Java Message Service

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

Version 2.0 (Proposed final draft RC2)

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

Nigel Deakin
Oracle
(Version 2.0)

20 February 2013

For information about how to contribute to the JMS specification visit http://jms-spec.java.net

License

Specification: JSR-343 Java Message Service 2.0 ("Specification")
Version: 2.0
Status: Proposed final draft RC2
Release: 20 February 2013

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

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 Platform, Enterprise Edition (Java EE). See section 1.4 "Relationship to other Java APIs" for additional requirements for JMS when it is integrated into a Java EE 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 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 specifies how the JMS API participates in bean-managed or container-managed transactions. The EJB specification restricts how to use JMS interfaces when implementing EJB clients. Refer to the EJB 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.

The JTA specification also defines a scope `@TransactionScope` which is referred to in section 12.4.4 "Scope of injected JMSContext objects".

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 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".

1.4.8. Contexts and dependency injection (CDI)

This specification defines how `JMSContext` objects may be injected into Java EE web or EJB applications. See section 12.4 "Injection of JMSContext objects" for more information. The CDI (Contexts and dependency injection) specification defines the concepts and technology on which this is based.

1.5. 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 A.1 "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:

- 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.

The following change has been made to aid scalability:

- Applications are now permitted to create multiple consumers on the same durable or non-durable topic subscription. In previous versions of JMS only a single consumer was permitted.

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 statement.

- 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.

- Although setting client ID remains mandatory when creating an unshared durable subscription, it is optional when creating a shared durable subscription.

- A new method `getBody` has been added to allow an application to extract the body directly from a `Message` without the need to cast it first to an appropriate subtype.

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.

A new chapter has been added which adds a new recommendation for a JMS provider to include a resource adapter, and which defines a number of activation configuration properties.

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

2.3. Administration

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

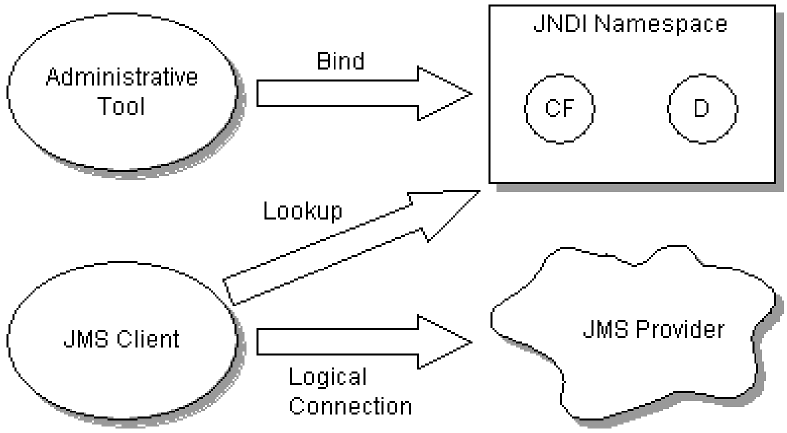
If JMS clients are to be portable, they must be isolated from these proprietary aspects of a provider. This is done by defining JMS administered objects that are created and customized by a provider's administrator and later used by clients. The client uses them through JMS interfaces that are portable. 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) or 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 4 “Messaging domains”.

2.5. JMS APIs

For historical reasons JMS offers four alternative sets of interfaces for sending and receiving messages.

JMS 1.0 defined two **domain-specific APIs**, one for point-to-point messaging (queues) and one for pub/sub (topics). Although these remain part of JMS for reasons of backwards compatibility they should be considered to be completely superseded by the later APIs.

JMS 1.1 introduced a new unified API which offered a single set of interfaces that could be used for both point-to-point and pub/sub messaging. This is referred to here as the **classic API**.

JMS 2.0 introduces a **simplified API** which offers all the features of the classic API but which requires fewer interfaces and is simpler to use.

Each API offers a different set of interfaces for connecting to a JMS provider and for sending and receiving messages. However they all share a common set of interfaces for representing messages and message destinations and to provide various utility features.

All interfaces are in the `javax.jms` package.

2.6. Interfaces common to multiple APIs

The main interfaces common to multiple APIs are as follows:

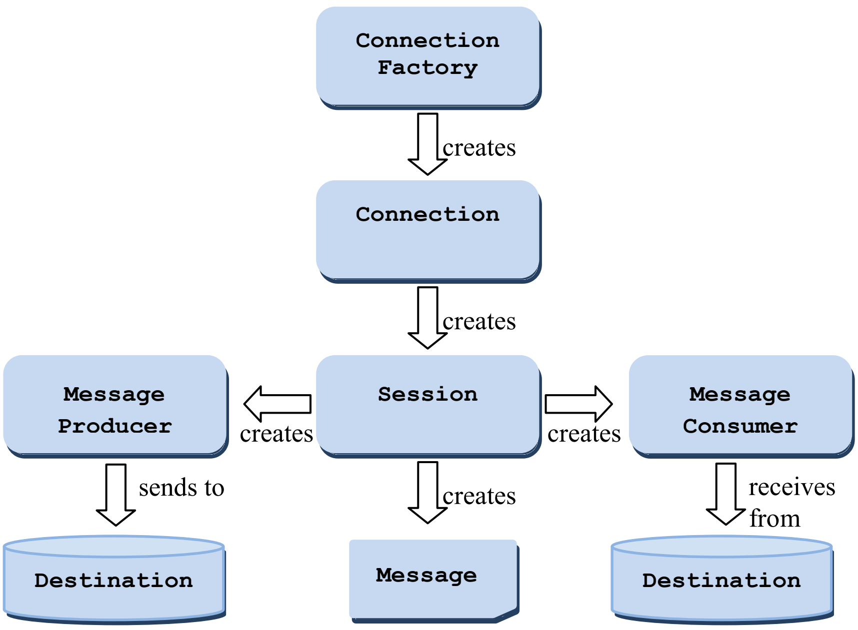
- `Message`, `BytesMessage`, `MapMessage`, `ObjectMessage`, `StreamMessage` and `TextMessage` – a message sent to or received from a JMS provider.
- `Queue` – an administered object that encapsulates the identity of a message destination for point-to-point messaging
- `Topic` – an administered object that encapsulates the identity of a message destination for pub/sub messaging.
- `Destination` - the common supertype of `Queue` and `Topic`

2.7. Classic API interfaces

The main interfaces provided by the classic API are as follows:

- `ConnectionFactory` - an administered object used by a client to create a Connection. This interface is also used by the simplified API.
- `Connection` - an active connection to a JMS provider
- `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 queue or topic
- `MessageConsumer` - an object created by a `Session` that is used for receiving messages sent to a queue or topic

Figure 2-2 Overview of classic API



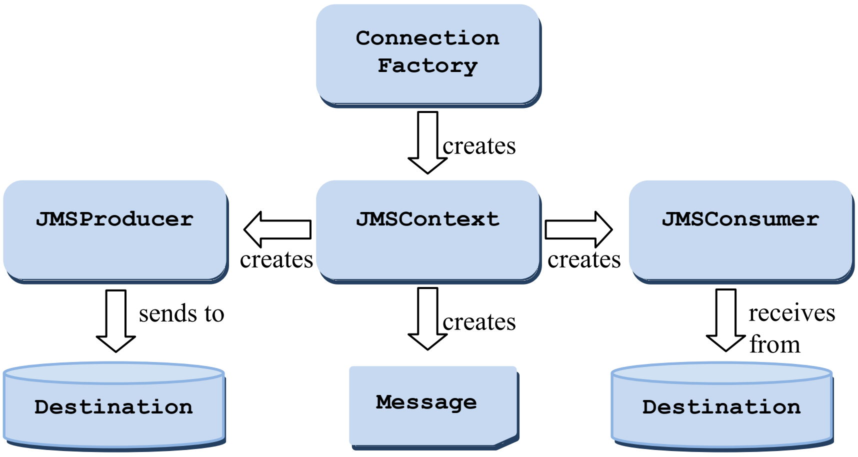
2.8. Simplified API interfaces

The simplified API provides the same messaging functionality as the classic API but requires fewer interfaces and is simpler to use.

The main interfaces provided by the simplified API are as follows:

- `ConnectionFactory` - an administered object used by a client to create a `Connection`. This interface is also used by the classic API.
- `JMSContext` - an active connection to a JMS provider and a single-threaded context for sending and receiving messages
- `JMSProducer` - an object created by a `JMSContext` that is used for sending messages to a queue or topic
- `JMSConsumer` - an object created by a `JMSContext` that is used for receiving messages sent to a queue or topic

Figure 2-3 Overview of simplified API



In the simplified API a single `JMSContext` object encompasses the behaviour which in the classic API is provided by two separate objects, a `Connection` and a `Session`. Although this specification refers to the `JMSContext` as having an underlying "connection" and "session", the simplified API does not use the `Connection` and `Session` interfaces.

2.8.1. 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` objects 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 environments which support it.

- To provide the option to send and receive the message body 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 that is unavailable in the simplified API.

- To be an alternative to, but not a replacement for, the classic API. The classic API remains and is not deprecated. Developers who are familiar with the classic API, or who prefer it, may continue to use the classic API.

2.8.2. Key features of the simplified API

The main object in the simplified API is `javax.jms.JMSContext`. This combines in a single object the functionality of several separate objects from the classic API. In particular it combines the functionality of a `Connection` and a `Session` in a single object.

Although the `JMSContext` does not expose constituent `Connection` and `Session` objects to applications, the concepts of connection and session remain important. A `Connection` represents a physical link to the JMS server, a `Session` represents a single-threaded context for sending and receiving messages, and a `JMSContext` represents both.

