivered* message header field will be set, and the `JMSXDeliveryCount` message property incremented, for a message redelivered under these circumstances.
- 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.
- Transacted Session - the next message for the listener is delivered. The client can either commit or roll back the session (in other words, 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 4.4.14 "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*. A *MessageProducer* is created by passing a *Queue* or *Topic* to a session's *createProducer* 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, time-to-live and delivery delay for messages sent by a producer.

A client can also specify delivery mode, priority, and time-to-live per message.

The following methods on `MessageProducer` may be used to send a message:

```
void send(Message message)

void send(Destination destination, Message message)

void send(Destination destination, Message message, int
deliveryMode, int priority, long timeToLive)

void send(Message message, int deliveryMode, int
priority, long timeToLive)
```

These methods will block until the message has been sent. If necessary the call will block until a confirmation message has been received back from the JMS server.

Clients may alternatively use the following methods which return immediately and use a separate thread to send the message and, if necessary, wait for a confirmation message back from the JMS server:

```
send(Destination destination, Message message,
CompletionListener completionListener)

send(Destination destination, Message message, int
deliveryMode, int priority, long timeToLive,
CompletionListener completionListener)

send(Message message, CompletionListener
completionListener)

send(Message message, int deliveryMode, int priority,
long timeToLive, CompletionListener completionListener)
```

When the sending of the message is complete, and any confirmation has been received from a JMS server, the JMS provider calls the `onCompletion(Message)` method of the specified completion listener. If an exception occurs in the separate thread then the JMS provider calls the `onException(Exception)` method of the specified completion listener.

JMS does not define what operations are performed in the calling thread and what operations, if any, are performed in the separate thread. In particular the use of this method does not itself specify whether the separate thread should obtain confirmation from a JMS server.

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. See Section 3.4.13 "JMSDeliveryTime" for more information on delivery delay.

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 take extra care to ensure the message is not lost in transit due to a JMS provider failure.

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

A JMS provider must deliver a PERSISTENT message *once-and-only-once*. This means a JMS provider failure must not cause it to be lost, and it must not deliver it twice.

PERSISTENT (once-and-only-once) and NON_PERSISTENT (at-most-once) message delivery are a way for a JMS client to select between delivery techniques that may lose a messages if a JMS provider dies and those which take extra effort to ensure that messages can survive such a failure. There is typically a performance/reliability trade-off implied by this choice. When a client selects the NON_PERSISTENT delivery mode, it is indicating that it values performance over reliability; a selection of PERSISTENT reverses the requested trade-off.

The use of PERSISTENT messages does not guarantee that all messages are always delivered to every eligible consumer. See Section 4.10 "Reliability" for further discussion on this topic.

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 for messages delivered from a queue or a durable subscription.

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 typically have lower overhead;however, they may be lost if a JMS provider 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 session. Unacknowledged messages redelivered due to system failure must have the *JMSRedelivered* message header field set, and the JMSXDeliveryCount incremented, by the JMS provider, as described in sections 3.4.7 "JMSRedelivered" and 3.5.11 "JMSXDeliveryCount"

If a NON_PERSISTENT message is delivered to a durable subscription or a queue, delivery is not guaranteed if the durable subscription becomes inactive (that is, if it has no current subscriber) or if the JMS provider is shut down and later restarted.

It is expected that important messages will be produced with a PERSISTENT delivery mode within a transaction and will be consumed within a transaction from a nontemporary queue or a durable subscription.

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 destinations 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 that adequate resources to service them are available.

Once-and-only-once message delivery, as described in this specification, has the important caveat that it does not cover message destruction due to message expiration or other administrative destruction criteria. It also does not cover loss due to resource restrictions. Configuration of adequate resources and processing power for JMS applications is the job of administrators, who must be aware of their JMS provider's reliability features.

NON_PERSISTENT messages, nondurable subscriptions, and temporary destinations are by definition unreliable. A JMS provider shutdown or

failure will likely cause the loss of NON_PERSISTENT messages and the loss of messages held by temporary destinations and nondurable subscriptions. The termination of an application will likely cause the loss of messages held by nondurable subscriptions and temporary destinations of the application.

4.11. Method Inheritance across Messaging Domains

As a result of unifying the domains, some methods that are not appropriate to a domain may be inherited in the domain-specific classes. For example, the *Session* interface has the method *createQueueBrowser.* Since *TopicSession* inherits from the *Session* interface, *TopicSession* inherits the *createQueueBrowser* method, though that method must not be used by a topic, as topics do not support *QueueBrowsers*. Table 4.1outlines these instances.

If a application attempts to call any of the methods listed, the JMS provider must throw an *IllegalStateException*.

Table 4.1 methods that throw an IllegalStateException

Interface	**Method**
QueueConnection	createDurableConnectionConsumer
QueueSession	createDurableSubscriber
	createTemporaryTopic
	createTopic
	unsubscribe
TopicSession	createQueueBrowser
	createQueue
	createTemporaryQueue

4.12. Batch delivery

For asynchronous delivery, a client can register a single listener object with a message consumer which may be either a `MessageListener` or a `BatchMessageListener` object.

A `MessageListener` is used to receive messages individually. As messages arrive at the message consumer, it delivers them by calling the `MessageListener`'s `onMessage` method.

A `BatchMessageListener` is used to receive messages in batches. Messages which arrive at the message consumer will be delivered in batches by calling the `BatchMessageListener`'s `onMessages` method.

When a `BatchMessageListener` is used the client is required to specify a maximum batch size and a batch timeout. Messages will be delivered to the specified `BatchMessageListener` in batches whose size is no greater than the specified maximum batch size. The JMS provider may defer message delivery until the specified batch timeout has expired in order to assemble a batch of messages that is as large as possible but no greater than the batch size.

4.13. Delivery delay

A client can specify a delivery delay value in milliseconds for each message it sends. This value defines a message delivery time which is the sum of the message’s delivery delay 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 message's delivery time is the earliest time when a JMS provider may make the message visible on the target destination and available for delivery to consumers. The provider must not deliver messages before the delivery time has been reached.

For more information on message delivery delay, see Section 3.4.13 "JMSDeliveryTime".

5. JMS Point-to-Point Model

5.1. Overview

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

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

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

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

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

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

Table 5.1 shows the interfaces that are specific to the PTP domain and the JMS common interfaces. The common interfaces are preferred for creating JMS application programs, because they are domain-independent.

Table 5.1 PTP Domain Interfaces and JMS Common Interfaces

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

5.2. Queue Management

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

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

5.5. QueueConnectionFactory

A client uses a *QueueConnectionFactory* to create *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 4.2 "Administered Objects" for more information.

5.7. QueueSession

A *QueueSession* provides methods for creating *QueueReceivers*, *QueueSenders*, *QueueBrowsers* 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 4.3 "Connection" 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. By definition, a message selector allows a QueueReceiver to skip messages. This means that when the skipped messages are eventually read, the total ordering of the reads does not

retain the partial order defined by each message producer. Only *QueueReceivers* without a message selector will read messages in message producer order.

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

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

For more information, see Section 4.6 "MessageProducer".

5.9. QueueBrowser

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

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

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

5.10. QueueRequestor

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

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

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

5.11. Reliability

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

6. JMS Publish/Subscribe Model

6.1. Overview

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

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

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

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

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

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

Table 6.1shows the interfaces that are specific to the PTP domain and the JMS common interfaces. The common interfaces are preferred for creating JMS application programs, because they are domain-independent.

Table 6.1 Pub/Sub Domain Interfaces and JMS Common Interfaces

PTP Domain Interfaces	**JMS Common Interfaces *Preferred***
TopicConnectionFactory	ConnectionFactory
TopicConnection	Connection
Topic	Destination
TopicSession	Session
TopicPublisher	MessageProducer
TopicSubscriber	MessageConsumer

6.2. Pub/Sub Latency

Since there is typically some latency in all pub/sub systems, the exact messages seen by a subscriber may vary depending on how quickly a JMS

provider propagates the existence of a new subscriber and the length of time a provider retains messages in transit.

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

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

6.3. Durable Subscription

Nondurable subscriptions last for the lifetime of their subscriber object. This means that a client will only see the messages published on a topic while its subscriber is active. If the subscriber is not active, it is missing messages published on its topic.

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

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

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

See also section 6.11.1 "Durable TopicSubscriber".

6.4. Topic Management

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

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

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

6.5. Topic

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

Many Pub/Sub providers group topics into hierarchies and provide various options for subscribing to parts of the hierarchy. JMS places no 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 *Connection* or *TopicConnection*. It is a system defined *Topic* that can only be consumed only by the *Connection* or *TopicConnection* that created it.

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

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

6.7. TopicConnectionFactory

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

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

6.8. TopicConnection

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

See Section 4.3 "Connection" for more information.

6.9. TopicSession

A *TopicSession* provides methods for creating *TopicPublishers*, *TopicSubscribers* and *TemporaryTopics*. It also provides the *unsubscribe* method for deleting its client's durable subscriptions.

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

See Section 4.4 "Session" for more information.

6.10. TopicPublisher

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

6.11. TopicSubscriber

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

Ordinary *TopicSubscribers* 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. When a non-durable subscription is created on a topic, the `noLocal` argument may be used to specify that the subscriber must not receive messages published to the topic by its own connection.

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

6.11.1. Durable TopicSubscriber

A durable subscription is used by a client which needs to receive all the messages published on a topic, including the ones published when there is no `MessageConsumer` or `TopicSubscriber` associated with it. The JMS provider retains a record of this durable subscription and ensures that all messages from the topic's publishers are retained until they are delivered to, and acknowledged by, a `MessageConsumer` or `TopicSubscriber` on the durable subscription or until they have expired.

A durable subscription is created, and a `MessageConsumer` created on that durable subscription, using the `createDurableConsumer` methods on a `Session` or `TopicSession`. The same methods may be used to create a `MessageConsumer` on an existing durable subscription.

TODO: Need to mention MessagingContext.subscribe()

There are also `createDurableSubscriber` methods which have the same behavior as `createDurableConsumer` but which return a `TopicSubscriber`. These methods are deprecated since the object they return is deprecated.

A durable subscription which has a `MessageConsumer` or `TopicSubscriber` associated with it is described as being active. A durable subscription which has no `MessageConsumer` or `TopicSubscriber` associated with it is described as being inactive.

Only one session at a time can have a `MessageConsumer` or `TopicSubscriber` for a particular durable subscription.

A durable subscription is identified by a name specified by the client and by the client identifier if set. If the client identifier was set when the durable subscription was first created then a client which subsequently

wishes to create a `MessageConsumer` or `TopicSubscriber` on that durable subscription must use the same client identifier.

A client can change an existing durable subscription by calling one of the `createDurableConsumer` methods with the same name and client identifier (if used), and a new topic and/or message selector. Changing a durable subscriber is equivalent to unsubscribing (deleting) the old one and creating a new one. A durable subscription will continue to accumulate messages until it is deleted using the `unsubscribe` method on the *Session* or *TopicSession*. This deletes the state being maintained on behalf of the subscriber by its provider. It is erroneous for a client to delete a durable subscription while it has an active `MessageConsumer` or `TopicSubscriber` for it or while a message received by it is part of a current transaction or has not been acknowledged in the session.

When a durable subscription is created on a topic, the `noLocal` argument may be used to specify that messages published to the topic by its own connection must not be added to the durable subscription.

See also section 6.3 "Durable Subscription" and section 4.3.2 "Client Identifier".

6.12. Recovery and Redelivery

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

Only durable subscriptions are reliably able to recover unacknowledged messages.

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

6.13. Administering Subscriptions

Ideally, publishers and subscribers are dynamically registered by a provider when they are created. From the client viewpoint this is always the case. From the administrator’s viewpoint, other tasks may be needed to support the creation of publishers and subscribers.

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

6.14. TopicRequestor

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

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

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

6.15. Reliability

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

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

Table 6.2 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

7. JMS Exceptions

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 class 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 **must be** thrown if *Session.commit()* is called on a non-transacted session. This exception is also **must be** called when domain inappropriate method is called, such as calling *TopicSession.CreateQueueBrowser()*.
- *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 identifier 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.writeObject()* is given an
- unsupported class or if *StreamMessage.readShort()* 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 must 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 thrown 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.

8. JMS Application Server Facilities

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 are primarily intended for the use of the JMS provider.

If JMS clients use the JTS aware facilitiest client program may be non-portable code, because JMS providers are not required to support these interfaces.

The facilities described in this chapter are a special category of JMS. They are optional and might only be supported by some 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 message listener class (either a `MessageListener` or a `BatchMessageListener`) to consume its messages. An application server will construct multiple objects of this class to concurrently consume messages.

8.2.1. Session

Sessions provide the following methods for use by application servers:

- *setMessageListener(), setBatchMessageListener(), getMessageListener() and getBatchMessageListener* - a session's message listener (either a `MessageListener` or a `BatchMessageListener`) consumes messages that have been assigned to the session by a *ConnectionConsumer*, as described in the next few paragraphs.
- *run()* - causes the messages assigned to its session by a *ConnectionConsumer* to be serially processed by the session's message listener. When the listener returns from processing the last message, *run()* returns.

An application server would typically be given a *MessageListener* or `BatchMessageListener` 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 and, in the case of batch delivery, the required maximum batch size and batch timeout.

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 message listener 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:

1. It gets a `ServerSession` from the its `ServerSessionPool`

2. It gets the `ServerSession's Session`

3. It loads the `Session` with one or more messages

4. It then starts the `ServerSession` to consume these messages

A *ConnectionConsumer* for a *QueueConnection* will expect to load its messages into a *QueueSession*, as one for a *TopicConnection* would 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 Application Server Implements a ServerSession

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

1. 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.

2. The *ServerSession's start* method calls its *Thread's start* method. As with 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.