Applications that send messages will use the `JMSContext` method `createProducer` to create a `javax.jms.JMSProducer` object. This provides an API to send messages. Although it provides similar functionality to an anonymous `MessageProducer` (one with no destination specified) it provides a more convenient API for configuring delivery options, message properties and message headers.

Applications that consume messages will use one of several methods on `JMSContext` to create a `javax.jms.JMSConsumer` object. This provides a similar API to a `MessageConsumer` for consuming messages from a particular queue or topic. Messages may be consumed either synchronously or asynchronously, except in a Java EE web or EJB container where messages may be consumed only synchronously.

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 `JMSContext` 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. This allows the same physical connection to be used in multiple threads simultaneously. Such applications which require multiple sessions to be created on the same connection should use the factory methods on the `ConnectionFactory` interface to create the first `JMSContext` and then use the `createContext` method on `JMSContext` to create additional `JMSContext` objects that use the same connection:

To simplify application code, methods on `JMSContext` throw unchecked exceptions rather than checked exceptions.

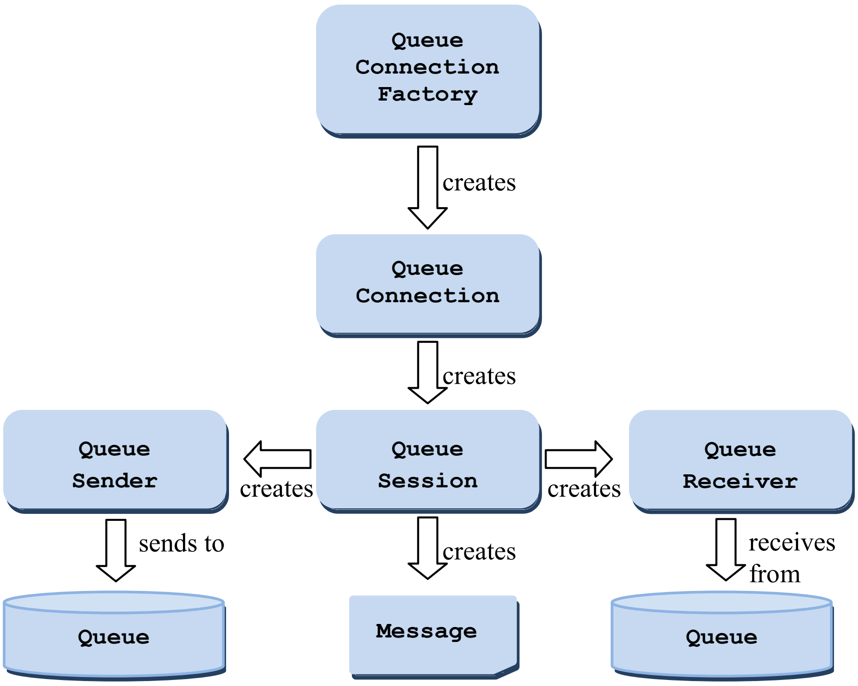
2.9. *Legacy domain-specific API interfaces*

Although the domain-specific API remains part of JMS for reasons of backwards compatibility it should be considered to be completely superseded by the classic and simplified APIs.

The main interfaces provided by the domain-specific API for **point-to-point** messaging are as follows:

- `QueueConnectionFactory` - an administered object used by a client to create a `QueueConnection`.
- `QueueConnection` - an active connection to a JMS provider
- `QueueSession` - a single-threaded context for sending and receiving messages
- `QueueSender` - an object created by a `QueueSession` that is used for sending messages to a queue
- `QueueReceiver` - an object created by a `QueueSession` that is used for receiving messages sent to a queue

Figure 2-4 Overview of legacy point-to-point-specific API

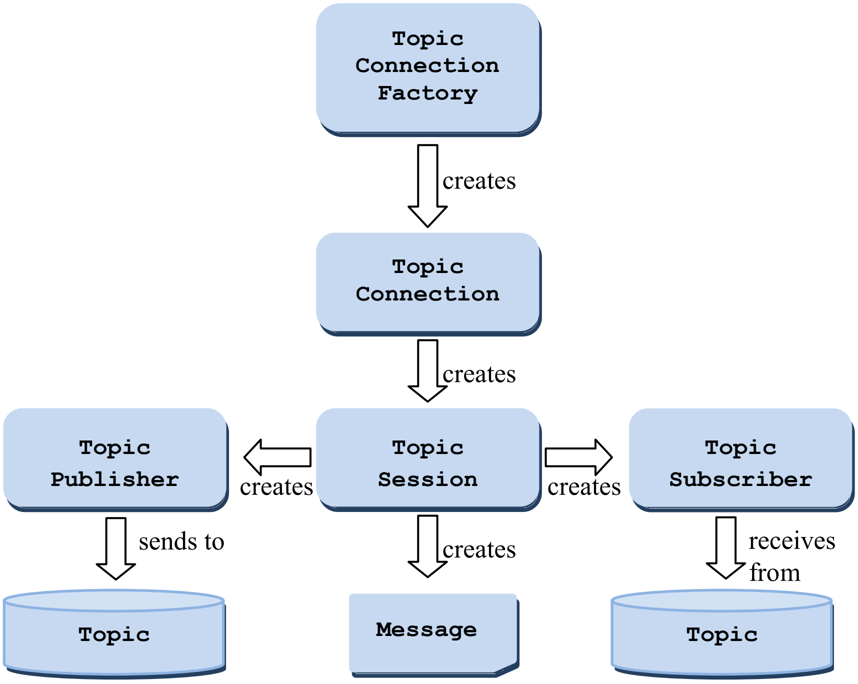


The main interfaces provided by the domain-specific API for **pub/sub** messaging are as follows:

- `TopicConnectionFactory` - an administered object used by a client to create a `TopicConnection`.
- `TopicConnection` - an active connection to a JMS provider
- `TopicSession` - a single-threaded context for sending and receiving messages
- `TopicPublisher` - an object created by a `TopicSession` that is used for sending messages to a topic

- `TopicSubscriber` - an object created by a `TopicSession` that is used for receiving messages sent to a topic

Figure 2-5 Overview of legacy pub/sub-specific API



2.10. *Relationship between interfaces*

The following table summarises the different interfaces used by the four APIs and how they correspond to one another:

Table 2-1 Relationship between interfaces used by each API

Classic API	**Simplified API**	**Domain-specific API for point-to-point messaging**	**Domain-specific API for pub/sub messaging**
`Connection Factory`	`Connection Factory`	`QueueConnection Factory`	`TopicConnection Factory`
`Connection`	`JMSContext`	`QueueConnection`	`TopicConnection`
`Session`		`QueueSession`	`TopicSession`
`MessageProducer`	`JMSProducer`	`QueueSender`	`QueueReceiver`

2.11. *Terminology for sending and receiving messages*

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.

2.12. 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.

2.12.1. Developing a JMS client

A typical JMS client using the classic API 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` object with message delivery inhibited
- Use the `Connection` to create one or more JMS `Session` objects
- Use a Session and the Destinations to create the `MessageProducer` and `MessageConsumer` objects needed
- Tell the `Connection` to start delivery of messages

In contrast, a typical JMS client using the simplified API does the following:

- Use JNDI to find a `ConnectionFactory` object
- Use JNDI to find one or more `Destination` objects
- Use the `ConnectionFactory` to create a `JMSContext` object
- Use the `JMSContext` to create the `JMSProducer` and `JMSConsumer` objects needed.
- Delivery of message is started automatically

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

2.13. 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.

2.14. 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 remaining objects are designed to be accessed by one logical thread of control at a time.

Table 2-2 Objects used in the classic API, showing which support concurrent use

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

Table 2-3 Objects used in the simplified API, showing which support concurrent use

JMS Object	Supports Concurrent Use
Destination	YES
ConnectionFactory	YES
JMSContext	NO
JMSProducer	NO
JMSConsumer	NO

Table 2-4 Objects used in the domain-specific API for point-to-point messaging, showing which support concurrent use

JMS Object	Supports Concurrent Use
Destination	YES
QueueConnectionFactory	YES
QueueConnection	YES
QueueSession	NO
QueueSender	NO
QueueReceiver	NO

Table 2-5 Objects used in the domain-specific API for pub/sub messaging, showing which support concurrent use

JMS Object	Supports Concurrent Use
Destination	YES
TopicConnectionFactory	YES
TopicConnection	YES
TopicSession	NO
TopicPublisher	NO
TopicSubscriber	NO

JMS defines some specific rules that restrict the concurrent use of sessions. These apply to the `Session` object in the classic API and to the QueueSession and TopicSession objects in the domain-specific APIs. They also apply to the `JMSContext object` in the simplified API since it encompasses a session. Since these rules 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. In the classic API and the domain-specific APIs this means using multiple session objects. In the simplified API this means using multiple `JMSContext` objects.

2.15. 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.

2.16. 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.

The legacy domain-specific APIs define request/reply helper classes (classes written using JMS) for both the point-to-point and pub/sub domains that implement a basic form of request/reply. See sections 4.1.7 "QueueRequestor" and 4.2.10 "TopicRequestor". 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 7.7 "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. Both `MessageProducer` and `JMSProducer` provide a method `setDisableMessageID` which allows the application to provide a hint to disable message ID. When an application 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 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. Both `MessageProducer` and `JMSProducer` provide a method `setDisableMessageTimestamp` which allows the application to provide a hint to disable timestamps. When an application 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.