3. 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

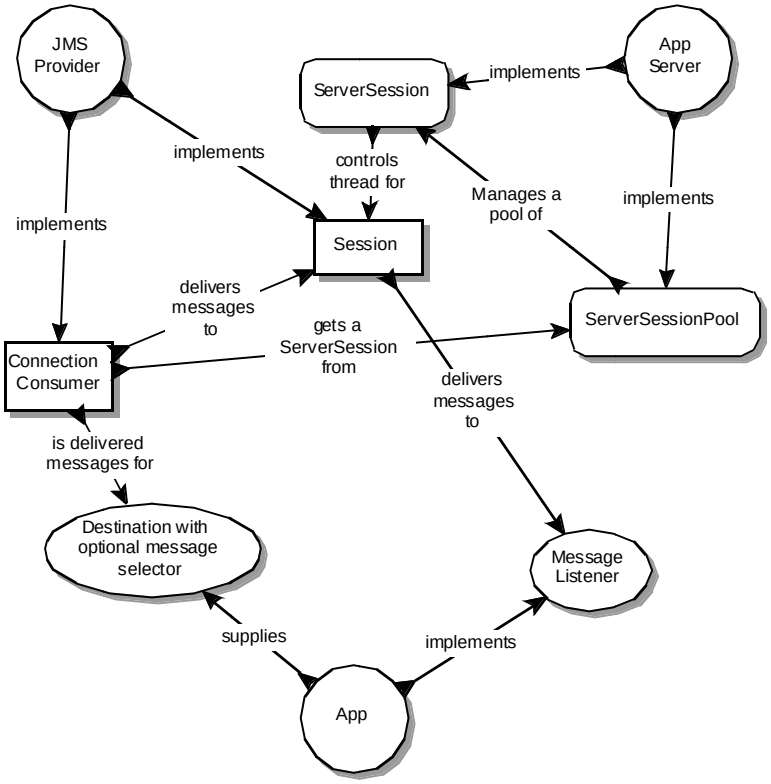
JMS has defined a flexible mechanism that partitions the job of concurrent message consumption into roles that are well suited for each participant.

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

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

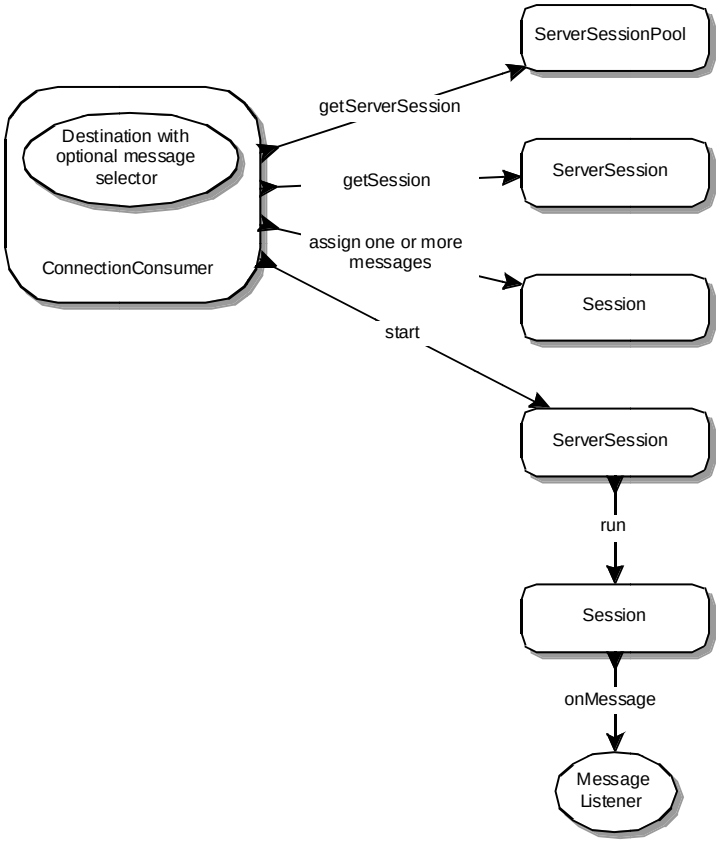
The application server is in control of setting up the *ConnectionConsumer* and managing all the threads used for executing its message listeners.

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



The diagram above shows a `MessageListener`. It could equally be a `BatchMessageListener`.

The following diagram illustrates the process a *ConnectionConsumer* uses to deliver a message to a *MessageListener*. It could equally be a `BatchMessageListener`.



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* objects. 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, and so on.

An *XAResource* provides some fairly sophisticated facilities for interleaving work on multiple transactions, recovering a list of transactions in progress, and so on. 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; that is, 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 provider's 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.

However, there are common interfaces available, should be used in preference to the domain-specific interfaces. These are listed as JMS common interfaces in Table 8.1.

Table 8.1 Relationship of optional interfaces in domains

JMS Common Interfaces	**PTP Interfaces**	**Pub/Sub Interfaces**
ServerSessionPool	*Not domain-specific*	*Not domain-specific*
ServerSession	*Not domain-specific*	*Not domain-specific*
ConnectionConsumer	*Not domain-specific*	*Not domain-specific*
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

9. JMS Example Code

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

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

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

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

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

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

9.1. Preparing to Send and Receive Messages

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

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

9.1.1. Getting a ConnectionFactory

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

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

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

ConnectionFactory connectionFactory;

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

9.1.2. Getting a Destination

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

```
Queue stockQueue;

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

9.1.3. Creating a Connection

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

```
Connection connection;

connection = connectionFactory.createConnection();
```

A `Connection` must be closed after use. This may be done explicitly using the close method:

```
connection.close();
```

Alternatively a connection may be closed automatically using the try-with-resources statement:

```
try (Connection connection =
        connectionFactory.createConnection();){
   // use connection in this try block
   // it will be closed when try block completes
} catch (JMSException e){
   // exception handling
}
```

9.1.4. Creating a Session

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

There are three `createSession` methods on `Connection`, with different numbers of arguments. Java SE applicatrions such as this example should use the method with one integer argument, `sessionMode`. This single argument indicates

- whether the session will use a local transaction or whether it is non-transacted and,
- if the session is non-transacted, what mode whould be used for acknowledging the receipt of messages.

```
Session session;

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

9.1.5. Creating a MessageProducer

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

```
MessageProducer sender;

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

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

9.1.6. Creating a MessageConsumer

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

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

```
MessageConsumer receiver;

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

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

9.1.7. Starting Message Delivery

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

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

9.1.8. Using a TextMessage

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

The following demonstrates how to create such a message:

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

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

9.2. Sending and Receiving Messages

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

- Create a message
- Send a message
- Receive a message synchronously

9.2.1. Sending a Message

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

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

9.2.2. Receiving a Message Synchronously

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

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

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

```
TextMessage stockMessage;

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

9.2.3. Unpacking a TextMessage

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

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

newStockData = message.getText();
```

9.3. Other Messaging Features

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

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

9.3.1. Receiving Messages Asynchronously

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

```
import javax.jms.*;

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

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

```
StockListener myListener = new StockListener();

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

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

```
public void onMessage(Message message)
{
   String newStockData;

   /* Unpack and handle the messages received */
   newStockData = message.getText();
   if(...)
   {
      /* Logic related to the data */
   }
}
```

9.3.2. Using Message Selection

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

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

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

/* Set the message’s text to be the stockData string */
```

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

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

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

```
String selector;

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

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

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

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

9.3.3. Using Durable Subscriptions

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

9.3.3.1. Creating a Durable Subscription

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

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

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

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

/* Create connection and session */
Connection connection;
Session session;
connection = ConnectionFactory.createConnection();
session = connection.createSession(
   Session.AUTO_ACKNOWLEDGE);
```

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

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

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

9.3.3.2. Reconnecting to a Topic using a Durable Subscription

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

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

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

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

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

9.4. JMS Message Types

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

9.4.1. Creating a TextMessage

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

The following code demonstrates how to create such a message:

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

message = session.createTextMessage();

/* Set the stockData string to the message body */
message.setText(stockData);
```

9.4.2. Unpacking a TextMessage

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

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

stockInfo = message.getText();
```

9.4.3. Creating a BytesMessage

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

Such a message can be constructed in the following way:

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

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

9.4.4. Unpacking a BytesMessage

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

```
/* Byte array to hold stock information */
byte[] stockData;

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

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

9.4.5. Creating a MapMessage

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

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

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

```
String stockName;  /* Name of the stock */
double stockValue; /* Current value of the stock */
long stockTime;    /* Time stock quote was updated */
double stockDiff;  /* +/- 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.

```
/* First parameter is the name of the map element,
*  second is the value
*/
message.setString("Name", "SUNW");
message.setDouble("Value", stockValue);
message.setLong("Time", stockTime);
message.setDouble("Diff", stockDiff);
message.setString("Info",
  "Recent server announcement causes market interest");
```

9.4.6. Unpacking a MapMessage

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

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

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

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

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

9.4.7. Creating a StreamMessage

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

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

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

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

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

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

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

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

9.4.8. Unpacking a StreamMessage

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

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

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

9.4.9. Creating an ObjectMessage

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

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

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

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

```
/* Establish the values for the StockObject */
stockObject.setName(stockName);
stockObject.setValue(stockValue);
stockObject.setTime(stockTime);
stockObject.setDiff(stockDiff);
stockObject.setInfo(stockInfo);
```

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

```
/* Create an ObjectMessage */
ObjectMessage message;
message = session.createObjectMessage();

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

9.4.10. Unpacking an ObjectMessage

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

```
StockObject stockObject;

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

/* Extract data from the StockObject by using
   StockObject methods */
String stockName;  /* Name of the stock quote */
double stockValue; /* Current value of the stock */
long stockTime;    /* Time stock quote was updated */
double stockDiff;  /* +/- change in the stock quote */
String stockInfo;  /* Information on this stock */
stockName  = stockObject.getName();
stockValue = stockObject.getValue();
stockTime  = stockObject.getTime();
stockDiff  = stockObject.getDiff();
stockInfo  = stockObject.getInfo();
```

10. Use of JMS API in Java EE applications

10.1. Overview

The Java™ Platform, Enterprise Edition (Java EE) Specification, v7 requires support for the JMS API as part of the full Java EE platform.

The Java EE platform provides a number of additional features which are not available in the Java Platform Standard Edition (Java SE). These include the following:

- Support for distributed transactions which are demarcated either programmatically, using methods on `javax.transaction.UserTransaction`, or automatically by the container. These are referred to in this specification as JTA transactions to distinguish them from JMS local transactions.
- Support for JMS message-driven beans.

These features are defined in detail in other specifications including the Java EE 7 specification and the Enterprise JavaBeans 3.2 specification. However the use of the Java EE platform imposes restrictions on the way that the JMS API may be used by applications, and those restrictions are described here.

The JMS specification does not define how a Java EE container integrates with its JMS provider. Different Java EE containers may integrate with their JMS provider in different ways.

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

10.2. Restrictions on the use of JMS API in the Java EE web or EJB container

JMS applications which run in the Java EE web or EJB container are subject to a number of restrictions in the way the JMS API may be used. These restrictions are necessary for the following reasons:

- In a Java EE web or EJB container, a JMS provider operates as a transactional resource manager which must participate in JTA transactions as defined in the Java EE platform specification. This overrides the behaviour of JMS sessions as defined elsewhere in the JMS specification. For more details see section 12.3 "Behaviour of JMS sessions in the Java EE web or EJB container".
- The Java EE web or EJB containers need to be able to manage the threads used to run applications.
- The Java EE web and EJB containers perform connection management which may include the pooling of JMS connections.

The restrictions described in this section do not apply to the Java EE application client container.

Applications running in the Java EE web and EJB containers must not attempt to create more than one active (not closed) `Session` object per connection.

- If an application attempts to use the `Connection` object's `createSession` method when an active `Session` object exists for that connection then a `JMSException` should be thrown.
- If an application attempts to use the `MessagingContext` object's `createMessagingContext` method then a `JMSRuntimeException` should be thrown, since the first messaging context already contains a connection and session and this method would create a second session on the same connection.

The following methods are intended for use by the application server and their use by applications running in the Java EE web or EJB container may interfere with the container's ability to properly manage the threads used in the runtime environment. They must therefore not be called by applications running in the Java EE web or EJB container:

- `javax.jms.Session` method `setMessageListener`
- `javax.jms.Session` method `getMessageListener`
- `javax.jms.Session` method `setBatchMessageListener`
- `javax.jms.Session` method `getBatchMessageListener`
- `javax.jms.Session` method `run`
- `javax.jms.Connection` method `createConnectionConsumer`
- `javax.jms.Connection` method `createDurableConnectionConsumer`

The following methods may interfere with the container's ability to properly manage the threads used in the runtime environment and must not be used by applications running in the Java EE web or EJB container:

- `javax.jms.MessageConsumer` method `setMessageListener`
- `javax.jms.MessageConsumer` method `getMessageListener`
- `javax.jms.MessageConsumer` method `setBatchMessageListener`
- `javax.jms.MessageConsumer` method `getBatchMessageListener`
- `javax.jms.MessagingContext` method `setMessageListener`
- `javax.jms.MessagingContext` method `getMessageListener`
- `javax.jms.MessagingContext` method `setBatchMessageListener`
- `javax.jms.MessagingContext` method `getBatchMessageListener`

This restriction means that applications running in the Java EE web or EJB container which need to receive messages asynchronously may only do so using message-driven beans.

The following methods may interfere with the container's management of connections and must not be used by applications running in the Java EE web or EJB container:

- `javax.jms.Connection` method `setClientID`
- `javax.jms.Connection` method `stop`
- `javax.jms.Connection` method `setExceptionListener`
- `javax.jms.MessagingContext` method `setClientID`
- `javax.jms.MessagingContext` method `stop`
- `javax.jms.MessagingContext` method `setExceptionListener`

Applications which need to use a specific client identifier must set it on the connection factory, as described in section 4.3.2 "Client Identifier"

All the methods listed in this section may throw a `javax.jms.JMSException` (if allowed by the method) or a `javax.jms.JMSRuntimeException` (if not) when called by an application running in the Java EE web or EJB container. This is recommended but not required.

10.3. Behaviour of JMS sessions in the Java EE web or EJB container

The behaviour of a JMS session in respect of transactions and message acknowledgement is different for applications which run in a Java EE web or EJB container than it is for applications which run in a normal Java SE environment or in the Java EE application client container.

These differences also apply to JMS messaging context objects, since these incorporate a JMS session.

When an application creates a JMS session or messaging context in a Java EE web or EJB container, and there is an active JTA transaction in progress, then the session that is created will participate in the JTA transaction and will be committed or rolled back when the JTA transaction is committed or rolled back. Any session parameters that are specified when creating the session or messaging context are ignored. The use of local transactions or client acknowledgement is not permitted.

This applies irrespective of whether the JTA transaction is demarcated automatically by the container or programmatically using methods on `javax.transaction.UserTransaction`.