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

3.4.6. JMSReplyTo

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

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

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

3.4.7. JMSRedelivered

If a client receives a message with the `JMSRedelivered` indicator set, it is likely, but not guaranteed, that this message was delivered 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 6.2.10"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 `JMSType` 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 ensures 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, the JMS provider calculates its expiration time by adding the time-to-live value specified on the send method to the time the message was sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed). It is represented as a `long` value which is defined as the difference, measured in milliseconds, between the expiration time and midnight, January 1, 1970 UTC.

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, the message's `JMSExpiration` header field is set to zero to indicate that the message does not expire.

When an undelivered message's expiration time is reached, 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

The following table lists the message header fields supported by JMS and whether they are set by the JMS provider or by the client application.

Table 3-1 Message header field values

Header Fields	**Set By**	**Setter method**
`JMSDestination`	JMS provider send method	`setJMSDestination` (not for client use)
`JMSDeliveryMode`	JMS provider send method	`setJMSDeliveryMode`(not for client use)
`JMSExpiration`	JMS provider send method	`setJMSExpiration` (not for client use)
`JMSDeliveryTime`	JMS provider send method	`setJMSDeliveryTime` (not for client use)
`JMSPriority`	JMS provider send method	setJMSPriority (not for client use)
`JMSMessageID`	JMS provider send method	setJMSMessageID (not for client use)
`JMSTimestamp`	JMS provider send method	setJMSTimestamp (not for client use)
`JMSCorrelationID`	Client application	`setJMSCorrelationID`, `setJMSCorrelationIDAsBytes`
`JMSReplyTo`	Client application	`setJMSReplyTo`
`JMSType`	Client application	`setJMSType`
`JMSRedelivered`	JMS provider prior to delivery	`setJMSRedelivered` (not for client use)

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

Message header fields that are defined as being set by the "JMS provider on send" will be available on the sending client as well as on the receiving client. If a message is sent synchronously (see section 7.2 "Synchronous send") then these message header fields may be accessed on the sending client when the send method returns. If a message is sent asynchronously (see section 7.3 "Asynchronous send") then these message header fields may be accessed on the sending client only after the completion listener has been invoked. The JMS provider sets these header fields using the appropriate setter methods. These setter methods are public to allow a JMS provider to set these fields when handling a message whose implementation is not its own. Client applications should not use these setter methods. Any values set by calling these methods prior to sending a message will be ignored and overwritten.

A client application may specify the delivery mode, priority, time to live and delivery delay of a message using appropriate methods on the `MessageProducer` or `JMSProducer` object, but not by methods on the `Message` object itself.

Message header fields that are defined as being set by the "JMS provider prior to delivery" will be set by the JMS provider on the message delivered to the receiving client.

3.4.12. Overriding message header fields

JMS permits an administrator to configure JMS to override the client specified values for delivery mode, priority, time to live and delivery delay. If this is done, the `JMSDeliveryMode`, `JMSPriority`, `JMSExpiration` and `JMSDeliveryTime` 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, the JMS provider calculates its delivery time by adding the delivery delay value specified on the send method to the time the message was sent (for transacted sends, this is the time the client sends the message, not the time the transaction is committed). It is represented as a `long` value which is defined as the difference, measured in milliseconds, between the delivery time time and midnight, January 1, 1970 UTC.

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 "Message selection"), to have a JMS provider select messages on its behalf using application-specific criteria.

3.5.1. Property names

Property names must obey the rules for a message selector identifier. See Section 3.8 "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 be 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

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
`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.

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.

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 6.2.10 "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 4.1.2 "Queue semantics" and 4.2.2 "Topic semantics" 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.

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

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".
  - 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` millisecond value. When a date or time literal is included in a message selector, it should be an integer literal for a millisecond value. The standard way to produce millisecond 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 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 `String` objects 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 `BytesMessage` 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. Messaging domains

JMS supports two styles of messaging:

- point-to-point (PTP) messaging using *queues*
- publish-and-subscribe (pub/sub)messaging using *topics*

4.1. JMS point-to-point model

4.1.1. 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.

4.1.2. Queue semantics

When point-to-point messaging is being used, an application sends messages to a *queue*.

An application may consume messages from the queue by creating a consumer (a `MessageConsumer`, `JMSConsumer` or `QueueReceiver` object) on that queue. A consumer may be used to consume messages either synchronously or asynchronously.

A queue may have more than one consumer. Each message in the queue is delivered to only one consumer.

A consumer may be configured to use a message selector. In this case only messages whose properties match the message selector will be delivered to the consumer. Messages which are not selected remain on the queue or are delivered to another consumer.

The order in which an individual consumer receives messages is described in section 6.2.9 “Message order” below.

By definition, if a consumer uses a message selector, or there are other consumers on the same queue, then a consumer may not receive all the messages on the queue. However those messages that are delivered to the consumer will be delivered in the order defined in section 6.2.9.

Apart from the requirements of any message selectors, JMS does not define how messages are distributed between multiple consumers on the same queue.

4.1.3. Queue management

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

4.1.4. Queue

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

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

See Section 5 “Administered objects” for more information about JMS `Destination` objects.

4.1.5. TemporaryQueue

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

See Section 6.2.2 "Creating temporary destinations" for more information.

4.1.6. QueueBrowser

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

A message must not be returned by a `QueueBrowser` before its delivery time has been reached.

4.1.7. QueueRequestor

The legacy domain-specific API for point-to-point messaging provides a `QueueRequestor` helper class to simplify making service requests.

The `QueueRequestor` constructor is given a non-transacted `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 very basic request/reply abstraction which assumes the session is non-transacted with a delivery mode of either `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`. It is expected that most applications will create less basic implementations.

There is no equivalent to this class for the classic or simplified APIs. Applications using these APIs are expected to create their own implementations.

4.1.8. 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.

4.2. JMS publish/subscribe model

4.2.1. 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 pub/sub messaging, the term *publish* is sometimes used to refer to the act of sending messages to a topic instead of the more generic terms *send* or *produce.*

The term *subscribe* is used to refer to the act of registering an interest in a topic. This creates a *subscription* from which a client consumes or receives messages.

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.

4.2.2. Topic semantics

When pub/sub messaging is being used, an application sends messages to a *topic*.

An application consumes messages from a topic by creating a *subscription* on that topic, and creating a consumer (a `MessageConsumer`, `JMSConsumer` or `TopicSubscriber` object) on that subscription.

A subscription may be thought of as an entity within the JMS provider itself whereas a consumer is a JMS object within the application.

A subscription will receive a copy of every message that is sent to the topic after the subscription is created, except if a message selector is specified. If a message selector is specified then only those messages whose properties match the message selector will be added to the subscription.

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.

Some subscriptions are restricted to a single consumer. In this case all the messages in the subscription are delivered to that consumer. Some subscriptions allow multiple consumers. In this case each message in the subscription is delivered to only one consumer. JMS does not define how messages are distributed between multiple consumers on the same subscription.

The order in which messages are delivered to a consumer is described in section 6.2.10 “Message order” below. By definition, if a subscription uses a message selector, or there are other consumers on the same subscription, then a consumer may not receive all the messages sent to the topic. However those messages that are delivered to the consumer will be delivered in the order defined in section 6.2.10.

Subscriptions may be *durable* or *non-durable*.

A *non-durable subscription* only exists for as long as there is an active consumer on the subscription. This means that any messages sent to the topic will only be added to the subscription whilst a consumer exists and is not closed.

A non-durable subscription may be either *unshared* or *shared*.

- An *unshared non-durable subscription* does not have a name and may have only a single consumer object associated with it. It is created automatically when the consumer object is created. It is not persisted and is deleted automatically when the consumer object is closed. See section 8.3.1 "Unshared non-durable subscriptions" below.

- A *shared non-durable subscription* is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it. It is created automatically when the first consumer object is created. It is not persisted and is deleted automatically when the last consumer object is closed. See section 8.3.2 “Shared non-durable subscriptions” below.

At the cost of higher overhead, a subscription may be *durable*. A *durable subscription* is persisted and continues to accumulate messages until explicitly deleted, even if there are no consumer objects consuming messages from it.

A durable subscription has a unique identity that is retained by JMS. Subsequent consumer objects can resume the subscription in the state it was left by the prior consumer. If there are no active consumers on a durable subscription, JMS retains the subscription’s messages until they are consumed or until they expire.

A durable subscription may also be either *unshared* or *shared*.

- An *unshared durable subscription* is identified by name and client identifier (which must be set) and may have only a single consumer object associated with it. See section 8.3.3 “Unshared durable subscriptions” below

- A *shared durable subscription* is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it. See section 8.3.4 “Shared durable subscriptions” below.

A durable subscription which exists but which does not currently have a non-closed consumer object associated with it is described as being *inactive*.

When an unshared non-durable or durable subscription is created, the `noLocal` parameter may be specified. The effect of setting this parameter is defined in sections 8.3.1 "Unshared non-durable subscriptions" and 8.3.3 “Unshared durable subscriptions” below.

4.2.3. 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.

4.2.4. Subscription name characters and length

The JMS provider must allow a durable or non-durable subscription name to contain the following characters:

- Java letters
- Java digits
- Underscore (_)
- Dot (.)
- Minus (-)

JMS providers may support additional characters to these, but applications which use them may not be portable.

The JMS provider must allow a durable or non-durable subscription name to have up to 128 characters.

JMS providers may support names longer than this, but applications which use longer names may not be portable.

4.2.5. 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 *temporary topic* is provided for creating a topic that is unique to a particular connection. See Section 4.2.7 "Temporary topics" for more details.

4.2.6. 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 JMS 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.

4.2.7. Temporary topics

A `TemporaryTopic` is a unique `Topic` object created for the duration of a `JMSContext`, `Connection` or `TopicConnection`. It is a system defined `Topic` whose messages may be consumed only by the connection 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 6.2.2 "Creating temporary destinations" for more information.

4.2.8. 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 durable subscription should be used.

4.2.9. 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.

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.

4.2.10. TopicRequestor

The legacy domain-specific API for pub/sub messaging provides a `TopicRequestor` helper class to simplify making service requests.

The `TopicRequestor` constructor is given a non-transacted `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 very basic request/reply abstraction which assumes the session is non-transacted with a delivery mode of either `AUTO_ACKNOWLEDGE` or

DUPS_OK_ACKNOWLEDGE. It is expected that most applications will create less basic implementations.

There is no equivalent to this class for the classic or simplified APIs. Applications using these APIs are expected to create their own implementations.

4.2.11. Reliability

When all messages for a topic must be received, a durable subscriber should be used. JMS ensures 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 4-1 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

5. Administered objects

5.1. Overview

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.

5.2. 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 6.2.3 "Creating Destination objects" for more information.

`Destination` objects support concurrent use.

5.3. Connection factories

A connection factory object 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.

- The classic API a connection uses connection factories of type `ConnectionFactory`.
- The simplified API uses connection factories of type `ConnectionFactory`.
- The domain-specific API for point-to-point messaging uses connection factories of type `QueueConnectionFactory`.
- The domain-specified API for pub-sub messaging uses connection factories of type `TopicConnectionFactory`.

Connection factory objects support concurrent use.

For information on how to use a connection factory to create a connection, see section 6.1 "Connections".

6. Connecting to a JMS provider

6.1. Connections

JMS uses the term *connection* to refer to a client's active connection to its JMS provider. It will typically allocate provider resources outside the Java virtual machine.

A connection is created using a connection factory. For more information about connection factories see section 5.3 "Connection factories".

A connection may be used to create one or more sessions. Sessions are used to send and consume messages and are described in section 6.2 "Sessions".

- In the classic API a connection is represented by a `Connection` object and is created using one of the following methods on `ConnectionFactory`:

  `createConnection()`

  `createConnection(String userName, String password)`

  A `Connection` object may be used to create separate `Session` objects. `Connection` objects support concurrent use.

- In the simplified API a connection is represented by a `JMSContext` object and is created using one of the following methods on `ConnectionFactory`.

  `createContext()`

  `createContext(int sessionMode)`

  `createContext(String userName, String password)`

  `createContext(String userName, String password, int sessionMode)`

  A `JMSContext` represents both a connection and a session. Although a connection supports concurrent use, a session does not. `JMSContext` objects therefore do not support concurrent use

- In the domain-specific API for point-to-point messaging a connection is represented by a `QueueConnection` object and is created using one of the following methods on `QueueConnectionFactory`:

  `createQueueConnection()`

  `createQueueConnection(String userName, String password)`

  A `QueueConnection` object may be used to create separate `QueueSession` objects. `QueueConnection` objects support concurrent use.

- In the domain-specified API for pub-sub messaging a connection is represented by a `TopicConnection` object and is created using one of the following methods on `TopicConnectionFactory`:

`createTopicConnection()`

`createTopicConnection(String userName, String password)`

A `TopicConnection` object may be used to create separate `TopicSession` objects. `TopicConnection` objects 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 provides `ConnectionMetaData`.
- It supports an optional `ExceptionListener`.

Due to the authentication and communication setup done when a connection is created, the objects that represent a connection are relatively heavyweight JMS objects. 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.

6.1.1. 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.

6.1.2. 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 connection'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 an `IllegalStateException`.

An application may explicitly set a connection's client identifier by calling the `setClientID` method on the `Connection`, `JMSContext`, `QueueConnection` or `TopicConnection` object.

If a client explicitly sets a connection's client identifier it must do so immediately after creating the `Connection`, `JMSContext`, `QueueConnection` or `TopicConnection` and before any other action on this object 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 a `JMSException` being thrown when such use is attempted; it may result in the offending client being blocked; or some other solution. A JMS provider must ensure 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 mandatory use in identifying an unshared durable subscription or its optional use in identifying a shared durable or non-durable subscription.

6.1.3. Connection setup

- In the classic API, a JMS client typically creates a `Connection`, one or more `Session` objects, and a number of `MessageProducer` and `MessageConsumer` objects.
- In the simplified API, a JMS client typically creates a `JMSContext` and a number of `JMSProducer` and `JMSConsumer` objects.
- In the domain-specific API for point-to-point messaging, a JMS client typically creates a `QueueConnection`, one or more `QueueSession` objects, and a number of `QueueSender` and `QueueReceiver` objects.
- In the domain-specific API for pub/sub messaging, a JMS client typically creates a TopicConnection, one or more TopicSession objects, and a number of TopicPublisher and TopicSubscriber objects.

6.1.4. Starting a connection

When a `Connection`, `JMSContext`, `QueueConnection` or `TopicConnection` is created, it is in *stopped* mode. That means that no messages are being delivered to it.

In the case of a `Connection`, `QueueConnection` or `TopicConnection` it is typical to leave the connection in stopped mode until setup is complete. At that point the `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.

These objects 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.

In the case of a `JMSContext` the connection is started automatically when the first consumer is created. Applications may disable this behaviour by calling `setAutoStart(false)` and then calling `start()` explicitly when required.

Whether a connection is started or stopped only affects the use of a connection to *receive* messages. It has no effect on the use of the connection to *send* messages. A connection may be used to send messages irrespective of whether it is started or stopped.

It is important to note that clients rely on the fact that no messages will be delivered to a consumer until its connection has been started. JMS Providers must ensure that this is the case.

6.1.5. 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 consumer objects is inhibited: synchronous receives block, and messages are not delivered to any message listeners.

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`.

6.1.6. ConnectionMetaData

All the objects that represent a connection provide 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.

6.1.7. 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 `getExceptionListener()` 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 `onException` 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 `onException` 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.

6.1.8. 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`.

Closing a `Connection`, `QueueConnection` or `TopicConnection` closes its constituent sessions, producers, consumers or queue browsers. The connection close is sufficient to signal the JMS provider that all resources for the connection should be released.

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

The `Connection`, `JMSContext`, `QueueConnection and TopicConnection` interfaces all extend the `java.lang.AutoCloseable` interface. This means that applications which create these objects in a `try-with-resources` statement do not need to call the `close` method when they are no longer needed. Instead these objects 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 ensure 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.

6.2. Sessions

In JMS a *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 lightweight JMS object.

---

[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.

[5] There are no restrictions on the number of threads that can use a session or any objects 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.

- In the classic API a session is represented by a `Session` object and is creating using one of the following methods on Connection:

  ```
  createSession()
  ```

  ```
  createSession(boolean transacted, int acknowledgeMode)
  ```

  ```
  createSession(int sessionMode)
  ```

- In the simplified API a connection and a session are represented by a single `JMSContext` object. When a `JMSContext` is created the underlying session is created automatically.

- In the domain-specific API for point-to-point messaging a session is represented by a `QueueSession` object and is created using the following method on `QueueConnection`:

  ```
  createQueueSession(boolean transacted, int acknowledgeMode)
  ```

- In the domain-specified API for pub-sub messaging a session is represented by a `TopicSession` object and is created using the following method on `TopicConnection`:

  ```
  createTopicSession(boolean transacted, int acknowledgeMode)
  ```

A session serves several purposes:

- It is a factory for producer and consumer objects. These are described in chapter 7 “Sending messages” and chapter 8 “Receiving messages”.
- It is a factory for `TemporaryTopic` and `TemporaryQueue` objects.
- 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` objects registered with it.
- It is a factory for `QueueBrowser` objects.
- It provides the `unsubscribe` method for deleting durable topic subscriptions.

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

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

6.2.1. Producer and consumer creation

A session can create and service multiple producer and consumer objects. See section 7 “Sending messages” and section 8 “Receiving messages” 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.

6.2.2. 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 consumer 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 consumer objects for them.

One typical use for a temporary destination is as the `JMSReplyTo` destination for service requests.

Each `TemporaryQueue` or `TemporaryTopic` 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.

6.2.3. Creating Destination objects

Most clients will use `Destination` objects 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 `Destination` objects by dynamically manufacturing one using a provider specific destination name. Sessions provide a JMS provider-specific method for doing this.

6.2.4. 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".

6.2.5. *Threading restrictions on 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 6.1.8 "Closing a connection" and Section 6.2.15 "Closing a " for further details.

One typical use is to have a thread call `receive()` on a consumer, which blocks until a message arrives. The thread may then use one or more of the session's producer objects.

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, 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 message producers, message consumers and message listeners are created in the right order. For instance, a bad order may cause a `MessageListener` to use a producer object that has yet to be created; or messages may arrive in the wrong order due to the order in which `MessageListener` objects 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 consumer, 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 `JMSContext` incorporates a session it is subject to the same threading restrictions as a Session. For more information, and an exception to this, see section 6.2.6 “Threading restrictions on a JMSContext”.

Additional threading restrictions apply to applications which perform an asynchronous send. See section 7.3 "Asynchronous send" and in particular section 7.3.7 "Restrictions on threading"

6.2.6. Threading restrictions on a JMSContext

Since a `JMSContext` incorporates a session it is subject to the same threading restrictions as a session. These are described in section 6.2.5 "Threading restrictions on a session" which explains how a session may only be used by one thread at a time.