The term "session parameters" here refers to the arguments that may be passed into a call to the createSession or createMessagingContext methods to specify whether the session should use a local transaction and, if the session is non-transacted, what the acknowledgement mode should be.

When an application creates a JMS session or messaging context in a Java EE web or EJB container, and there is no active JTA transaction in progress, then the session that is created will be non-transacted and will be automatically-acknowledged. The use of local transactions or client acknowledgement is still not permitted. Parameters may be specified when creating the session or messaging context to specify whether the acknowledgement mode should be `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`. If any other session parameter values are

specified they will be ignored and an acknowledgement mode of `AUTO_ACKNOWLEDGE` used.

The use of local transactions or client acknowledgement is not permitted in a Java EE web or EJB container even if there is no active JTA transaction because this would require applications to be written differently depending on whether there was a JTA transaction or not.

The JMS API provides the following methods to create a session which allow the session to be defined using either the two parameters `transacted` and `acknowledgeMode` or by the single parameter `sessionMode`. If these methods are called in a Java EE web or EJB container then these parameters will be overridden as described above.

- `javax.jms.Connection` method `createSession(boolean transacted, int acknowledgeMode)`
- `javax.jms.QueueConnection` method `createQueueSession(boolean transacted, int acknowledgeMode)`
- `javax.jms.TopicConnection` method `createTopicSession(boolean transacted, int acknowledgeMode)`
- `javax.jms.Connection` method `createSession(int sessionMode)`

It is recommended that applications that run in the Java EE web or EJB container create a session using the following method which does not specify a parameter:

- `javax.jms.Connection` method `createSession()`

The JMS API provides the following methods to create a messaging context which allow the session to be defined using the single parameter `sessionMode`. If these methods are called in a Java EE web or EJB container then this parameter will be overridden as described above.

- `javax.jms.ConnectionFactory` method `createMessagingContext(int sessionMode)`
- `javax.jms.ConnectionFactory` method `createMessagingContext(String userName, String password, int sessionMode)`

The following method to create a messaging context from an existing messaging context is not permitted in a Java EE web or EJB container because it would create a second session on an existing connection, which is not permitted in a java EE web or EJB container.]

- `javax.jms.MessagingContext` method `createMessagingContext(int sessionMode)`

It is recommended that applications that run in the Java EE web or EJB container creates a messaging context using one of the following methods which do not specify a `sessionMode`:

- `javax.jms.ConnectionFactory` method `createMessagingContext()`

- `javax.jms.ConnectionFactory` method `createMessagingContext(String username, String password)`

When programmatic transaction demarcation is being used, the session should be both created and used within an active JTA transaction.

If a session or messaging context is created when there is an active JTA transaction, then after that transaction is committed or rolled back the session remains available for use in any subsequent JTA transaction until the session or messaging context is closed.

However, if a session or messaging context is created when there is an active JTA transaction but is subsequently used to send or receive messages when there is no active JTA transaction then the behaviour is undefined.

Similarly, if a session or messaging context is created when there is no active JTA transaction but subsequently used to send or receive messages when there is an active JTA transaction then the behaviour is undefined.

The Bean Provider should not make use of the JMS request/reply paradigm (sending of a JMS message, followed by the synchronous receipt of a reply to that message) within a single transaction. Because a JMS message is typically not delivered to its final destination until the transaction commits, the receipt of the reply within the same transaction will not take place.

11. Simplified JMS API

The simplified JMS API offers an alternative to the standard JMS API. It is designed to allow applications which use JMS to be simpler and less verbose than is possible when using the standard API. It was added to the JMS specification in version 2.0.

This section uses the term "standard API" to refer to the following JMS interfaces which have existing since JMS 1.1:

- `javax.jms.Connection` and its subtypes `javax.jms.QueueConnection` and `javax.jms.TopicConnection`.
- `javax.jms.Session` and its subtypes `javax.jms.QueueSession` and `javax.jms.TopicSession`.
- `javax.jms.MessageProducer` and its subtypes javax.jms.QueueSender and `javax.jms.TopicPublisher`.
- `javax.jms.MessageConsumer` and its subtypes `javax.jms.QueueReceiver` and `javax.jms.TopicSubscriber`.

The term "simplified API" refers to the following JMS interfaces which were added in JMS 2.0:

- `javax.jms.MessagingContext`
- `javax.jms.SyncMessageConsumer`
- `javax.jms.JMSRuntimeException` and its subtypes `javax.jms.IllegalStateRuntimeException`, `javax.jms.InvalidClientIDRuntimeException`, `javax.jms.InvalidDestinationRuntimeException`, `javax.jms.InvalidSelectorRuntimeException`, `javax.jms.JMSSecurityRuntimeException`, `javax.jms.MessageFormatRuntimeException` and `javax.jms.TransactionRolledBackRuntimeException`.

All the other methods in the JMS API can be considered as part of both APIs. This includes:

- `javax.jms.ConnectionFactory` and its subtypes `javax.jms.QueueConnectionFactory` and `javax.jms.TopicConnection`
- `javax.jms.Message` and its subtypes `javax.jms.ObjectMessage`, `javax.jms.TextMessage`, `javax.jms.MapMessage`, `javax.jms.StreamMessage`, `javax.jms.BytesMessage`

11.1. Goals of the simplified API

The simplified API has the following goals:

- To reduce the number of objects needed to send and receive messages, and in particular to combine the JMS `Connection`,

`Session`, `MessageProducer` and `MessageConsumer` objects as much as possible into a single object.

- To maintain a consistent style with the existing API where possible so that users of the old API feel it to be an evolution which that can learn quickly. In particular the simplified API will continue to use the concepts of connection and session even though it doesn't require the use of `Connection` or `Session` objects.
- To be capable of use in both Java EE and Java SE applications.
- To allow resource injection to be exploited in those environment which support it.
- To provide the option to send and receive message payloads directly without the need to use `javax.jms.Message` objects.
- To remove where possible the need to catch `JMSException` on method calls
- To be functionally as complete as the standard API, so that users of the simplified API will not have the need to switch back to the standard API in order to perform an operation this is unavailable in the simplified API.
- To be an alternative to, but not a replacement for, the standard API. The standard API remains and is not deprecated. Developers who are familiar with the standard API, or who prefer it, may continue to use the standard API.

11.2. Key features of the simplified API

11.2.1. MessagingContext

The main object in the simplified API is `javax.jms.MessagingContext`. This combines in a single object the functionality of several separate objects from the standard API: a `Connection`, a `Session` and an anonymous `MessageProducer` (one with no destination specified). In addition, a messaging context provides the functionality of a `MessageConsumer`, but only for receiving messages asynchronously.

For receiving messages synchronously a separate object is provided. This is javax.jms.`SyncMessageConsumer`, which provides the functionality of `MessageConsumer` for synchronous message delivery. See section 13.2.5 "Consuming messages synchronously" for more information.

In terms of the standard API a `MessagingContext` may be thought of as containing a `Connection` and a `Session`. Although the simplified API does not require applications to use those objects, the concepts of connection and session remain important. A connection represents a physical link to the JMS server and a session represents a single-threaded context for sending and receiving messages.

A `MessagingContext` object may be created by calling one of four methods on a `ConnectionFactory`:

`ConnectionFactory` methods to create a `MessagingContext`	
`createMessagingContext()`	Creates a `MessagingContext` with the default user identity and an unspecified `sessionMode`.
`createMessagingContext(` `  int sessionMode)`	Creates a `MessagingContext` with the default user identity and the specified session mode.

`createMessagingContext(`
`  String userName,`
`  String password)`
	Creates a MessagingContext with the specified user identity and an unspecified sessionMode.

`createMessagingContext(`
`  String userName,`
`  String password,`
`  int sessionMode)`
	Creates a MessagingContext with the specified user identity and the specified session mode.

`sessionMode` here has the same meaning and values as the `createSession(int sessionMode)` method on a `Connection`.

Applications running in the Java EE web and EJB containers are not permitted to create more than one active session on a connection (see Section 12.2 "Restrictions on the use of JMS API in the Java EE web or EJB container"). Since a messaging context contains a single connection and a single session it is ideally suited for use by such applications.

Applications running in a Java SE environment or in the Java EE application client container are permitted to create multiple active sessions on the same connection. Such applications may create multiple `MessagingContext` objects which use different sessions but the same connection by calling the `createMessagingContext` method on an existing `MessagingContext` object:

`MessagingContext` methods to create a new `MessagingContext`	
`createMessagingContext(` `  int sessionMode)`	Creates a new `MessagingContext` with the specified session mode using the same connection as this `MessagingContext` and creating a new session.

11.2.1. Static constants for session mode

New static integer constants have been defined for use with the simplified API.

`MessagingContext.AUTO_ACKNOWLEDGE`,
`MessagingContext.CLIENT_ACKNOWLEDGE`,
`MessagingContext.DUPS_OK_ACKNOWLEDGE` and
`MessagingContext.SESSION_TRANSACTED`

These have the same values, and the same meaning, as the equivalent constants on `Session`. This avoids the need for applications that use the new API to use the `Session` interface.

11.2.2. Creating messages

An application using the standard API creates messages objects using methods on the `Session`.

An application using the simplified API may create messages using methods on the `MessagingContext`. These methods are identical in behaviour to the corresponding methods on Session.

`MessagingContext` methods to create a message	
`createMessage()`	Creates a `Message` object.
`createBytesMessage()`	Creates a `BytesMessage` object.
`createMapMessage()`	Creates a `MapMessage` object.
`createObjectMessage()`	Creates an `ObjectMessage` object.
`createObjectMessage( Serializable object)`	Creates an initialized `ObjectMessage` object.
`createStreamMessage()`	Creates a `StreamMessage` object.
`createTextMessage()`	Creates a `TextMessage` object.
`createTextMessage(String text)`	Creates an initialized `TextMessage` object.

11.2.3. Sending messages

An application using the standard API sends messages using methods on `MessageProducer`.

An application using the simplified API applications does not create a `MessageProducer`. Instead it simply uses methods on `MessagingContext`.

The messaging context behaves as an anonymous producer. An anonymous producer is one which has not been configured to use a particular destination. The `Destination` object must therefore be supplied as the first argument of each `send` method.

`Destination` objects can be obtained either by JNDI lookup or by calling the `MessagingContext` methods `createQueue(String queueName)` or `createTopic (String topicName).`

The following two methods may be used to send a message synchronously.

`MessagingContext` methods to send a message	
`send( Destination destination, Message message)`	Sends a message to the specified destination, using the `MessagingContext`'s default delivery mode, priority, and time to live.

`send(`
`  Destination destination,`
`  Message message,`
`  int deliveryMode,`
`  int priority,`
`  long timeToLive)`
	Sends a message to the specified destination, specifying delivery mode, priority and time to live.

The following two methods may be used to send a message asynchronously. The call to `send` returns immediately and a separate thread is used to send the message. When the sending of the message is complete a `CompletionListener` supplied by the caller is invoked.

`MessagingContext` methods to send a message asynchronously	

`send(`
`  Destination destination,`
`  Message message,`
`  CompletionListener`
`  completionListener)`
	Sends a message to the specified destination, using the `MessagingContext`'s default delivery mode, priority, and time to live, returning immediately and notifying the specified completion listener when the operation has completed.

`send(`
`  Destination destination,`
`  Message message,`
`  int deliveryMode,`
`  int priority,`
`  long timeToLive,`
`  CompletionListener`
`  completionListener)`
	Sends a message to the specified destination, specifying delivery mode, priority and time to live, returning immediately and notifying the specified completion listener when the operation has completed.

The following methods may be used to send a message payload directly, without the need for the application to explicitly create a message object.

`MessagingContext` methods to send a method payload directly	

`send(`
`  Destination destination,`
`  String payload)`
	Send a `TextMessage` with the specified payload to the specified destination, using the `MessagingContext`'s default delivery mode, priority and time to live.

`send(`
`  Destination destination,`
`  Serializable payload)`
	Send an `ObjectMessage` with the specified payload to the specified destination, using the `MessagingContext`'s default delivery mode, priority and time to live.

`send(`
`  Destination destination,`
`  byte[] payload)`
	Send a `BytesMessage` with the specified payload to the specified destination, using the `MessagingContext`'s default delivery mode, priority and time to live.

`send(`
` Destination destination,`
` Map<String,Object> payload)`
	Send a `MapMessage` with the specified payload to the specified destination, using the `MessagingContext`'s default delivery mode, priority and time to live.

11.2.4. Consuming messages asynchronously

An application using the standard API to consume messages asynchronously needs to create a `MessageConsumer` on the specified destination and use its `setMessageListener` and `setBatchMessageListener` methods to register a message listener which is invoked when a message is delivered.

An application using the simplified API simply calls one of a number of `setMessageListener` and `setBatchMessageListener` methods on the `MessagingContext` itself. The `Destination` object must be supplied as the first argument of these methods.

The following methods are used to set a `MessageListener`:

`MessagingContext` methods to set a `MessageListener` to consume messages asynchronously	

`setMessageListener(`
`  Destination destination,`
`  MessageListener listener)`
	Creates a new consumer on the specified destination that will deliver messages to the specified `MessageListener`.

`setMessageListener(`
`  Destination destination,`
`  String messageSelector,`
`  boolean noLocal,`
`  MessageListener listener)`
	Creates a consumer on the specified destination that will deliver messages to the specified `MessageListener`, using a message selector. This method can specify whether messages published by its own connection should be delivered to it, if the destination is a topic.

`setMessageListener(`
`  Destination destination,`
`  String messageSelector,`
`  MessageListener listener)`
	Creates a consumer on the specified destination that will deliver messages to the specified `MessageListener`, using a message selector.

`setMessageListener(`
`  Topic topic,`
`  String subscriptionName,`
`  MessageListener listener)`
	Creates a durable subscription with the specified name on the specified topic, and creates a consumer on that durable subscription that will deliver messages to the specified `MessageListener`.

`setMessageListener(`
`  Topic topic,`
`  String subscriptionName,`
`  String messageSelector,`
`  boolean noLocal,`
	Creates a durable subscription with the specified name on the specified topic, and creates a consumer on that durable subscription that will deliver

`MessageListener listener)`	messages to the specified `MessageListener`, using a message selector and specifying whether messages published by its own connection should be added to the durable subscription.

The following methods are used to set a `BatchMessageListener`:

`MessagingContext` methods to set a `BatchMessageListener` to consume messages asynchronously in batches.	

`setBatchMessageListener(`
`Destination destination,`
`BatchMessageListener`
`listener,`
`int maxBatchSize,`
`long batchTimeout)`
	Creates a new consumer on the specified destination that will deliver messages in batches to the specified `BatchMessageListener` using the specified maximum batch size and timeout.

`setBatchMessageListener(`
`Destination destination,`
`String messageSelector,`
`BatchMessageListener`
`listener,`
`int maxBatchSize,`
`long batchTimeout)`
	Creates a consumer on the specified destination, that will deliver messages in batches to the specified `BatchMessageListener` using a message selector and the specified maximum batch size and timeout.

`setBatchMessageListener(`
`Destination destination,`
`String messageSelector,`
`boolean noLocal,`
`BatchMessageListener`
`listener,`
`int maxBatchSize,`
`long batchTimeout)`
	Creates a consumer on the specified destination that will deliver messages in batches to the specified `BatchMessageListener`, using a message selector and the specified maximum batch size and timeout. This method can specify whether messages published by its own connection should be delivered to it, if the destination is a topic.

`setBatchMessageListener(`
`Topic topic,`
`String subscriptionName,`
`BatchMessageListener`
`listener,`
`int maxBatchSize,`
`long batchTimeout)`
	Creates a durable subscription with the specified name on the specified topic, and creates a consumer on that durable subscription that will deliver messages in batches to the specified `BatchMessageListener` using the specified maximum batch size and timeout.

`setBatchMessageListener(`
`Topic topic,`
`String subscriptionName,`
`String messageSelector,`
`boolean noLocal,`
`BatchMessageListener`
`listener,`
	Creates a durable subscription with the specified name on the specified topic, and creates a consumer on that durable subscription that will deliver messages in batches to the specified

`int maxBatchSize,` `long batchTimeout)`	`BatchMessageListener` using a message selector, the specified maximum batch size and timeout, and specifying whether messages published by its own connection should be added to the durable subscription.

11.2.5. Consuming messages synchronously

An application using the standard API to consume messages synchronously needs to create a `MessageConsumer` on the specified destination and then call its `receive()`, `receive(int timeout)` or `receiveNoWait()` methods.

An application using the simplified API will continue to need to create a separate consumer object, a `SyncMessageConsumer`. This may be thought of as a stripped-down `MessageConsumer` for synchronous delivery only.

The following methods are used to create a `SyncMessageConsumer`:

`MessagingContext` methods to create a `SyncMessageConsumer`	
`createSyncConsumer(` `Destination destination)`	Creates a `SyncMessageConsumer` for the specified destination.

`createSyncConsumer(`
`Destination destination,`
`String messageSelector)`
	Creates a `SyncMessageConsumer` for the specified destination, using a message selector.

`createSyncConsumer(`
`Destination destination,`
`String messageSelector,`
`boolean noLocal)`
	Creates a `SyncMessageConsumer` for the specified destination, using a message selector. This method can specify whether messages published by its own connection should be delivered to it, if the destination is a topic.

`createSyncDurableConsumer(`
`Topic topic,`
`String name)`
	Creates a durable subscription with the specified name on the specified topic, and creates a `SyncMessageConsumer` on that durable subscription.

`createSyncDurableConsumer(`
`Topic topic,`
`String name,`
`String messageSelector,`
`boolean noLocal)`
	Creates a durable subscription with the specified name on the specified topic, and creates a `SyncMessageConsumer` on that durable subscription, specifying a message selector and whether messages published by its own connection should be added to the durable subscription.

The following methods on SyncMessageConsumer may be used to synchronously consume messages:

SyncMessageConsumer methods to synchronously consume messages	
`receive()`	Receives the next message produced for this `SyncMessageConsumer`.
`receive(` `  long timeout)`	Receives the next message that arrives within the specified timeout interval.
`receiveNoWait()`	Receives the next message if one is immediately available.

The reason that `SyncMessageConsumer` is provided as a separate object rather than combined with `MessagingContext` is to cater for JMS providers which choose to cache messages in the consumer prior to delivery to consumers. JMS does not require that messages are cached in the consumer prior to delivery or define how such a cache would behave. However the use of the `SyncMessageConsumer` object allows such a cache to be created when the `SyncMessageConsumer` is created and destroyed when `SyncMessageConsumer close` method is called.

In addition, the following methods may be used to receive a message payload directly:

SyncMessageConsumer methods to consume message mayloads directly	
`<T> T receivePayload(` `  Class<T>  c);`	Receives the next message produced for this SyncMessageConsumer and returns its payload, which must be of the specified type
`<T> T receivePayload(` `  Class<T> c, long timeout);`	Receives the next message produced for this SyncMessageConsumer and that arrives within the specified timeout period, and returns its payload, which must be of the specified type
`<T> T receivePayloadNoWait(` `  Class<T> c);`	Receives the next message produced for this SyncMessageConsumer and returns its payload, which must be of the specified type

These methods may be used when receiving nessages of type `TextMessage` and `ObjectMessage`, and the application has no need to access message headers or properties.

If the next message is a `TextMessage` then the first argument should be set to `String.class`. If the next message is an `ObjectMessage` this should be set to `Serializable.class`. If the next message is not of the expected type a `ClassCastException` will be thrown and the message will not be delivered.

11.2.6. Closing the MessagingContext

A `MessagingContext` needs to be closed after use using the `close()` method.

This closes the underlying session and any underlying producers and consumers. If there are no other active (not closed) `MessagingContext` objects using the underlying connection then this method also closes the underlying connection.

The `MessagingContext` interface extends the `java.lang.AutoCloseable` interface. This means that applications which create the `MessagingContext` in a `try-with-resources` statement do not need to call the `close` method when the connection is no longer needed. Instead the session will be closed automatically at the end of the statement. The use of a `try-with-resources` statement also simplifies the handling of any exceptions thrown by the `close` method.

11.2.7. Automatic start of message delivery

An application using the standard API to consume messages needs to call the connection's `start` method to start delivery of incoming messages. It may temporarily suspend delivery by calling `stop`, after which a call to `start` will restart delivery. This is described in section 4.3.3 "Connection Setup".

The simplified API provides corresponding `start` and `stop` methods on `MessagingContext`. The `start` method will be called automatically when `setMessageListener, setBatchMessageListener` or `createSyncMessageConsumer` are called on the `MessageContext` object. This means there is no need for the application to call `start` when the consumer is first established. As with the standard API, an application may temporarily suspend delivery by calling `stop`, after which a call to `start` will restart delivery.

Sometimes an application will need the connection to remain in stopped mode until setup is complete and not commence message delivery until the `start` method is explicitly called, as with the standard API/ This can be configured by calling `setAutoStart(false)` on the `MessagingContext` prior to calling `setMessageListener, setBatchMessageListener` or `createSyncMessageConsumer`.

11.2.8. Threading restrictions on a MessagingContext

Since a MessagingContext incorporates a Session it is subject to the same threading restrictions as a Session. These are described in section 4.4.6 "Conventions for Using a Session" which explains how a session may only be used by one thread at a time.

The `MessagingContext` method `createMessagingContext` does not use its underlying Session and so it not subject to this threading restriction.

When a MessagingContext is used to consume messages asynchromously, message delivery will begin as soon a message listener is set, as described in section 13.2.7 "Automatic start of message delivery". This means that once `setMessageListener` or `setBatchMessageListener` has been called, the messaging context's session will become dedicated to the thread of control that delivers messages to the listener and the application must

not subsequently call methods on the `MessagingContext` from another thread of control.

The only exceptions to this are the messaging context's `close` method (since closing the session from another thread is permitted), and its `setMessageListener` or `setBatchMessageListener` methods. The JMS provider will be responsible for ensuring that a second message listener may be safely configured even if the underlying connection has been started.

11.2.9. Exceptions

Most methods in the standard API are declared as throwing a javax.jms.JMSException or a subtype of it. This means that the calling method must either catch this exception or declare it in its own `throws` statement.

The simplified API does not throw checked exceptions. Instead, equivalent unchecked exceptions are thrown instead.

The new unchecked exceptions, and the checked exceptions to which they correspond are listed below:

Old checked exception	Corresponding new unchecked exception
`JMSException`	`JMSRuntimeException`
`TransactionRolledBack Exception`	`TransactionRolledBack RuntimeException`
`IllegalStateException`	`IllegalStateRuntimeException`
`InvalidDestination Exception`	`InvalidDestination RuntimeException`
`InvalidSelectorException`	`InvalidSelectorRuntimeExcepti on`
`MessageFormatException`	`MessageFormatRuntimeException`
`JMSSecurityException`	`JMSSecurityRuntimeException`
`InvalidClientID Exception`	`InvalidClientID RuntimeException`

11.3. Injection of MessagingContext objects

This section relates to application classes which run in the Java EE web, EJB or application client containers and which support injection. Section EE.5 of the Java EE specification lists the application classes that support injection.

Applications may declare a field of type `javax.jms.MessagingContext` and annotate it with the `javax.inject.Inject` annotation:

```
@Inject
private MessagingContext context;
```

The container will inject a `MessagingContext`. It will have request scope and will be automatically closed when the request ends. However, unlike a normal CDI request-scoped object, a separate `MessagingContext` instance will be injected for every injection point.

The annotation `javax.jms.JMSConnectionFactory` may be used to specify the JNDI lookup name of the `ConnectionFactory` used to create the messaging context. For example:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
private MessagingContext context;
```

If no lookup name is specified or the `JMSConnectionFactory` annotation is omitted then the platform default JMS connection factory will be used.

The annotation `javax.jms.JMSPasswordCredential` may be used to specify a user name and password which will be used when the messaging context is created. For example:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSPasswordCredential(userName="admin",password="mypassword")
private MessagingContext context;
```

The annotation `javax.jms.JMSSessionMode` may be used to specify the session mode of the messaging context. For example:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSSessionMode(MessagingContext.AUTO_ACKNOWLEDGE)
private MessagingContext context;
```

The meaning and possible values of session mode are the same as for the `ConnectionFactory` method `createMessagingContext(int sessionMode)`:

- In the Java EE application client container, session mode may be set to any of `MessagingContext.SESSION_TRANSACTED`, `MessagingContext.CLIENT_ACKNOWLEDGE`, `MessagingContext.AUTO_ACKNOWLEDGE` or `MessagingContext.DUPS_OK_ACKNOWLEDGE`. If no session mode is specified or the `JMSSessionMode` annotation is omitted a session mode of `MessagingContext.AUTO_ACKNOWLEDGE` will be used.
- In a Java EE web or EJB container, when there is an active JTA transaction in progress, session mode is ignored and the `JMSSessionMode` annotation is unnecessary.
- In a Java EE web or EJB container, when there is no active JTA transaction in progress, session mode may be set to either of `MessagingContext.AUTO_ACKNOWLEDGE` or `MessagingContext.DUPS_OK_ACKNOWLEDGE`. If no session mode is specified or the `JMSSessionMode` annotation is omitted a session mode of `MessagingContext.AUTO_ACKNOWLEDGE` will be used.

For more information about the use of session mode when creating a messaging context, see section 10.3 "Behaviour of JMS sessions in the Java EE web or EJB container" and the API documentation for the `ConnectionFactory` method `createMessagingContext(int sessionMode)`.

By default the injected messaging context will automatically start its underlying connection when a consumer is created. This can be disabled using the `javax.jms.JMSAutoStart` annotation. For example:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
@JMSSessionMode(MessagingContext.AUTO_ACKNOWLEDGE)
@JMSAutoStart(false)
private MessagingContext context;
```

This is equivalent to calling `setAutoStart(false)` on the injected messaging context.

11.4. Examples using the simplified API

The examples in this section compare the use of the standard and simplified JMS APIs for some common JMS operations.

11.4.1. Sending a message (Java EE)

This example compares the use of the standard and simplified JMS APIs for sending a `TextMessage` in a Java EE web or EJB container.

11.4.1.1. Example using the standard API

Here's how you might do this using the standard API:

```
@Resource(lookup = "jms/connectionFactory ")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageOld (String payload) throws JMSException{
   try (Connection connection =
         connectionFactory.createConnection()) {
      Session session = connection.createSession();
      MessageProducer messageProducer =
         session.createProducer(inboundQueue);
      TextMessage textMessage =
         session.createTextMessage(payload);
      messageProducer.send(textMessage);
   }
}
```

11.4.1.2. Example using the simplified API

Here's how you might do this using the simplified API:

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public void sendMessageNew (String payload) {

   try (MessagingContext context =
         connectionFactory.createMessagingContext();){
      context.send(inboundQueue,payload);
   }
}
```

Note that `sendMessageNew` does not need to throw `JMSException`.

11.4.1.3. Example using the simplified API and injection

Here's how you might do this using the simplified API with the MessagingContext created by injection:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
private MessagingContext messagingContext;

@Resource(mappedName = "jms/inboundQueue")
private Queue inboundQueue;

public void sendMessageNew(String payload) {
   messagingContext.send(inboundQueue, payload);
}
```

11.4.2. Sending a message (Java SE)

This example compares the use of the standard and simplified JMS APIs for sending a `TextMessage` in a Java SE environment.

11.4.2.1. Example using the standard API

Here’s how you might do this using the standard API:

```
public void sendMessageOld(String payload)
      throws JMSException, NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (Connection connection =
         connectionFactory.createConnection()) {
      Session session = connection.createSession();
      MessageProducer messageProducer =
         session.createProducer(inboundQueue);
      TextMessage textMessage = session.createTextMessage(payload);
      messageProducer.send(textMessage);
   }
}
```

11.4.2.2. Example using the simplified API

Here's how you might do this using the simplified API:

```
public void sendMessageNew(String payload) throws NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context =
         connectionFactory.createMessagingContext();){
      context.send(inboundQueue,payload);
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

11.4.3. Receiving a message synchronously (Java EE)

This example compares the use of the standard and simplified JMS APIs for synchronously receiving a `TextMessage` in a Java EE web or EJB container.

11.4.3.1. Example using the standard API

Here's how you might do this using the standard API:

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageOld() throws JMSException {

   try (Connection connection =
         connectionFactory.createConnection()) {
      connection.start();
      Session session = connection.createSession();
      MessageConsumer messageConsumer =
         session.createConsumer(inboundQueue);
      TextMessage textMessage =
         (TextMessage)messageConsumer.receive();
      String payload = textMessage.getText();
      return payload;
   }
}
```

11.4.3.2. Example using the simplified API

Here's how you might do this using the simplified API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageNew() {
   try (MessagingContext context =
         connectionFactory.createMessagingContext()) {
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncConsumer(inboundQueue);
      return syncMessageConsumer.receivePayload(String.class);
   }
}
```

Note that `receiveMessageNew` does not need to throw `JMSException`.

11.4.3.3. Example using the simplified API and injection

Here's how you might do this using the simplified API with the MessagingContext created by injection:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory")
private MessagingContext messagingContext;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageNew() {
   SyncMessageConsumer syncMessageConsumer =
      messagingContext.createSyncConsumer(inboundQueue);
   return syncMessageConsumer.receivePayload(String.class);
}
```

11.4.4. Receiving a message synchronously (Java SE)

This example compares the use of the standard and simplified JMS APIs for synchronously receiving a `TextMessage` in a Java SE environment.