The `JMSContext` method `createContext` does not use its underlying session and so is not subject to this threading restriction.

This restriction also does not apply to the `close` method on `JMSContext` or `JMSConsumer` (since closing a session or consumer from another thread is permitted).

By default, when `createConsumer` or `createDurableConsumer` is used to create a `JMSConsumer` the connection will automatically be started. This behaviour is described in section 6.1.4 “Starting a connection”. It means that if `setMessageListener` is called to configure the asynchronous delivery of messages then the `JMSContext`'s session will immediately become dedicated to the thread of control that delivers messages to the listener and the application must not subsequently call methods on the `JMSContext` from another thread of control. However this restriction does not apply to applications which call `setMessageListener` to set a second or subsequent message listener. 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.

6.2.7. 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 a 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 6.2.10 "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`.

6.2.8. *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 11 "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 11 "JMS application server facilities" for more information.

6.2.9. *Message order*

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

6.2.9.1. *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 6.2.10 "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 6.2.9.2 "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.

6.2.9.2. *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 6.2.7 "Transactions" for more information.

6.2.10. 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 message listener 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

6.2.11. 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.

6.2.12. 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.

6.2.13. 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 single-threaded ones.

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

To receive messages asynchronously, a client creates a consumer object (`MessageConsumer`, `JMSConsumer`, `QueueReceiver` or `TopicConsumer`) and uses the `setMessageListener` method to register with it an object that implements the JMS `MessageListener` interface. *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.

6.2.14. 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.

6.2.15. Closing a session

Since a provider may allocate some resources on behalf of a session outside the JVM, clients should close a session when it is not needed. Relying on garbage collection to eventually reclaim these resources may not be timely enough. The same is true for any producer and consumer objects created by a session.

The `close` methods on `Session`, `QueueSession` and `TopicSession` allow a session to be closed separately from the connection used to create it.

The `close` method on `JMSContext` closes the underlying session. If there are no other active (not closed) `JMSContext` objects using the underlying connection then it also closes the underlying connection.

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 section 6.1.8 "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 producers, 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`, `JMSContext`, `QueueSession` and `TopicSession` interfaces all extend the `java.lang.AutoCloseable` interface. This means that applications which create these objects in a `try-with-resources` statement do not need to call the `close` method when they no longer needed. Instead these objects 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 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.

7. Sending messages

7.1. Producers

A client application uses a *producer* to send messages to a `Destination`.

- In the classic API a producer is represented by a `MessageProducer` object and is created using the method `createProducer(Destination destination)` on `Session`. The `destination` parameter specifies the destination to which the producer will send messages.

  If `destination` is set to `null` then the destination must be specified on every send operation. A typical use for this style of producer is to send replies to requests using the request's `JMSReplyTo` destination.

- In the simplified API a producer is represented by a `JMSProducer` object and is created using the method `createProducer()` on `JMSContext`. The destination must be specified on every send operation.

- In the domain-specific API for point-to-point messaging a producer is represented by a `QueueSender` object and is created using the method `createSender(Queue queue)` on `QueueSession`. The `queue` parameter specifies the queue to which the producer will send messages. If `queue` is set to `null` then the queue must be specified on every send operation.

- In the domain-specified API for pub-sub messaging a producer is represented by a `TopicPublisher` object and is created using the method `createPublisher(Topic topic)` on `TopicSession`. The `topic` parameter specifies the topic to which the producer will send messages. If `topic` is set to `null` then the topic must be specified on every send operation.

A producer may be used to send a message either synchronously or asynchronously. For more details see sections 7.2 "Synchronous send" and 7.3 "Asynchronous send".

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

7.2. Synchronous send

- In the classic API the following methods on `MessageProducer` may be used to send a message synchronously:

  ```
  send(Message message)
  ```

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

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

- In the simplified API the following methods on `JMSProducer` may be used to send a message synchronously:

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

  ```
  send(Destination destination, String body)
  ```

  ```
  send(Destination destination, Map<String,Object>
  body)
  ```

  ```
  send(Destination destination, byte[] body)
  ```

  ```
  send(Destination destination, Serializable body)
  ```

  ```
  send(Destination destination, String body)
  ```

- In the domain-specific API for point-to-point messaging the following methods on QueueSender may be used to send a message synchronously:

  ```
  send(Message message)
  ```

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

  ```
  send(Queue queue, Message message)
  ```

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

  These are in addition to the methods inherited from `MessageProducer` and listed above.

- In the domain-specific API for pub/sub messaging the following methods on `TopicPublisher` may be used to send a message synchronously:

  ```
  publish(Message message)
  ```

  ```
  publish(Message message, int deliveryMode, int
  priority, long timeToLive)
  ```

  ```
  publish(Topic topic, Message message)
  ```

  ```
  publish(Topic topic, Message message, int
  deliveryMode, int priority, long timeToLive)
  ```

  These are in addition to the methods inherited from `MessageProducer` and listed above.

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.

7.3. Asynchronous send

Clients may alternatively send a message asynchronously. This permits the JMS provider to perform part of the work involved in sending the message in a separate thread.