11.4.4.1. Example using the standard API

Here's how you might do this using the standard API:

```
public String receiveMessageOld()
         throws JMSException, NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (Connection connection =
         connectionFactory.createConnection();){
      Session session = connection.createSession(AUTO_ACKNOWLEDGE);
      MessageConsumer messageConsumer =
         session.createConsumer(inboundQueue);
      connection.start();
      TextMessage textMessage=
         (TextMessage) messageConsumer.receive();
      return textMessage.getText();
   }
}
```

11.4.4.2. Example using the simplified API

Here's how you might do this using the simplified API.

```
public String receiveMessageNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue)initialContext.lookup("jms/inboundQueue");

   try (MessagingContext messagingContext =
         connectionFactory.createMessagingContext(
         AUTO_ACKNOWLEDGE);) {
      SyncMessageConsumer syncMessageConsumer =
         messagingContext.createSyncConsumer(inboundQueue);
      return syncMessageConsumer.receivePayload(String.class);
   }
}
```

Note that `receiveMessageNew` does not need to throw `JMSException`.

11.4.5. Receiving a message synchronously from a durable subscription (Java EE)

This example compares the use of the standard and simplified JMS APIs for synchronously receiving a `TextMessage` from a durable topic subscription in a Java EE web or EJB container.

11.4.5.1. Example using the standard API

Here's how you might do this using the standard API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageOld() throws JMSException {

   try (Connection connection =
         connectionFactory.createConnection()) {
      Session session = connection.createSession();
      MessageConsumer messageConsumer =
         session.createDurableConsumer(inboundTopic, "mysub");
      connection.start();
      TextMessage textMessage=
         (TextMessage) messageConsumer.receive();
      return textMessage.getText();
   }
}
```

11.4.5.2. Example using the simplified API

Here's how you might do this using the simplified API.

```
@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/inboundQueue")
Queue inboundQueue;

public String receiveMessageNew() {

   try (MessagingContext context =
         connectionFactory.createMessagingContext()) {
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncDurableConsumer(inboundTopic, "mysub");
      return syncMessageConsumer.receivePayload(String.class);
   }
}
```

Note that `receiveMessageNew` does not need to throw an exception.

11.4.5.3. Example using the simplified API and injection

Here's how you might do this using the simplified API with the MessagingContext created by injection:

```
@Inject
@JMSConnectionFactory("jms/connectionFactory2")
private MessagingContext context;

@Resource(lookup="jms/inboundTopic")
Topic inboundTopic;

public String receiveMessageNew() {
    SyncMessageConsumer syncMessageConsumer =
       context.createSyncDurableConsumer(inboundTopic, "mysub");
    return syncMessageConsumer.receivePayload(String.class);
}
```

11.4.6. Receiving messages asynchronously (Java SE)

This example compares the use of the standard and simplified JMS APIs for asynchronously receiving `TextMessage` objects in a Java SE environment.

11.4.6.1. Example using the standard API

Here’s how you might do this using the standard API, using a message listener class `MyListener`:

```
public void receiveMessagesOld()
      throws JMSException, NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (Connection connection =
         connectionFactory.createConnection();){
      Session session = connection.createSession(AUTO_ACKNOWLEDGE);
      MessageConsumer consumer =
         session.createConsumer(inboundQueue);
      MessageListener messageListener = new MyListener();
      consumer.setMessageListener(messageListener);
      connection.start();

      // wait for messages to be received - details omitted
    }
}
```

11.4.6.2. Example using the simplified API

Here's how you might do this using the simplified API.

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context =
      connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);){
      MessageListener messageListener = new MyListener();
      context.setMessageListener(inboundQueue,messageListener);

      // wait for messages to be received - details omitted
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

11.4.7. Receiving a message asynchronously from a durable subscription (Java SE)

This example compares the use of the standard and simplified JMS APIs for asynchronously receiving a `TextMessage` from a durable topic subscription in a Java SE environment.

11.4.7.1. Example using the standard API

Here's how you might do this using the standard API, using a message listener class `MyListener`:

```
public void receiveMessagesOld()
      throws JMSException, NamingException{

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Topic inboundTopic =
      (Topic)initialContext.lookup("jms/inboundTopic");

   try (Connection connection =
         connectionFactory.createConnection();) {
      Session session =
         connection.createSession( AUTO_ACKNOWLEDGE);
      session.createDurableSubscriber(inboundTopic, "");
      TopicSubscriber topicSubscriber =
         session.createDurableSubscriber(inboundTopic, "mysub");
      MessageListener messageListener = new MyListener();
      topicSubscriber.setMessageListener(messageListener);

      connection.start();

     // wait for messages to be received - details omitted
   }
}
```

11.4.7.2. Example using the simplified API

Here's how you might do this using the simplified API:

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Topic inboundTopic =
      (Topic) initialContext.lookup("jms/inboundTopic");

   try (MessagingContext context =
      connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);){
      MessageListener messageListener = new MyListener();
      context.setMessageListener(
         inboundTopic, "mysub", messageListener);

      // wait for messages to be received - details omitted
   }
}
```

Note that `receiveMessagesNew` does not need to throw `JMSException`.

11.4.8. Receiving a message in multiple threads (Java SE)

This example compares the use of the standard and simplified JMS APIs for asynchronously receiving `TextMessage` objects from a queue using multiple threads in a Java SE environment. In this example two threads are used, which means two sessions are needed. In this example, both sessions use the same connection.

11.4.8.1. Example using the standard API

Here’s how you might do this using the standard API, using a message listener class `MyListener`:

```
public void receiveMessagesOld()
      throws JMSException, NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory =  ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (Connection connection =
         connectionFactory.createConnection();){
      Session s1 = connection.createSession(AUTO_ACKNOWLEDGE);
      MessageConsumer messageConsumer1 =
         s1.createConsumer(inboundQueue);
      MyListener messageListener1 = new MyListener();
      messageConsumer1.setMessageListener(messageListener1);

      Session s2 = connection.createSession(AUTO_ACKNOWLEDGE);
      MessageConsumer messageConsumer2 =
         s2.createConsumer(inboundQueue);
      MyListener messageListener2 = new MyListener();
      messageConsumer2.setMessageListener(messageListener2);

      connection.start();

      // wait for messages to be received - details omitted
   }
}
```

11.4.8.2. Example using the simplified API

Here’s how you might do this using the simplified API:

```
public void receiveMessagesNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");

   try (MessagingContext context1 =
      connectionFactory.createMessagingContext(AUTO_ACKNOWLEDGE);
      MessagingContext context2 =
           context1.createMessagingContext(AUTO_ACKNOWLEDGE);){
      MyListener messageListener1 = new MyListener("One");
      context1.setMessageListener(inboundQueue,messageListener1);

      MyListener messageListener2 = new MyListener("Two");
      context2.setMessageListener(inboundQueue,messageListener2);

      // wait for messages to be received - details omitted
   }
}
```

11.4.9. Receiving synchronously and sending a message in the same local transaction (Java SE)

This example compares the use of the standard and simplified JMS APIs for the use case in which a Java SE application repeatedly consumes a message from one queue and forwards it to another queue in a Java SE environment. In this example each message is received and forwarded in the same local transaction. This means that the receiving and sending of the message must be done using the same transacted session which is then committed.

In this example the application consumes the incoming messages synchronously. However since this is a Java SE application the message could also be consumed asynchronously using a `MessageListener`.

11.4.9.1. Example using the standard API

Here's how you might do this using the standard API:

```
public void receiveAndSendMessageOld()
      throws JMSException, NamingException {
   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");
   Queue outboundQueue =
      (Queue) initialContext.lookup("jms/outboundQueue");

   try (Connection connection  =
         connectionFactory.createConnection();){
      Session session =
         connection.createSession(SESSION_TRANSACTED);
      MessageConsumer messageConsumer =
         session.createConsumer(inboundQueue);
      MessageProducer messageProducer =
         session.createProducer(outboundQueue);
      connection.start();

      TextMessage textMessage = null;
      do {
         textMessage = (TextMessage) messageConsumer.receive(1000);
         if (textMessage!=null){
            messageProducer.send(textMessage);
            session.commit();
         }
      } while (textMessage!=null);
   }
}
```

11.4.9.2. Example using the simplified API

Here’s how you might do this using the simplified API:

```
public void receiveAndSendMessageNew() throws NamingException {

   InitialContext initialContext = getInitialContext();
   ConnectionFactory connectionFactory = (ConnectionFactory)
      initialContext.lookup("jms/connectionFactory");
   Queue inboundQueue =
      (Queue) initialContext.lookup("jms/inboundQueue");
   Queue outboundQueue =
      (Queue) initialContext.lookup("jms/outboundQueue");

   try (MessagingContext context =
         connectionFactory.createMessagingContext(
         SESSION_TRANSACTED);){
      SyncMessageConsumer syncMessageConsumer =
         context.createSyncConsumer(inboundQueue);
      TextMessage textMessage = null;
      do {
         textMessage =
            (TextMessage) syncMessageConsumer.receive(1000);
         if (textMessage != null) {
            context.send(outboundQueue, textMessage);
            context.commit();
         }
      } while (textMessage != null);
   }
}
```

Note that receiveAndSendMessageNew does not need to throw `JMSException`.

11.4.10. Request/reply pattern using a TemporaryQueue (Java EE)

This example compares the use of the standard and simplified JMS APIs for implementing a request/reply pattern in a Java EE EJB container.

In this example, a session bean (the requestor) sends a request message to some queue (the request queue). The `setJMSReplyTo` property of the request message is set to a `TemporaryQueue`, to which the reply should be set. After sending the request, the session bean listens on the temporary queue until it receives the reply.

Since the request message won't actually be sent until the transaction is committed, the request message is sent in a separate transaction from that used to receive the reply.

A message-driven bean (the responder) listens on the request queue for request messages. When it receives a message it creates a reply message and sends it to the reply queue specified in the `setJMSReplyTo` property of the incoming message.

When implementing this pattern, the following features of JMS must be borne in mind:

- The same `Connection` object that was used to create the `TemporaryQueue` must also be used to consume the response message from it. (This is a restriction of temporary queues).
- If the request message is sent in a transaction then the response message must be consumed in a separate transaction. That's why the

message is sent in a separate business which has the transactional attribute `REQUIRES_NEW`.

11.4.10.1. Example using the standard API

Here's how you might implement the requestor this using the standard API:

There are two session beans involved in sending the request message. The first bean

The first session bean `RequestReplyOld` creates the creates the temporary reply queue, calls a second bean `SenderBeanOld` to send the request in a separate transaction and then listens for the reply:

```
@Stateless
@LocalBean
public class RequestReplyOld {

@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@EJB private SenderBeanOld senderBean;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String requestReplyOld(String request) throws JMSException {

   try (Connection connection =
         connectionFactory.createConnection()) {
      Session session = connection.createSession();
      TemporaryQueue replyQueue = session.createTemporaryQueue();

      // call a second bean to
      // send the request message in a separate transaction
     senderBeanOld.sendRequestOld(request,replyQueue);

      // now receive the reply, using the same connection
      // as was used to create the temporary reply queue
      MessageConsumer consumer= session.createConsumer(replyQueue);
      connection.start();
      TextMessage reply = (TextMessage) consumer.receive();
      return reply.getText();
   }
}
}
```

The second session bean `SenderBeanOld` simply sends the request to the request queue in a separate transaction:

```
@Stateless
@LocalBean
public class SenderBeanOld {

@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/requestQueue")
Queue requestQueue;

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void sendRequestOld(
      String requestString, TemporaryQueue replyQueue)
      throws JMSException {
   try (Connection connection =
         connectionFactory.createConnection()) {
      Session session = connection.createSession();
      TextMessage requestMessage =
         session.createTextMessage(requestString);
      requestMessage.setJMSReplyTo(replyQueue);
      MessageProducer messageProducer =
         session.createProducer(requestQueue);
      messageProducer.send(requestMessage);
   }
}
}
```

Here is the message-driven bean `RequestResponderOld` which receives request messages and sends responses:

```
@MessageDriven(mappedName = "jms/requestQueue")
public class RequestResponderOld implements MessageListener {

@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

public void onMessage(Message message) {

   try (Connection connection =
         connectionFactory.createConnection()){
      Session session = connection.createSession();

      // extract request from request message
      String request = ((TextMessage)message).getText();

      // extract temporary reply destination from request message
      Destination replyDestination = message.getJMSReplyTo();

      // prepare response
      TextMessage replyMessage =
         session.createTextMessage("Reply to: "+request);

      // send response
      MessageProducer messageProducer =
         session.createProducer(replyDestination);
      messageProducer.send(replyMessage);
   } catch (JMSException ex) {
      // log an error here
   }
}
}
```

11.4.10.2. Example using the simplified API

Here’s how the same example might look when using the simplified API:

There are two session beans involved in sending the request message. The first bean

The first session bean `RequestReplyNew` creates the creates the temporary reply queue, calls a second bean `SenderBeanNew` to send the request in a separate transaction and then listens for the reply:

```
@Stateless
@LocalBean
public class RequestReplyNew {

@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@EJB private SenderBeanNew senderBean;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String requestReplyNew(String request) throws JMSException {

   try (MessagingContext context =
         connectionFactory.createMessagingContext()) {
      TemporaryQueue replyQueue = context.createTemporaryQueue();

      // send the request message in a separate transaction
      // so use a separate bean
      // this call may throw JMSException
      senderBean.sendRequestNew(request,replyQueue);

      // now receive the reply, using the same connection
      // as was used to create the temporary reply queue
      SyncMessageConsumer consumer =
         context.createSyncConsumer(replyQueue);
      return consumer.receivePayload(String.class);
   }
}
}
```

The second session bean `SenderBeanNew` simply sends the request to the request queue in a separate transaction:

```
@Stateless
@LocalBean
public class SenderBeanNew {

@Resource(lookup = "jms/connectionFactory")
ConnectionFactory connectionFactory;

@Resource(lookup="jms/requestQueue")
Queue requestQueue;

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void sendRequestNew(
      String requestString, TemporaryQueue replyQueue)
      throws JMSException {
   try (MessagingContext context =
         connectionFactory.createMessagingContext()) {
      TextMessage requestMessage =
         context.createTextMessage(requestString);
      // this call may throw JMSException
      requestMessage.setJMSReplyTo(replyQueue);
      context.send(requestQueue,requestMessage);
   }
}
}
```

Here is the message-driven bean `RequestResponderNew` which receives request messages and sends responses:

```
@MessageDriven(mappedName = "jms/requestQueue")
public class RequestResponderNew implements MessageListener {

@Resource(lookup = "jms/connectionFactory")
    ConnectionFactory connectionFactory;

public void onMessage(Message message) {

   try (MessagingContext context =
         connectionFactory.createMessagingContext()){

      // extract request from request message
      // this may throw a JMSException
      String request = ((TextMessage)message).getText();

      // extract temporary reply destination from request message
      // this may throw a JMSException
      Destination replyDestination = message.getJMSReplyTo();

      // prepare response
      TextMessage replyMessage =
         context.createTextMessage("Reply to: "+request);

      // send response
      context.send(replyDestination,replyMessage);
   } catch (JMSException ex) {
      // log an error here
   }
}
}
```

Note that in this example it is not possible to eliminate the need to declare to catch `JMSException` since it uses methods on `Message` and `TextMessage` which throw `JMSException`.

11.4.10.3. Example using the simplified API and injection

Here's how the same example might look when using the simplified API with the MessagingContext created by injection:

There are two session beans involved in sending the request message. The first bean

The first session bean `RequestReplyNew` creates the creates the temporary reply queue, calls a second bean `SenderBeanNew` to send the request in a separate transaction and then listens for the reply:

```
@Stateless
@LocalBean
public class RequestReplyNew {

@Inject
@JMSConnectionFactory("jms/connectionFactory2")
private MessagingContext context;

@EJB private SenderBeanNew senderBean;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public String requestReplyNew(String request) throws JMSException {

   TemporaryQueue replyQueue = context.createTemporaryQueue();

   // send the request message in a separate transaction
   // so use a separate bean
   // this call may throw JMSException
   senderBean.sendRequestNew(request,replyQueue);

   // now receive the reply, using the same connection
   // as was used to create the temporary reply queue
   SyncMessageConsumer consumer =
      context.createSyncConsumer(replyQueue);
   return consumer.receivePayload(String.class);
}
}
```

The second session bean `SenderBeanNew` simply sends the request to the request queue in a separate transaction:

```
@Stateless
@LocalBean
public class SenderBeanNew {

@Inject
@JMSConnectionFactory("jms/connectionFactory")
private MessagingContext context;

@Resource(lookup="jms/requestQueue")
Queue requestQueue;

@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void sendRequestNew(
      String requestString, TemporaryQueue replyQueue)
      throws JMSException {

   TextMessage requestMessage =
      context.createTextMessage(requestString);
   // this call may throw JMSException
   requestMessage.setJMSReplyTo(replyQueue);
   context.send(requestQueue,requestMessage);
}
}
```

Here is the message-driven bean `RequestResponderNew` which receives request messages and sends responses:

```
@MessageDriven(mappedName = "jms/requestQueue")
public class RequestResponderNew implements MessageListener {

@Inject
@JMSConnectionFactory("jms/connectionFactory")
private MessagingContext context;

public void onMessage(Message message) {

   try {
      // extract request from request message
      // this may throw a JMSException
      String request = ((TextMessage)message).getText();

      // extract temporary reply destination from request message
      // this may throw a JMSException
      Destination replyDestination = message.getJMSReplyTo();

      // prepare response
      TextMessage replyMessage =
         context.createTextMessage("Reply to: "+request);

      // send response
      context.send(replyDestination,replyMessage);
   } catch (JMSException ex) {
      // log an error here
   }
}
}
```

Note that in this example it is not possible to eliminate the need to declare to catch `JMSException` since it uses methods on `Message` and `TextMessage` which throw `JMSException`.

A. Issues

A.1. Resolved Issues

A.1.1 JDK 1.1.x Compatibility

JMS is compatible with JDK 1.1.x.

A.1.1 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.

A.1.2 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 that 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).

A.1.3 Should JMS Specify a Set of JMS JavaBeans?

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

A.1.4 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).

A.1.5 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).

A.1.6 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.

A.1.7 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.

B. Change History

B.1. Version 1.0.1

B.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.

B.2. Version 1.0.2

The objective of JMS 1.0.2 is to correct errata in the JMS 1.0.1 specification and code that have been uncovered by implementors and users. It also contains many clarifications that resolve ambiguities found in the previous versions.

B.2.1 The Multiple Topic Subscriber Special Case

JMS 1.0.1 specified that in the special case of two topic subscribers on a session with overlapping subscriptions, a message that was selected by both would only be delivered to one. Implementation experience revealed that this case was better handled in the same way that overlapping subscriptions from different sessions are treated, so this special case has been removed.

B.2.2 Message Selector Comparison of Exact and Inexact Numeric Values

JMS 1.0.1 specified that message selectors did not support the comparison of exact and inexact numeric values. This conflicted with the requirement to support numeric promotion. This has been changed to support exact and inexact comparison.

B.2.3 Connection and Session Close

JMS 1.0.1 did not fully specify the sequence for closing a connection and its sessions. This sequence is now fully specified.

JMS 1.0.1 was ambiguous about whether or not calls to connection and session close returned immediately. Connection and session close now explicitly state that they block until message processing has been shutdown in an orderly fashion.

B.2.4 Creating a Session on an Active Connection

When a session is created on an active (as opposed to stopped) connection it is only possible to create at most a single asynchronous consumer for it. A more detailed discussion of this case is provided.

B.2.5 Delivery Mode and Message Retention

The effect that delivery mode has on message retention for a consumer has been clarified.

B.2.6 The ‘single thread’ Use of Sessions

Sessions are designed to minimize the need to write for multithreaded code in order to support the asynchronous consumption of messages. Clarification on the benefits and the programming model of this design have been added.

B.2.7 Clearing a Message’s Properties and Body

A clarification has been added that notes that clearing a message’s properties and clearing its body are independent.

B.2.8 Message Selector Numeric Literal Syntax

A note has been added that states that the numeric literal syntax is that specified by the Java language.

B.2.9 Comparison of Boolean Values in Message Selectors

A note has been added that only equality and inequality comparisons are supported.

B.2.10 Order of Messages Read from a Queue

A note has been added that explains that a client can read messages from a destination in an order different from the order they have been sent by using a selector that matches a later message and then using a selector that matches an earlier message.

B.2.11 Null Values in Messages

A note has been added that message values are allowed to be null.

B.2.12 Closing Constituents of Closed Connections and Sessions

There was some ambiguity about whether or not close needed to be called on all JMS objects. A note has been added that states that there is no need to close the sessions of a closed connection; and, there is no need to close the producers and consumers of a closed session.

B.2.13 The Termination of a Pending Receive on Close

JMS 1.0.1 did not describe how a pending message receive is terminated if its session or connection is closed. It is now specified that in this case receive returns a null message.

B.2.14 Incorrect Entry in Stream and Map Message Conversion Table

This table erroneously included a required conversion between char and String. This has been removed.

B.2.15 Inactive Durable Subscription

A note explaining that an inactive durable subscription is one that exists but does not at the time have a TopicSubscriber created for it.

B.2.16 Read-Only Message Body

The read-only semantics of received message bodies was documented in the *Message* javadoc but was not included in the spec. It has been added.

B.2.17 Changing Header Fields of a Received Message

When a message is received, its header field values may be changed; however, its property entries and its body are read-only. A note clarifying this has been added.

B.2.18 Null/Missing Message Properties and Message Fields

The result of accessing a null/missing value as a Java primitive type was previously not fully specified. This has clarified.

B.2.19 JMS Source Errata

Two methods required by the spec were left out of the source, the getJMSXPropertyNames method of ConnectionMetaData and the getExceptionListener method of Connection. These have been added.

The type of the time-to-live parameter of setTimeToLive and getTimeToLive methods of MessageProducer and the type of the default time-to-live constant were int and have been changed to long.

The close sequence of TopicRequestor and QueueRequestor did not agree with the order specified in the specification and this has been corrected.

The type of the parameter of the createTextMessage method that takes an input value was changed from StringBuffer to String.

The subscription name parameter was missing from the createDurableSubscription method of TopicConnection. It has been added.

B.2.20 JMS Source Java API documentation Errata

The correct end-of-message indicator for the readBytes method of BytesMessage is a return value of -1.

The setPriority method of MessageProducer should have a parameter named ‘priority’ not ‘deliveryMode’.

B.2.21 JMS Source Java API documentation Clarifications

Note that byte values are returned as *byte[]* not *Byte[]* by the *readObject* method of *StreamMessage* and the *getObject* method of *MapMessage*.

Note that the *acknowledge* method of *Message* acknowledges all messages received on that message’s session.

Note that the *InvalidClientIDException* is used for any client id value that a JMS provider considers invalid. Since client id value is JMS provider specific the criteria for determining a valid value is provider specific.

A note has been added to the *readBytes* method of *StreamMessage* and *BytesMessage* to describe how values that overflow the size of the input buffer are handled.

A note has been added that clarifies when *setClientID* method of Connection should be used.

Note that calling the *setMessageListener* method of *MessageConsumer* with a null value is equivalent to unsetting the *MessageListener*.

Note that the *unsubscribe* method of *TopicSession* should not be called to delete a durable subscription if there is a *TopicConsumer* currently consuming it.

Note that result of calling the *setMessageListener* method of *MessageConsumer* while messages are being consumed by an existing listener or the consumer is being used to synchronously consume messages is undefined.

Note the *createTopic* method of *TopicSession* and the *createQueue* method of *QueueSession* are used for converting a JMS provider specific name into a *Topic* or *Queue* object that represents an existing topic or queue by that name. These methods are not for creating the physical topic or queue. The physical creation of topics and queues are administrative tasks and are not done by JMS. The one exception is the creation of temporary topics and queues which is done using the *createTemporaryTopic* and *createTemporaryQueue* methods.

Note that the *setObject* method of *ObjectMessage* places a copy of the input object in a message.

Note that a connection is created in stopped mode and, for incoming messages to be delivered to the message listeners of its sessions, its *start* method must be called.

Documentation of *Message* default constants has been added.

Note the result of *readBytes* method of *StreamMessage* when the caller's *byte[]* buffer is smaller than the *byte[]* field value being read.

The documentation of *QueueRequestor* and *TopicRequestor* has been improved.

The *IllegalStateException* has been noted as a required exception for several more error conditions. They are: acknowledging a message received from a closed session; attempting to call the *recover* method of a transacted session; attempting to call any method of a closed connection, session, consumer or producer (with the exception of the *close* method itself); attempting to set a connection's client identifier at the wrong time or when it has been administratively configured.

B.3. Version 1.0.2b

The objective of version 1.0.2b of the JMS API Specification and Java API documentation is to correct errata in the JMS 1.02 Specification and the JMS 1.0.2a Java API documentation that have been uncovered by implementors and users.

Version 1.0.2b incorporates two sets of errata, which are marked by change bars in the Specification:

- Major errata and clarifications approved by a Java Community Process℠ program Maintenance Review that closed June 25, 2001.
- Minor errata formerly listed on the JMS documentation web page.

B.3.1 JMS API Specification, version 1.0.2: Errata and Clarifications

The following errata and clarifications have been incorporated into the Specification. They are listed in the order in which they occur in the Specification.

- Section 3.4.7 "JMSRedelivered": Change first paragraph to clarify when a provider must set this header field.
- Section 3.5.9 "JMS Defined Properties": Correct return value of *ConnectionMetaData.getJMSXPropertyNames* method.
- Section 3.8.1.1 "Message Selector Syntax": After the first sentence, add sentence about the interpretation of a message selector whose value is an empty string. In the third sub-bullet item under "Identifiers," add ESCAPE to the list of prohibited identifiers. In the fourth sub-bullet item, add sentence about data types of property values, and move description of the value of nonexistent properties referenced in a selector from last bullet item to here. Add sub-bullet item clarifying that data type conversions do not apply to properties used in message selectors. In the first sub-bullet item under "Comparison Operators," clarify the result of comparing non-like type values. At end of section, correct quotation marks in example.
- Section 3.10 "Changing the Value of a Received Message": Add paragraph clarifying the semantics of redelivering a message that was modified after being received.
- Section 3.12 "Provider Implementations of JMS Message Interfaces": Insert paragraph clarifying the handling of destinations for foreign message implementations.
- Section 4.4.12 "Duplicate Delivery of Messages" (formerly misnumbered 4.4.14): Add sentence about *JMSRedelivered* message header field.
- Section 4.5.2 "Asynchronous Delivery": Clarify explanation of redelivery for AUTO_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE acknowledgment modes.
- Section 4.10 "Reliability": Clarify meaning of PERSISTENT and NON_PERSISTENT delivery modes throughout section.
- Section 6.9 "TopicSession": Clarify redelivery of messages for durable and nondurable subscriptions.

  Rationale for this change: The scope of redelivery is the lifetime of a destination, not of the session that is consuming it. Each nondurable subscription is a different destination, and its lifetime is the session that creates it. Each temporary queue or topic is a different destination whose lifetime is the connection that creates it.
- Section 6.12 "Recovery and Redelivery": Clarify recoverability of messages for nondurable subscriptions.

  Rationale for this change: Update the Specification to meet the expectation that a nondurable subscriber performs the same as a durable subscriber as long as the nondurable subscriber is in existence. The original statement in the specification could be interpreted to mean that message recovery was optional for a nondurable subscriber. It would be valuable to have a lower quality of

service that did not require acknowledgement overhead, but a new mechanism should be provided to specify the lower quality of service option; the minimum quality of service required by the current specification should not be lowered.

- Section 7.3 "Standard Exceptions": In description of *IllegalStateException*, change “should” to “must”. In description of *MessageFormatException*, correct method names, and change “should” to “must” in last sentence.