- In the classic API the following methods on `MessageProducer` may be used to send a message asynchronously

```
send(Message message, CompletionListener
completionListener)

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

send(Destination destination, Message message,
CompletionListener completionListener)

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

- In the simplified API a `JMSProducer` may be used to send a message asynchronously by using calling the method `setAsync(CompletionListener completionListener)` on the `JMSProducer` prior to calling one of the normal `send` methods listed in section 7.2 "Synchronous send".
- In the domain-specific API for point-to-point messaging a `QueueSender` may be used to send a message synchronously any of the methods inherited from `MessageProducer` and listed above.
- In the domain-specific API for pub/sub messaging a `TopicPublisher` may be used to send a message synchronously using any of the methods inherited from `MessageProducer` and listed above.

When the message has been successfully sent the JMS provider invokes the callback method `onCompletion` on an application-specified `CompletionListener` object. Only when that callback has been invoked can the application be sure that the message has been successfully sent with the same degree of confidence as if a normal synchronous send had been performed. An application which requires this degree of confidence must therefore wait for the callback to be invoked before continuing.

The following information is intended to give an indication of how an asynchronous send would typically be implemented.

In some JMS providers, a normal synchronous send involves sending the message to a remote JMS server and then waiting for an acknowledgement to be received before returning. It is expected that such a provider would implement an asynchronous send by sending the message to the remote JMS server and then returning without waiting for an acknowledgement. When the acknowledgement is received, the JMS provider would notify the application by invoking the `onCompletion` method on the application-specified `CompletionListener` object. If for some reason the acknowledgement is not received the JMS provider would notify the application by invoking the `CompletionListener`'s `onException` method.

In those cases where the JMS specification permits a lower level of reliability, a normal synchronous send might not wait for an acknowledgement. In that case it is expected that an asynchronous send would be similar to a synchronous send: the JMS provider would send the message to the remote JMS server and then return without waiting for an acknowledgement. However the JMS provider would still notify the application that the send had completed by invoking the `onCompletion` method on the application-specified `CompletionListener` object.

It is up to the JMS provider to decide exactly what is performed in the calling thread and what, if anything, is performed asynchronously, so long as it satisfies the requirements given in the following sections:

7.3.1. *Quality of service*

After the send operation has completed successfully, which means that the message has been successfully sent with the same degree of confidence as if a normal synchronous send had been performed, the JMS provider must invoke the `CompletionListener`'s `onCompletion` method.The `CompletionListener` must not be invoked earlier than this.

7.3.2. *Exceptions*

If an exception is encountered during the call to the `send` method then an appropriate exception should be thrown in the thread that is calling the `send` method. In this case the JMS provider must not invoke the `CompletionListener`'s `onCompletion` or `onException` method.

If an exception is encountered which cannot be thrown in the thread that is calling the `send` method then the JMS provider must call the `CompletionListener`'s `onException` method.

In both cases if an exception occurs it is undefined whether or not the message was successfully sent.

7.3.3. *Message order*

If the same producer is used to send multiple messages then JMS message ordering requirements (see section 6.2.9 "Message order") must be satisfied. This applies even if a combination of synchronous and asynchronous sends has been performed. The application is not required to wait for an asynchronous send to complete before sending the next message.

7.3.4. *Close, commit or rollback*

If the application calls `close` to close the producer, session or connection then the JMS provider must block until any incomplete send operations have been completed and all `CompletionListener` callbacks have returned before closing the object and returning.

If the session is transacted (uses a local transaction) then when the `commit` or `rollback` method is called the JMS provider must block until any incomplete send operations have been completed and all `CompletionListener` callbacks have returned before performing the commit or rollback.

Incomplete sends should be allowed to complete normally unless an error occurs.

A `CompletionListener` callback method must not call `close` on its own producer, session (including `JMSContext`) or connection or call `commit` or `rollback` on its own session. Doing so will cause the `close`, `commit` or `rollback` to throw an `IllegalStateException` or `IllegalStateRuntimeException` (depending on the method signature).

7.3.5. *Restrictions on usage in Java EE*

An asynchronous send is not permitted in a Java EE EJB or web container. If the application component violates this restriction the send method may throw a `JMSException` or `JMSRuntimeException` (depending on the method signature).

7.3.6. *Message headers*

JMS defines a number of message header fields and message properties which must be set by the "JMS provider on send". See section 3.4.11 "How message header values are set" and section 3.5.9 "JMS defined properties". If the send is asynchronous these fields and properties may be accessed on the sending client only after the `CompletionListener` has been invoked. If the `CompletionListener`'s `onException` method is called then the state of these message header fields and properties is undefined. See also section 7.3.9 "Restrictions on the use of the Message object" below.

7.3.7. *Restrictions on threading*

Applications that perform an asynchronous send must confirm to the threading restrictions defined in section 6.2.5 "Threading restrictions on a session". This means that the session may be used by only one thread at a time.

Setting a `CompletionListener` does not cause the session to be dedicated to the thread of control which calls the `CompletionListener`. The application thread may therefore continue to use the session after performing an asynchronous send. However the `CompletionListener`'s callback methods must not use the session if an application thread might be using the session at the same time.

7.3.8. *Use of the CompletionListener by the JMS provider*

A session will only invoke one `CompletionListener` callback method at a time. For a given `MessageProducer` or `JMSContext`, callbacks (both `onCompletion` and `onException`) will be performed in the same order as the corresponding calls to the asynchronous send method.

A JMS provider must not invoke the `CompletionListener` from the thread that is calling the asynchronous send method.

7.3.9. *Restrictions on the use of the Message object*

Applications which perform an asynchronous send must take account of the restriction that a `Message` object is designed to be accessed by one logical thread of control at a time and does not support concurrent use. See section 2.14 "Multi-threading".

After the `send` method has returned, the application must not attempt to read the headers, properties or body of the `Message` object until the `CompletionListener's` `onCompletion` or `onException` method has been called. This is because the JMS provider may be modifying the `Message` object in another thread during this time.

A JMS provider may throw a `JMSException` if the application attempts to access or modify the `Message` object after the `send` method has returned and before the `CompletionListener` has been invoked. If the JMS provider does not throw an exception then the behaviour is undefined.

7.4. Setting message delivery options

A client can specify a producer's delivery mode, priority, time-to-live and delivery delay. This sets these values for all messages sent by a producer,

An application that uses the classic or domain-specific APIs may also specify the delivery mode, priority, and time-to-live as parameters to the `send` method used to send the message. This overrides any values set on the producer itself.

For more information on these various options see sections 7.7 "Message delivery mode", Section 3.4.10 "JMSPriority", 7.8 "Message time-to-live" and 7.9 "Message delivery delay".

7.5. Setting message properties

Prior to sending a message, the client application may use methods on the `Message` object to set message properties.

Applications using the simplified API may also set message properties on the `JMSProducer.` There are nine methods on `JMSProducer`, all called `setProperty`. Any message properties set using these methods will override any values that have been set directly on the message. For applications which use the methods which send the message body directly, and which therefore do not use a `Message` object, these methods offer the only way to set message properties.

7.6. Setting message headers

Prior to sending a message, the application may use methods on the `Message` object to set the `JMSCorrelationID`, `JMSReplyTo` and `JMSType` message headers.

For more information see sections 3.4.5 "JMSCorrelationID" 3.4.6 "JMSReplyTo" and 3.4.8 "JMSType" above.

Applications using the simplified API may also set these message headers on the `JMSProducer.` Any message headers set using these methods will override any values that have been set directly on the message. For applications which use the methods which send the message body directly, and which therefore do not use a `Message` object, these methods offer the only way to set these message headers.

7.7. 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 9.1 "Reliability" for further discussion on this topic.

An application may specify the required delivery mode using the method `setDeliveryMode` on the producer object. This sets the delivery mode of all messages sent using that producer. An application that uses the classic or domain-specific APIs may also specify the delivery mode as a parameter to the `send` method used to send the message. Note however that the `setDeliveryMode` method on `Message` cannot be used to set the delivery mode of a message.

See also section 3.4.2 “JMSDeliveryMode”.

7.8. Message time-to-live

A client can specify a time-to-live value in milliseconds for each message it sends. This is used to determine the message's expiration time which is calculated by adding the time-to-live value specified on the send method to the time the message was 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.

An application may specify the required time-to-live using the method `setTimeToLive` on the producer object. This sets the time-to-live of all messages sent using that producer. An application that uses the classic or domain-specific APIs may also specify the time-to-live as a parameter to the `send` method used to send the message. Note however that the `setTimeToLive` method on `Message` cannot be used to set the time-to-live of a message.

See also section 3.4.9 "JMSExpiration".

7.9. Message delivery delay

A client can specify a delivery delay value in milliseconds for each message it sends. This is used to determine the message's delivery time which is calculated by adding the delivery delay value specified on the send method to the time the message was 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 deliver the message to a consumer. The provider must not deliver messages before the delivery time has been reached.

If a message is published to a topic, it will only be added to a durable or non-durable subscription on that topic if the subscription exists at the time the message is sent.

An application may specify the required delivery delay using the method `setDeliveryDelay` on the producer object. This sets the delivery delay of all messages sent using that producer. Note however that the `setDeliveryDelay` method on `Message` cannot be used to set the delivery delay of a message.

See also section 3.4.13 "JMSDeliveryTime".

7.10. JMSProducer method chaining

In the simplified API, the various setter methods on `JMSProducer` all return the `JMSProducer` object. This allows method calls to be chained together, allowing a fluid programming style. For example:

```
context.createProducer().
   setProperty("foo", "bar").
   setTimeToLive(10000).
   setDeliveryMode(NON_PERSISTENT).
   setDisableMessageTimestamp(true).
   send(inboundQueue, body);
```

Instances of `JMSProducer` are intended to be lightweight objects which can be created freely and which do not consume significant resources. `JMSProducer` therefore does not provide a close method.

8. Receiving messages

8.1. Consumers

A client uses a *consumer* to receive messages from a destination.

- In the classic API a consumer is represented by a `MessageConsumer` object and is created using one of several methods on `Session`.
- In the simplified API a consumer is represented by a `JMSConsumer` object and is created one of several methods on `JMSContext`.
- In the domain-specific API for point-to-point messaging a consumer is represented by a `QueueReceiver` object and is created using one of several methods on `QueueSession`.
- In the domain-specified API for pub-sub messaging a consumer is represented by a `TopicSubscriber` object and is created using one of several methods on `TopicSession`.

In all cases the destination from which the consumer will receive messages must be specified.

The methods used to create a consumer are described in sections 8.2 “Creating a consumer on a queue” and 8.3 “Creating a consumer on a topic” below.

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. See sections 8.5 “Receiving messages synchronously”, 8.6 “Receiving message bodies synchronously” and 8.7 “Receiving messages asynchronously” below.

8.2. Creating a consumer on a queue

The methods used to create a consumer on a queue vary depending on which API is being used. The basic semantics of queues were introduced in section 4.1.2 “Queue semantics”.

- In the classic API a consumer on a queue is created using one of several `createConsumer` methods on `Session`, all of which return a `MessageConsumer`.
- In the simplified API a consumer on a queue is created using one of several `createConsumer` methods on `JMSContext`, all of which return a `JMSConsumer`.
- In the domain-specific API for point-to-point messaging a consumer on a queue is created using one of several `createReceiver` methods on `QueueSession`, all of which return a `QueueReceiver`:

8.3. Creating a consumer on a topic

The methods used to create a consumer on a topic vary depending on what kind of topic subscription is required, and which API is being used. The basic concepts of topics were introduced in section 4.2.2 "Topic semantics" and are explained in more detail below.

8.3.1. Unshared non-durable subscriptions

An unshared non-durable subscription is the simplest way to consume messages from a topic.

An unshared non-durable subscription is created, and a consumer object created on that subscription, using one of the following methods:

- In the classic API, one of several `createConsumer` methods on `Session`. These return a `MessageConsumer` object.
- In the simplified API, one of several `createConsumer` methods on `JMSContext`. These return a `JMSConsumer` object.
- In the legacy domain-specific API for pub/sub, using one of several `createSubscriber` methods on `TopicSession`. These return a `TopicSubscriber` object.
- In the legacy domain-specific API for pub/sub, using one of several `createConsumer` methods on `TopicSession`. As these methods are inherited from `Session` they return a `MessageConsumer` object.

An unshared non-durable subscription does not have a name. Each call to `createConsumer` or `createSubscriber` creates a new subscription.

An unshared non-durable subscription only exists for as long as the consumer remains active. This means that any messages sent to the topic will only be added to the subscription for as long as the consumer object exists and is not closed. The subscription is not persisted and will be deleted (together with any undelivered messages associated with it) when the consumer is closed.

If a message selector is specified then only messages with properties matching the message selector expression will be added to the subscription.

The `noLocal` parameter may be used to specify that messages published to the topic by its own connection must not be added to the subscription.

Each unshared non-durable subscription has a single consumer. If the application needs to create multiple consumers on the same subscription then a shared non-durable subscription should be used instead. See section 8.3.2 "Shared non-durable subscriptions".

If the application needs to be able to receive messages that were sent to the topic even when there was no active consumer on it then a durable subscription should be used instead. See section 8.3.3 "Unshared durable subscriptions".

8.3.2. Shared non-durable subscriptions

A non-durable shared subscription is used by a client that needs to be able to share the work of receiving messages from a non-durable topic subscription amongst multiple consumers. A non-durable shared subscription may therefore have more than one consumer. Each message from the subscription will be delivered to only one of the consumers on that subscription.

A shared non-durable subscription is created, and a consumer created on that subscription, using one of the following methods:

- In the classic API, one of several `createSharedConsumer` methods on `Session`. These return a `MessageConsumer` object.
- In the simplified API, one of several `createSharedConsumer` methods on `JMSContext`. These return a `JMSConsumer` object.
- In the legacy domain-specific API for pub/sub, using one of several `createSharedConsumer` methods on `TopicSession`. As these methods are inherited from `Session` they return a `MessageConsumer` object.

The same methods may be used to create a consumer on an existing shared non-durable subscription.

A shared non-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 shared non-durable subscription was first created then a client which subsequently wishes to create a consumer on that shared non-durable subscription must use the same client identifier.

A shared non-durable subscription only exists for as long as there is an active consumer on the subscription. This means that any messages sent to the topic will only be added to the subscription whilst a consumer object exists and is not closed. The subscription is not persisted and will be deleted (together with any undelivered messages associated with it) when the last consumer on the subscription is closed.

If there is an active (i.e. not closed) consumer on the shared non-durable subscription, and an attempt is made to create an additional consumer, specifying the same name and client identifier (if set) but a different topic or message selector, then a `JMSException` or `JMSRuntimeException` (depending on the method signature) will be thrown.

If a message selector is specified then only messages with properties matching the message selector expression will be added to the subscription.

There is no restriction to prevent a shared non-durable subscription and a durable subscription having the same name. Such subscriptions would be completely separate.

See also section 6.1.2 "Client identifier".

8.3.3. Unshared durable subscriptions

A durable subscription is used by an application that needs to receive all the messages published on a topic, including the ones published when there is no consumer 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 consumer on the durable subscription or until they have expired.

An *unshared* durable subscription may have only one active (i.e. not closed) consumer at the same time.

An unshared durable subscription is created, and a consumer created on that subscription, using one of the following methods:

- In the classic API, one of several `createDurableConsumer` methods on `Session`. These return a MessageConsumer object.
- In the simplified API, one of several `createDurableConsumer` methods on JMSContext. These return a JMSConsumer object.
- In the legacy domain-specific API for pub/sub, one of several `createDurableSubscriber` methods on `Session and TopicSession`. These return a `TopicSubscriber` object.

The same methods may be used to create a consumer on an existing unshared durable subscription.

An unshared durable subscription is identified by a name specified by the client and by the client identifier, which must be set. A client which subsequently wishes to create a consumer on that unshared durable subscription must use the same client identifier.

An unshared durable subscription is persisted and will continue to accumulate messages until it is deleted using the `unsubscribe` method on the `Session, JMSContext` or `TopicSession.` It is erroneous for a client to delete a durable subscription while it has an active consumer or while a message received from it is part of a current transaction or has not been acknowledged in the session.

If there is an active (i.e. not closed) consumer on the unshared durable subscription, and an attempt is made to create an additional consumer, specifying the same name and client identifier, then a `JMSException` or `JMSRuntimeException` (depending on the method signature) will be thrown.

If there is no active (i.e. not closed) consumer on the unshared durable subscription, and an attempt is made to create a new consumer on that unshared durable subscription, specifying the same name and client identifier but a different topic, message selector or `noLocal` value, then this is equivalent to unsubscribing (deleting) the old one and creating a new one.

A shared durable subscription and an unshared durable subscription may not have the same name and client identifier. If the application calls one of the `createDurableConsumer` or `createDurableSubscriber` methods, and a shared durable subscription already exists with the same name and client identifier, then a `JMSException` or `JMSRuntimeException` (depending on the method signature) will be thrown.

If a message selector is specified then only messages with properties matching the message selector expression will be added to the subscription.

The `noLocal` parameter may be used to specify that messages published to the topic by the `Session`, `JMSContext` or `TopicSession`'s own connection, or any other connection with the same client identifier, will not be added to the durable subscription.

There is no restriction to prevent a durable subscription and a shared non-durable subscription having the same name. Such subscriptions would be completely separate.

See also section 6.1.2 "Client identifier".

8.3.4. Shared durable subscriptions

A durable subscription is used by an application that needs to receive all the messages published on a topic, including the ones published when there is no consumer 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 consumer on the durable subscription or until they have expired.

A *shared* non-durable subscription is used by a client that needs to be able to share the work of receiving messages from a durable subscription amongst multiple consumers. A shared durable subscription may therefore have more than one consumer. Each message from the subscription will be delivered to only one of the consumers on that subscription.

A shared durable subscription is created, and a consumer created on that subscription, using one of the following methods:

- In the classic API, one of several `createSharedDurableConsumer` methods on `Session`. These return a `MessageConsumer` object.
- In the simplified API, one of several `createSharedDurableConsumer` methods on `JMSContext`. These return a `JMSConsumer` object.
- In the legacy domain-specific API for pub/sub, using one of several `createSharedDurableConsumer` methods on `JMSContext` return a `JMSConsumer`.

The same methods may be used to create a consumer on an existing shared durable subscription.

A shared 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 shared durable subscription was first created then a client which subsequently wishes to create a consumer on that shared durable subscription must use the same client identifier.

A durable subscription is persisted and will continue to accumulate messages until it is deleted using the `unsubscribe` method on the `Session, TopicSession` or `JMSContext`. It is erroneous for a client to delete a durable subscription while it has an active consumer or while a message received from it is part of a current transaction or has not been acknowledged in the session.

If there are no active (i.e. not closed) consumers on the shared durable subscription, and an attempt is made to create a new consumer, specifying the same name and client identifier (if set) but a different topic or message selector, then this is equivalent to unsubscribing (deleting) the old one and creating a new one.

If there is an active (i.e. not closed) consumer on the shared durable subscription, and an attempt is made to create an additional consumer, specifying the same name and client identifier (if set) but a different topic or message selector, then a `JMSException` or `JMSRuntimeException` (depending on the method signature) will be thrown.

A shared durable subscription and an unshared durable subscription may not have the same name and client identifier. If the application calls one of the `createSharedDurableConsumer` methods, and an unshared durable subscription already exists with the same name and client identifier, then a `JMSException` or `JMSRuntimeException` is thrown.

If a message selector is specified then only messages with properties matching the message selector expression will be added to the subscription.

There is no restriction to prevent a durable subscription and a shared non-durable subscription having the same name. Such subscriptions would be completely separate.

See also section 6.1.2 "Client identifier".

8.4. Starting 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 6.1.3 "Connection setup".

The simplified API provides corresponding `start` and `stop` methods on `JMSContext`. The `start` method will be called automatically when `createConsumer` or `createDurableConsumer` are called on the `JMSContext` 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 `JMSContext` prior to calling `createConsumer` or `createDurableConsumer`.

8.5. Receiving messages synchronously

A client application can request the next message from a consumer by callinge `receive`, `receive(long timeout)` or `receiveNoWait` methods. These methods return a `Message` object.