B.3.2 JMS API Java API documentation, version 1.0.2a: Major Errata

The following items represent significant clarifications of the JMS API Java API documentation, version 1.0.2a. They are categorized as follows:

- Corrections of mistakes
- Reconciliations between the Specification and the Java API documentation

Less important clarifications to the Java API documentation are listed in Section 11.3.3 "JMS API Java API documentation, version 1.0.2a: Lesser Errata"JMS API Java API documentation, version 1.0.2a: Lesser Errata.

B.3.2.1. Corrections of Mistakes

In the following cases, the Java API documentation was in error and has been corrected:

- *BytesMessage* and *StreamMessage* interfaces: Correct discussion of modification of sent messages.
- *TemporaryQueue.delete* and *TemporaryTopic.delete* methods: Remove references to senders and publishers.

B.3.2.2. Reconciliations between the Specification and the Java API documentation

The following items update the Java API documentation to match the correct language in the Specification:

- *Message* interface: Correct description of getting values for unset property names to match Section 3.5.4 "Property Value Conversion". Remove incorrect bullet items about NULL values in arithmetic operations and BETWEEN operations.
- *Message.acknowledge* method: Clarify that the method applies to all consumed messages of the session.

  Rationale for this change: A possible misinterpretation of the existing Java API documentation for Message.acknowledge assumed that only messages received prior to “this” message should be acknowledged. The updated Java API documentation statement emphasizes that message acknowledgement is really a session-level activity and that this message is only being used to identify the session in order to acknowledge all messages consumed by the session. The acknowledge method was placed in the message object only to enable easy access to acknowledgement capability within a message listener’s onMessage method. This change aligns the specification

and Java API documentation to define Message.acknowledge in the same manner.

- *Message.getJMSTimestamp* and *MessageProducer.setDisableMessageTimestamp* methods: Correct descriptions of effect of disabling timestamps.
- *TopicSession.createSubscriber* and *TopicSession.createDurableSubscriber* methods: Correct **Throws:** lists.

B.3.3 JMS API Java API documentation, version 1.0.2a: Lesser Errata

The Java API documentation corrections listed in this section concern the application of logic from the Specification or elsewhere in the Java API documentation:

- Corrections to the Specification listed in Section 11.3.1 "JMS API Specification, version 1.0.2: Errata and Clarifications".
- Information in the Specification not previously reflected in the Java API documentation
- Information provided in some parts of the Java API documentation, but not in others where it also belongs.

4. Message interface: Add the corrections to Section 3.8.1.1 "Message Selector Syntax" to the section on message selectors.

   Also change the Java API documentation for the following methods to reflect these changes:

   ```
   QueueConnection.createConnectionConsumer
   QueueSession.createReceiver
   QueueSession.createBrowser
   TopicConnection.createConnectionConsumer
   TopicConnection.createDurableConnectionConsumer
   TopicSession.createSubscriber
   TopicSession.createDurableSubscriber
   QueueBrowser.getMessageSelector
   MessageConsumer.getMessageSelector
   ```

5. Correct the Java API documentation for the following methods to add *InvalidDestinationException* to the **Throws:** list, in accordance with Section 7.3 "Standard Exceptions":

   ```
   QueueConnection.createConnectionConsumer
   QueueRequestor.QueueRequestor
   TopicConnection.createConnectionConsumer
   TopicConnection.createDurableConnectionConsumer
   TopicRequestor.TopicRequestor
   ```

6. *TopicSession.createDurableSubscriber* method: Change the description of the two-argument form to accord with the description of the one-argument form of the *TopicSession.createSubscriber* method.

7. *QueueSender* and *TopicPublisher* interfaces: Add clarifications from Section 3.9 "Access to Sent Messages" and Section 3.4.11 "How Message Header Values Are Set". Add

UnsupportedOperationException to *send* and *publish* method **Throws:** lists where relevant.

8. *QueueReceiver* interface: Add language from Section 5.8 "QueueReceiver".

9. *IllegalStateException* and *MessageFormatException* classes: Add corrections from Section 7.3 "Standard Exceptions".

B.4. Version 1.1

This section describes the changes to the JMS specification for version 1.1. The changes include API unification of messaging domains, specification clarifications, and several additional methods.

B.4.1 Unification of messaging domains

This maintenance release addresses the unification of the programming interfaces for the Point-to-Point and Pub/Sub messaging domains in the Java Message Service (JMS) API. In the 1.0.2b version of the JMS specification, the client programming model made a strong distinction between these two domains. One consequence of domain separation is that actions from the Point-to-Point domain and the Pub/Sub domain could not be used in the same transaction.

In this version of the interfaces, methods have been added to support the ability to include PTP and Pub/Sub messaging in the same transaction. In addition, domain unification simplifies the client programming model, so that the client programmer can use a simplified set of APIs to create an application.

The scope of a transaction in JMS is on a per *Session* basis. To add the ability to work across both domains, a number of methods have been added the *javax.jms.Session* interface. Adding these methods supports the creation of *javax.jms.MessageConsumer*s and *javax.jms.MessageProducers* for either domain at the *Session* level, and supports sending and receiving message using either domain within the same *Session*.

For example, using these proposed methods, a JMS client can create a transacted *Session*, and then receive messages from a *Queue* and send messages to a *Topic* within the same transaction. In JMS 1.0.2b, this was not possible.

At the same time, the semantic differences between the two domains are retained. While it is possible to support actions on PTP and Pub/Sub destinations within the same session, the semantic differences that cause different behaviors within those domains are retained.

B.4.2 JMS API Specification, version 1.1: Domain Unification

The following is a list of changes that were added to the JMS specification to describe the requirements related to domain unification. They are listed in the order in which they appear in the specification.

- Section 1.2.3.3 "JMS Domains" describes the backward compatibility relationship between earlier versions of the JMS specification and version 1.1 of the JMS specification.

- Section 2.4 "Two Messaging Styles" and Section 2.5 "JMS Interfaces" add information about the two message domains, and a table that

describes the relationship between the JMS common interfaces and the domain-specific interfaces that are derived from the JMS common interfaces.

- Section 4.4 "Session" updates the list of the activities that can be performed in a *Session* to include that it a *Session* is a factory for *MessageProducers* and *MessageConsumers*, and is a factory for *TemporaryTopics* and *TemporaryQueues*.
- Section 4.5 "MessageConsumer" adds that a *MessageConsumer* can be created from a *Session*.
- Section 4.11 "Method Inheritance across Messaging Domains" is a new section that describes how to handle methods that are not appropriate to a specific messaging domain.
- Section 5.1 "Overview" adds a table that describes the relationship between the JMS common interfaces and the domain-specific interfaces that are derived from the JMS common interfaces.
- Section 5.4 "TemporaryQueue" adds that the lifespan of the *TemporaryQueue* is the life of the *Connection* or *QueueConnection* that creates it. Previously, only *QueueConnection* was mentioned, but *TemporaryQueues* can now be created from *Connections*.
- Section 5.9 "QueueBrowser" adds that a *QueueBrowser* can be created from *Session*, as well as from the *QueueSession*.
- Section 6.1 "Overview" adds a table that describes the relationship between the JMS common interfaces and the domain-specific interfaces that are derived from the JMS common interfaces.
- Section 6.6 "TemporaryTopic" adds that the lifespan of the *TemporaryTopic* is the life of the *Connection* or *TopicConnection* that creates it. Previously, only *TopicConnection* was mentioned, but a *TemporaryTopic* can now be created from a *Connection*.
- Section 6.10 "TopicPublisher" adds a comment that messages can be sent to a *Topic* either using a *TopicPublisher* or a *MessageProducer*.
- Section 6.11.1 "Durable TopicSubscriber" adds that durable *TopicSubscribers* can be created either relative to a *TopicConnection* or a *Connection*. The *unsubscribe* method for the durable Topic subscription can also be used either at the *TopicConnection* or *Connection* level.
- Section 8.6 "JMS Application Server Interfaces" adds a table that shows the relationships between the JMS common interfaces and the domain-specific interfaces that are derived from the JMS common interfaces.
- Chapter 9 "JMS Example Code" has been rewritten to reflect the use of the JMS common interfaces. It now uses the new methods associated with domain unification.

B.4.3 JMS API Specification, version 1.1: Updates and Clarifications

The following are additional updates to the JMS specification. They include both updated material, and clarifications of existing material. They are listed in the order in which they appear in the specification.

B.4.3.1. Updates to Introduction, Added Figures

These changes reflect new information about how the JMS API relates to other technologies, and what's new in this version of the specification. It also reflects the addition some diagrams to assist in the reading of the specification.

- Section 1.4 "Relationship to Other Java APIs" is updated with the current status of the relationship of the JMS specification to the JDBC API, the EJB API, and the JTA API.
- Section 1.4 "Relationship to Other Java APIs" has two new sections: Section 1.4.7 "Java Platform, Enterprise Edition (Java EE) " and, Section 1.4.8 "". These new sections describe how JMS relates to the J2EE platform.
- Section 1.5 "What is New in JMS 1.1?" is a new section that summarizes the changes in this maintenance release.
- Section 2.3 "Administration" contains a new figure which illustrates the relationship between the Administered objects and other elements of the JMS system.
- Section 2.5 "JMS Interfaces" adds a figure which illustrates the relationships among the basic JMS elements.

B.4.3.2. Clarifications

- Section 3.8.1.1 "Message Selector Syntax" clarifies when semantic checking may occur. At the end of that section, there is a requirement that JMS providers must check syntactic correctness at the time that a message selector is presented. This statement has been updated to note that JMS providers are also permitted to check semantic correctness at that time, although not all semantic correctness can be verified at that time.
- Section 4.3.8 "ExceptionListener" adds a clarification that if no *ExceptionListener* is registered, the *getExceptionListener* method should return null.
- Section 4.4.6 "Conventions for Using a Session" adds a clarification that use of the JTS aware interfaces described in Chapter 8 are not intended for use in the client JMS program. Use of these interfaces may create nonportable software, as the JTS aware interfaces described in Chapter 8 are optional.
- Section 4.4.9 "Multiple Sessions" adds a further clarification that the JMS specification does not assign a semantic meaning to multiple *QueueReceiver*s consuming from one *Queue*.
- Section 6.12 "Recovery and Redelivery" adds an additional clarification that setting the delivery mode to PERSISTENT will not change the delivery model for messages sent to a non durable subscriber. This is already described in Section 6.15, “Reliability”.
- Section 8.1 "Overview" adds a clarification that JMS client program use of the JTS aware interfaces described in Chapter 8 may create non-portable code.

B.4.4 JMS API Java API documentation, version 1.1: Domain Unification

In order to support domain unification a number of existing interfaces were updated with additional methods.Table 11-1 lists the interfaces that were changed, and the new methods for those interfaces.

Table 11.1 New JMS API methods for domain unification

Interface	New Methods
Connection	`ConnectionConsumer createConnectionConsumer (Destination destination, String messageSelector, ServerSessionPool sessionPool, int maxMessages) throws JMSException;`
	`ConnectionConsumer createDurableConnectionConsumer(Topic topic,String subscriptionName, String messageSelector, ServerSessionPool sessionPool, int maxMessages) throws JMSException;`
Connection Factory	`Connection createConnection() throws JMSException;`
	`Connection createConnection(String userName, String password) throws JMSException;`
Message Producer	`Destination getDestination() throws JMSException;`
	`void send(Message message) throws JMSException;`
	`void send(Message message, int deliveryMode, int priority, long timeToLive) throws JMSException;`
	`void send(Destination destination, Message message) throws JMSException;`
Session	`MessageProducer createProducer (Destination destination) throws JMSException;`
	`MessageConsumer createConsumer (Destination destination) throws JMSException;`
	`MessageConsumer createConsumer (Destination destination, java.lang.String messageSelector) throws JMSException;`
	`MessageConsumer createConsumer (Destination destination, java.lang.String messageSelector, boolean NoLocal) throws JMSException;`

Interface	New Methods
	`TopicSubscriber createDurableSubscriber (Topic topic, java.lang.String name) throws JMSException;`
Session	`TopicSubscriber createDurableSubscriber( Topic topic, java.lang.String name, java.lang.String messageSelector, boolean noLocal) throws JMSException;`
	`QueueBrowser createBrowser(Queue queue) throws JMSException;`
	`QueueBrowser createBrowser(Queue queue, String messageSelector) throws JMSException;`
	`Queue createQueue (String name) throws JMSException;`
	`Topic createTopic (String name) throws JMSException;`
	`TemporaryTopic createTemporaryTopic() throws JMSException;`
	`TemporaryQueue createTemporaryQueue() throws JMSException;`
	`void unsubscribe(String string name);`
XAConnection	`XASession createXASession() throws JMSException;`
XAConnection Factory	`XAConnection createXAConnection() throws JMSException;`
	`XAConnection createXAConnection( String userName, String password) throws JMSException;`
XASession	`Session getSession() throws JMSException;`

In addition, explanatory material was added to describe the relationship between the new methods and existing interfaces and methods. The interfaces that have been changed to describe these relationships are:

```
Queue
QueueBrowser
QueueConnection
QueueConnectionFactory
QueueReceiver
QueueSender
QueueSession
Session
TemporaryQueue
TemporaryTopic
Topic
TopicConnection
```

```
TopicConnectionFactory
TopicPublisher
TopicSubscriber
TopicSession
```

B.4.5 JMS API documentation, version 1.1: Changes

The Java API documentation also includes updates that reflect new features or additional clarifications.

B.4.5.1. New Methods

1. BytesMessage.getBodyLength

   ```
   long getBodyLength() throws JMSException
   ```

   This method permits the programmer to determine he size of the *BytesMessage* body and, if necessary, allocate a bytes array to copy the body.

2. Session.getAckowledgeMode

   ```
   int getAcknowledgeMode() throws JMSException
   ```

   This method returns the currently set value of how a *Session* acknowledges messages. This method is added for completeness, because in previous versions of the specification there was no way to get this value.

B.4.5.2. Clarifications

1. Connection.getExceptionListener

   If *Connection.getExceptionListener*is called, and no *ExceptionListener* is registered, the JMS provider must return null. A client program may optionally associate an *ExceptionListener* with a Connection. If no *ExceptionListener* is associated with the *ConnectionListener*, a null should be returned. Previously, the Java API documentation did not specify what the return value should be.

2. MapMessage

   A clarification to the set methods of *MapMessage*, stating if the name parameter is null or an empty string, the method must throw the error *InvalidArgumentException*.

   The rationale is that each of the named elements in the `MapMessage` should have names that are not null or blank.

   Methods affected:

   ```
   MapMessage.setBoolean
   MapMessage.setByte
   MapMessage.setBytes
   MapMessage.setChar
   MapMessage.setDouble
   MapMessage.setFloat
   MapMessage.setInt
   MapMessage.setLong
   MapMessage.setObject
   MapMessage.setShort
   MapMessage.setString
   ```

3. Message

   A clarification to the set property methods of *Message* states if the property name parameter is null or empty string, the method must throw the error *InvalidArgumentException*.

   The rationale is that each of the properties in the *Message* should have names that are not null or blank.

   Methods affected:

   ```
   Message.setBooleanProperty
   Message.setByteProperty
   Message.setDoubleProperty
   Message.setFloatProperty
   Message.setIntProperty
   Message.setLongProperty
   Message.setObjectProperty
   Message.setShortProperty
   Message.setStringProperty
   ```

4. TextMessage

   A comment in the description for *TextMessage* that indicated that XML might become popular has been changed to state that *TextMessage* can be used to send XML messages.

5. XA interfaces

   The descriptions of the XA interfaces now state that those interfaces are primarily used by JMS providers, and are optional; that is, the JMS provider is not required to support them. Use of the XA interfaces by a client program may lead to non-portable code.

   Interfaces affected:

   ```
   XAConnection
   XAConnectionFactory
   XAQueueConnection
   XAQueueConnectionFactory
   XAQueueSession
   XASession
   XATopicConnection
   XATopicConnectionFactory
   XATopicSession
   ```

6. The descriptions of *QueueSession, TopicSession,* and *QueueConnection* now list methods that must not be called through these interfaces. These methods are a result of inheritance, but are not appropriate to be called from domain-specific interfaces. These are also described in Section 4.11 "Method Inheritance across Messaging Domains". If they are invoked, an IllegalStateException must be thrown.

Table 11.2 Domain Dependent Interfaces

Interface	**Method**
QueueConnection	createDurableConnectionConsumer
QueueSession	createDurableSubscriber

	createTemporaryTopic
	createTopic
	unsubscribe
TopicSession	createQueueBrowser
	createQueue
	createTemporaryQueue

B.5. Version 2.0

All changes made for JMS 2.0 are represented by individual issues in the JMS specification issue tracker at http://http://java.net/jira/browse/JMS_SPEC. The appropriate issue number (e.g. JMS_SPEC-64) is given for each change below.

B.5.1 Re-ordering of chapters

Chapter 10 "Issues" and chapter 11 "Change History" have become appendices A and B and moved to the end of the specification.

New chapters 10 "Use of JMS API in Java EE applications" and 11 "Simplified JMS API" have been added.

B.5.2 JMS providers must implement both P2P and Pub-Sub (JMS_SPEC-50)

The specification has been amended to state that a JMS provider must implement both point-to-point messaging (queues) and publish-subscribe messaging (topics). This was already required by the Java EE 6 specification, section EE.2.7, but was not previously required by the JMS specification itself.

Section 1.3 "What Is Required by JMS" has therefore been updated to delete the sentence that states " Providers of JMS point-to-point functionality are not required to provide publish/subscribe functionality and vice versa".

B.5.3 Use of JMS API in Java EE applications (JMS_SPEC-45 and JMS_SPEC-27)

A new chapter 12 "Use of JMS API in Java EE applications" has been added. This chapter incorporates and clarifies various additional requirements which were previously only described in the Java EE and EJB specifications. Section 12.2 "Restrictions on the use of JMS API in the Java EE web or EJB container" includes a list of methods which may not be used in a Java EE web or EJB container and section 12.3 "Behaviour of JMS sessions in the Java EE web or EJB container" clarifies how the arguments to `createSession` are mostly ignored when used in a Java EE web or EJB container.

Section 1.4.7 "Java Platform, Enterprise Edition (Java EE)" has been updated to refer to Java EE 7 rather than J2EE 1.3. A reference has also been added to the new chapter 12 "Use of JMS API in Java EE applications".

Section 1.4.8 "" has been deleted. It is superseded by the new chapter 12 "Use of JMS API in Java EE applications".

B.5.4 New methods to create a session (JMS_SPEC-45)

The Connection method `createSession(boolean transacted, int acknowledgeMode)` has sometimes been a cause of confusion because if the `transacted` argument is set to `true` then the `acknowledgeMode` argument is ignored but must still be given a value.

To simplify application code a new `Connection` method `createSession(int sessionMode)` has been added which provides the same functionality as the previous method but with a single argument.

Examples 9.1.4 "Creating a Session" and 9.3.3.1 "Creating a Durable Subscription" have been updated to use this new method.

In addition, a second new Connection method `createSession()` has been added. This has no arguments and is intended for use in a Java EE web or EJB container in the case when there is an active JTA transaction, when the `sessionMode` supplied to `createSession(int sessionMode)` is ignored.

B.5.5 Batch delivery (JMS_SPEC-36)

A new feature has been added to allow messages to be delivered in batches to an asynchronous consumer. Messages are delivered to a application-defined `BatchMessageListener` instead of a `MessageListener`.

Section 4.12 "Batch delivery" has been added to describe this new facility and section 4.5.2 "Asynchronous Delivery" has been modified to reflect this. Other references to `MessageListener` objects in the specification have been updated to reflect this change. In general, references to "MessageListeners" have been changed to "message listeners" to reflect that they might be either `MessageListener` or `BatchMessageListener` objects.

B.5.6 Delivery delay (JMS_SPEC-44)

A new feature "delivery delay" has been added which allows a producing client to specify the earliest time when a provider may make the message visible on the target destination and available for delivery to consumers.

A new section 4.13 "Delivery delay" and a corresponding new section 3.4.13 "JMSDeliveryTime" have been added to describe this new feature. Section 4.4.10.2 "Order of Message Sends" has been updated to state that messages with a later delivery time may be delivered after messages with an earlier delivery time.

Section 4.4.11 "Message Acknowledgment" has been updated to state that when a session's `recover` method is called the messages it now delivers may be different from those that were originallydelivered due to the delivery of messages which could not previously be delivered as they had not reached their specified delivery time.

Section 4.6 "MessageProducer" has been updated to mention that a client may now define a default delivery delay for messages sent by a producer.

B.5.7 Sending messages asynchronously (JMS_SPEC-43)

New `send` methods have been added to `MessageProducer` which allow messages to be sent asynchronously. These methods return immediately and perform the send in a separate thread without blocking the calling

thread. When the send is complete, a callback method is invoked on an object supplied by the caller. Similar methods are available on the new `MessagingContext` interface.

Section 4.6 "MessageProducer" has been extended to describe these additional `send` methods..

B.5.8 Use of AutoCloseable (JMS_SPEC-53)

The `Connection`, `Session`, `MessageProducer`, `MessageConsumer` and `QueueBrowser` interfaces have been modified to extend the `java.lang.Autocloseable` interface. This means that applications can create these objects using a Java SE 7 try-with-resources statement which removes the need for applications to explicitly call `close()` when these objects are no longer required.

The new MessagingContext and SyncMessageConsumer interfaces also extend the `java.lang.Autocloseable` interface.

Ssections 4.3.5 "Closing a Connection" and 4.4.1 "Closing a Session" and the new section 11.2.6 "Closing the MessagingContext" all explain that the use of a try-with-resources statement makes it easier to ensure that these objects are closed after use.

The example in section 9.1.3 "Creating a Connection" has been extended to add a second example which uses the the try-with-resources statement.

B.5.9 JMSXDeliveryCount (JMS_SPEC-42)

The existing message property `JMSXDeliveryCount` has been made mandatory. It was previously optional. This means that JMS providers must set this property when a message is delivered to the number of times the message has been delivered.

A new section 3.5.11 "JMSXDeliveryCount" has been added which describes this property and explains how it is not required to be guaranteed in all possible cases, such as after a server failure.

Section 3.5.9 "JMS Defined Properties" has been updated accordingly. Some of the wording in this section has been rearranged to reflect the fact that some properties are optional but that one (`JMSXDeliveryCount`) is now mandatory. A clarification has been added to state that the effect of setting a message selector on a property (such as `JMSXDeliveryCount`) which is set by the provider on receive is undefined.

Section 3.4.7 "JMSRedelivered" has been amended to mention the JMSXDeliveryCount property as well.

Section 4.4.11 "Message Acknowledgment": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well.

4.4.12 "Duplicate Delivery of Messages": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well..

4.5.2 "Asynchronous Delivery": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well.

4.10 "Reliability": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well.

B.5.10 Client ID optional on Durable subscriptions (JMS_SPEC-39)

It is no longer mandatory for the client identifier to be set when creating or activating a durable subscription.

The javadocs for the various methods which create a durable subscription have updated to reflect this change and rewritten to improve clarify.

In addition the following sections have been updated:

Section 4.3.2 "Client Identifier" has been changed to state that its use in identifying a durable subscription is optional.

Section 6.3 "Durable Subscription" has been amended by adding a cross-reference to Section 6.11.1 "Durable TopicSubscriber" at the end.

Section 6.11.1 "Durable TopicSubscriber" has been completely rewritten. This section was essentially a copy of the javadoc comment on the `createDurableSubscriber` methods and since those javadoc comments have been rewritten to make them clearer this section has been rewritten as well. The only substantive change is to state that the use of client identifier in identifying a durable subscription is optional. Cross-references to Section 6.3 "Durable Subscription" and Section 4.3.2 "Client Identifier" have been added at the end.

B.5.11 New createDurableConsumer methods (JMS_SPEC-51)

The Session interface has been extended to add two `createDurableConsumer` methods which return a `MessageConsumer`.

These are intended to replace the existing `createDurableSubscription` methods which return a `TopicSubscriber`. A TopicSubscriber is a domain-specific interface whose use has been discouraged since the domain-independent interfaces were introduced in JMS 1.1.

Section 6.11.1 "Durable TopicSubscriber" has been updated to reflect this.

B.5.12 Simplified API (JMS_SPEC-64)

Two new objects `MessagingContext` and `SyncMessageConsumer` have been added which together combine the functionality of the existing `Connection`, `Session`, `MessageProducer` and `MessageConsumer` objects. This provides an alternative API for using JMS which is referred to in this specification as the "simplified API".

`MessagingContext` objects may be created using new methods on `ConnectionFactory`. Java EE applications may alternatively create `MessagingContext` objects using injection.

The goals of the simplified API are described in 13.1 "Goals of the simplified API" and a summary of the API is given in section 13.2 "Key features of the simplified API".

Developers now have a choice as to whether to use the "standard API" (the `Connection`, `Session`, `MessageProducer` and `MessageConsumer` objects) or the "simplified API" (the `MessagingContext` and `SyncMessageConsumer` objects).

The two APIs are intended to offer identical functionality. The standard API is not deprecated and will remain part of JMS indefinitely.

Section 13.3 "Examples using the simplified API" contains a number of examples which compare the use of the simplified amd standard APIs in a number of simple Java EE and Java SE use cases.

B.5.13 Clarification: message may be sent using any session (JMS_SPEC-52)

The specification and javadocs have been clarified to make it clear that a message may be sent using any session, not just the session used to create the message.

Section 4.4.5 "Optimized Message Implementations" has been updated accordingly.

B.5.14 Clarification: use of ExceptionListener (JMS_SPEC-49)

Section 4.3.8 "ExceptionListener" has been amended to clarify how an ExceptionListener is used:

- The existing text which states that a connection "serializes execution of its ExceptionListener" has been extended to explain what this means.
- A note has been added to state that there are no restrictions on the use of the JMS API by the listener's `onException` method.

In addition, the following changes to javadoc comments have been made:

- The javadoc comments for the `stop` and `close` methods on the `Connection` interface have been amended to clarify that, if an exception listener for the connection is running when `stop` or `close` are invoked, there is no requirement for the `stop` or `close` call to wait until the exception listener has returned before it may return.
- Similarly, the javadoc comment for the `close` method on the `Session` interface has been amended to clarify that, if an exception listener for the session's connection is running when `close` is invoked, there is no requirement for the `close` call to wait until the exception listener has returned before it may return.
- The javadoc comments for the `stop` and `close` methods on the `MessagingContext` interface have been amended to clarify that, if an exception listener for the messaging context's connection is running when `stop` or `close` are invoked, there is no requirement for the `stop` or `close` call to wait until the exception listener has returned before it may return.

B.5.15 Clarification: use of stop or close from a message listener (JMS_SPEC-48)

The specification has been clarified to state that a message listener must not call its own connection's `stop` or `close` methods, or its own session's `stop` method. This is because the specification requires that these methods should not return until any message listeners have returned and so calling them from a message listener would lead to deadlock.

A new requirement has been added that the JMS provider throw a `javax.jms.IllegalStateException` in such cases.

The following sections have been updated accordingly:

- Section 4.3.4 "Pausing Delivery of Incoming Messages"
- Section 4.3.5 "Closing a Connection"
- Section 4.4.1 "Closing a Session"

The following javadoc comments have been updated accordingly:

- The `Connection` method `stop`
- The `Connection` method `close`
- The `Session` method `close`
- The `MessagingContext` method `stop`
- The `MessagingContext` method `close`

B.5.16 Clarification: use of noLocal when creating a durable subscription (JMS_SPEC-65)

The specification has been amended to clarify the effect of setting the `noLocal` argument when creating a durable subscription.

Section 6.11 "TopicSubscriber" has been reworded to state that "When a non-durable subscription is created on a topic, the `noLocal` argument may be used to specify that the subscriber must not receive messages published to the topic by its own connection."

Section 6.11.1 "Durable TopicSubscriber" has been reworded to state that "When a durable subscription is created on a topic, the `noLocal` argument may be used to specify that messages published to the topic by its own connection must not be added to the durable subscription."

B.5.17 Clarification: message headers that are intended to be set by the JMS provder (JMS_SPEC-34)

The specification has been clarified to state that the following methods on `Message` are not for use by client applications and setting them does not have any effect:

`setJMSDeliveryMode`, `setJMSExpiration`, `setJMSPriority`, `setJMSMessageID`, `setJMSTimestamp`, `setJMSRedelivered`, `setJMSDeliveryTime` (new header property: see section 11.5.4).

Section 3.4.11How Message Header Values Are Set" has been extended to explain this.