Table 8-1 `MessageConsumer,JMSConsumer, QueueReceiver and TopicSubscriber` methods to receive a message synchronously

`Message receive ();`	Returns the next message produced for this `JMSConsumer`
`Message receive ( long timeout);`	Returns the next message produced for this `JMSConsumer` that arrives within the specified timeout period
`Message receiveNoWait();`	Returns the next message produced for this `JMSConsumer` if one is immediately available

8.6. Receiving message bodies synchronously

A client application using the simplified API can use the following methods on `JMSConsumer` to receive a message body directly.

Table 8-2 `JMSConsumer` methods to receive a message body synchronously

`<T> T receiveBody( Class<T> c);`	Receives the next message produced for this `JMSConsumer` and returns its body as an object of the specified type
`<T> T receiveBody( Class<T> c, long timeout);`	Receives the next message produced for this `JMSConsumer` that arrives within the specified timeout period, and returns its body as an object of the specified type
`<T> T receiveBodyNoWait( Class<T> c);`	Receives the next message produced for this `JMSConsumer` if one is immediately available and returns its body as an object of the specified type

These methods may be used to receive any type of message except for `StreamMessage` and `Message`, so long as the message has a body which is capable of being assigned to the specified type. This means that the specified class or interface must either be the same as, or a superclass or superinterface of, the class of the message body. If the message is not one of the supported types, or its body cannot be assigned to the specified type, or it has no body, then a `MessageFormatRuntimeException` is thrown.

These methods do not give access to the message headers or properties (such as the `JMSRedelivered` message header field or the `JMSXDeliveryCount` message property) and should only be used if the application has no need to access them.

If the next message is expected to be a `TextMessage` then this should be set to `String.class` or another class to which a `String` is assignable.

If the next message is expected to be a `ObjectMessage` then this should be set to `java.io.Serializable.class` or another class to which the body is assignable.

If the next message is expected to be a `MapMessage` then this should be set to `java.util.Map.class` (or `java.lang.Object`).

If the next message is expected to be a `BytesMessage` then this should be set to `byte[].class` (or `java.lang.Object`).

The result of this method throwing a `MessageFormatRuntimeException` depends on the session mode:

- `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`: The JMS provider will behave as if the unsuccessful call to `receiveBody` or `receiveBodyNoWait` had not occurred. The message will be delivered again before any subsequent messages.

  This is not considered to be redelivery and does not cause the `JMSRedelivered` message header field to be set or the `JMSXDeliveryCount` message property to be incremented.

- `CLIENT_ACKNOWLEDGE`: The JMS provider will behave as if the call to `receiveBody` or `receiveBodyNoWait` had been successful and will not deliver the message again.

  As with any message that is delivered with a session mode of `CLIENT_ACKNOWLEDGE`, the message will not be acknowledged until acknowledge is called on the `JMSContext`. If an application wishes to have the failed message redelivered, it must call recover on the `JMSContext`. The redelivered message's `JMSRedelivered` message header field will be set and its `JMSXDeliveryCount` message property will be incremented.

- Transacted session: The JMS provider will behave as if the call to `receiveBody` or `receiveBodyNoWait` had been successful and will not deliver the message again.

  As with any message that is delivered in a transacted session, the transaction will remain uncommitted until the transaction is committed or rolled back by the application. If an application wishes to have the failed message redelivered, it must roll back the transaction. The redelivered message's `JMSRedelivered` message header field will be set and its `JMSXDeliveryCount` message property will be incremented.

8.7. Receiving messages asynchronously

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

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

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

- `AUTO_ACKNOWLEDGE` 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 `RuntimeException` as possibly malfunctioning.

See Section 6.2.13 "Serial execution of client code" for information about how `onMessage` calls are serialized by a session.

8.8. Closing a consumer

The `close` methods on `MessageConsumer`, `JMSConsumer`, `QueueReceiver` and `TopicSubscriber` allow a consumer to be closed separately from the session or connection used to create it.

Closing a consumer terminates the delivery of messages to the consumer.

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

If `close` is called in one thread whilst another thread is calling `receive` on the same consumer then the call to `close` must block until the `receive` call has completed. A blocked `receive` call returns null when the consumer is closed.

If `close` is called in one thread whilst a message listener for this consumer is in progress in another thread then the call to `close` must block until the message listener has completed.

If `close` is called from a message listener's `onMessage` method on its own consumer then after this method returns the `onMessage` method must be allowed to complete normally.

Closing a consumer has no effect on the acknowledgement of messages delivered to the application, or on any transaction in progress. This is because message acknowledgement and transactions are functions of the session, not the consumer.

- If the session mode is AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE then any messages delivered to the application will be automatically acknowledged as normal.
- If the session mode is CLIENT_ACKNOWLEDGE then any messages delivered to the application may be acknowledged by calling `acknowledge` in the normal way. It makes no difference whether this occurs before or after the consumer is closed.
- If the session is transacted then the application may commit or rollback the transaction as normal. It makes no difference whether this occurs before or after the consumer is closed.

9. Other JMS facilities

9.1. Reliability

Most clients should use producers that produce `PERSISTENT` messages. This ensures 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 ensures 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.

9.2. Method inheritance across messaging domains

When JMS 1.1 unified the domain-specific APIs for point-to-point and pub/sub messaging into a single "unified" API (now referred to as the "classic" API), some methods that are not appropriate to a messaging domain became inherited by the domain-specific interfaces. For example, the `Session` interface has the method `createBrowser`. Since `TopicSession` inherits from the `Session` interface, `TopicSession` inherits the `createBrowser` method, though that method must not be used by a topic, as topics do not support queue browsers. Table 9-1outlines these instances.

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

Table 9-1 methods that throw an `IllegalStateException`

Interface	**Method**
`QueueConnection`	`createSharedConnectionConsumer`
	`createDurableConnectionConsumer`
	`createSharedDurableConnectionConsumer`
`QueueSession`	`createDurableSubscriber`
	`createDurableConsumer`
	`createSharedConsumer`
	`createSharedDurableConsumer`
	`createTemporaryTopic`
	`createTopic`
	`unsubscribe`
`TopicSession`	`createBrowser`
	`createQueue`
	`createTemporaryQueue`

10. JMS exceptions

10.1. Overview

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

10.2. JMSException and JMSRuntimeException

JMS defines two sets of exceptions:

- `JMSException` is the base class for all checked exceptions
- `JMSRuntimeException` is the base class for all unchecked exceptions.

In general, methods on interfaces defined in JMS 1.1 and earlier throw checked exceptions, whilst methods on the `JMSContext`, `JMSProducer` and `JMSConsumer` interfaces that were defined for the simplified API throw unchecked exceptions.

For those methods which throw checked exceptions, catching `JMSException` provides a generic way of handling all exceptions thrown by JMS.

Similarly, for those methods which throw unchecked exceptions only, catching `JMSRuntimeException` provides a generic way of handling all exceptions thrown by JMS. The Java language does not require unchecked exceptions to be explicitly caught by the application.

`JMSException` and `JMSRuntimeException` provide 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.

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

Methods which only throw unchecked exceptions do not include any exception in their signature. The javadoc for these methods documents the mandatory exception cases.

10.3. Standard exceptions

In addition to `JMSException and JMSRuntimeException`, 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. In most cases there is a checked exception (a subclass of `JMSException`) and a corresponding unchecked exception (a subclass of `JMSRuntimeException`). The unchecked version may only be thrown on those methods whose method signature does not permit the checked version to be thrown.

- `IllegalStateException` and `IllegalStateRuntimeException`. These exceptions are 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, `IllegalStateException` **must be** thrown if `Session.commit()` is called on a non-transacted session. `IllegalStateException` also **must be** called when a domain inappropriate method is called, such as calling `TopicSession.createBrowser()`.

- `JMSSecurityException` and `JMSSecurityRuntimeException`: These exceptions **must be** thrown when a provider rejects a user name/password submitted by a client. They may also be thrown for any case where a security restriction prevents a method from completing.

- `InvalidClientIDException` and `InvalidClientIDRuntimeException`: These exceptions **must be** thrown when a client attempts to set a connection's client identifier to a value that is rejected by a provider.

- `InvalidDestinationException` and `InvalidDestinationRuntimeException`: These exceptions **must be** thrown when a destination is either not understood by a provider or is no longer valid.

- `InvalidSelectorException and InvalidSelectorRuntimeException`: These exceptions **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` and `MessageFormatRuntimeException`: These exceptions **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. They must also be thrown when equivalent type errors are made with message property values. For example, a `MessageFormatException` **must be** thrown if `StreamMessage.writeObject()` is given an unsupported class or if `StreamMessage.readShort()` is used to read a boolean value. These exceptions 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` and `MessageNotWriteableRuntimeException`: These exceptions **must be** thrown when a JMS client attempts to write to a read-only message.

- `ResourceAllocationException` and `ResourceAllocationRuntimeException`: 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` and `TransactionInProgressRuntimeException`: These exceptions are 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` and `TransactionRolledBackRuntimeException`: A `TransactionRolledBackException` exception **must be** thrown when a call to `Session.commit()` results in a rollback of the current transaction. A `TransactionRolledBackRuntimeException` **must be** thrown when a call to `JMSContext.commit()` results in a rollback of the current transaction

11. JMS application server facilities

11.1. Overview

This chapter describes JMS facilities for concurrent processing of a subscription's messages. It also defines how a JMS provider supplies JTA aware sessions. These facilities are primarily intended for the use of the JMS provider.

If JMS clients use the JTA aware facilities the 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.

11.2. Concurrent processing of a subscription's messages

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

This facility partitions the work into three roles:

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

This facility requires the use of the classic API or the domain-specific APIs. It is not available in the simplified API. However since this facility is intended for use by application servers only this restriction does not affect applications.

11.2.1. Session

The `Session`, `QueueSession` and `TopicSession` objects provide the following methods for use by application servers:

- `setMessageListener()` and `getMessageListener()` - a session's `MessageListener` consumes messages that have been assigned to the session by a `ConnectionConsumer`, 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 `MessageListener`. When the listener returns from processing the last message, `run()` returns.

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

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

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

11.2.2. 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.

11.2.3. ServerSessionPool

A `ServerSessionPool` is an object implemented by an application server to provide a pool of `ServerSession` objects 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 `ServerSession` objects or it could use a sophisticated algorithm to dynamically create `ServerSession` objects as needed.

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

11.2.4. ConnectionConsumer

For application servers, the `Connection`, `QueueConnection` and `TopicConnection` objects provide a special method `createConnectionConsumer` 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.

11.2.5. 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 `Connection` will expect to load its messages into a `Session. 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.

11.2.6. 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.

11.2.7. The result

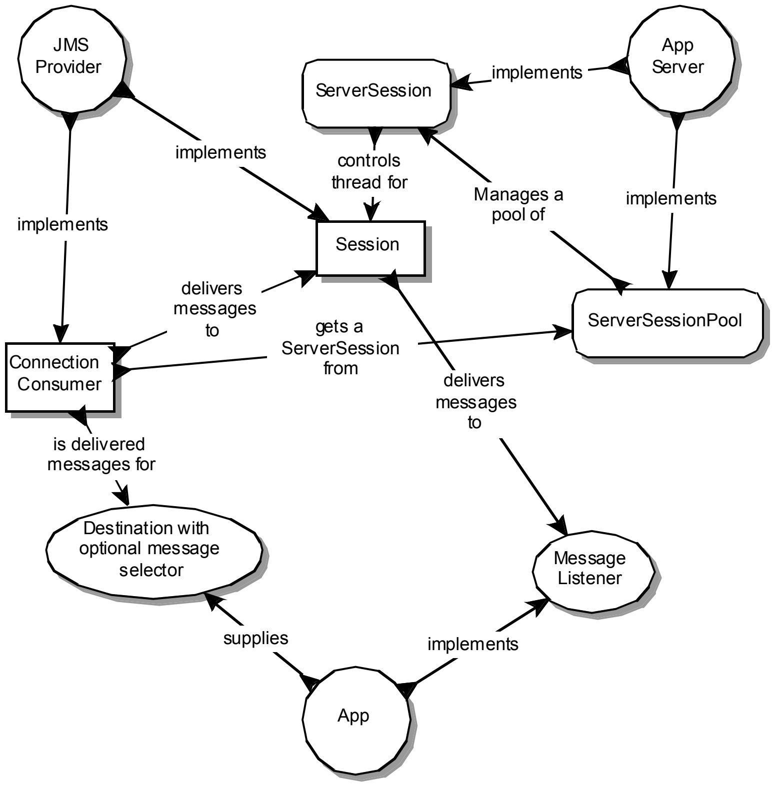
JMS has defined a flexible mechanism that partitions the job of concurrent message consumption into roles that are well suited for each participant.

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

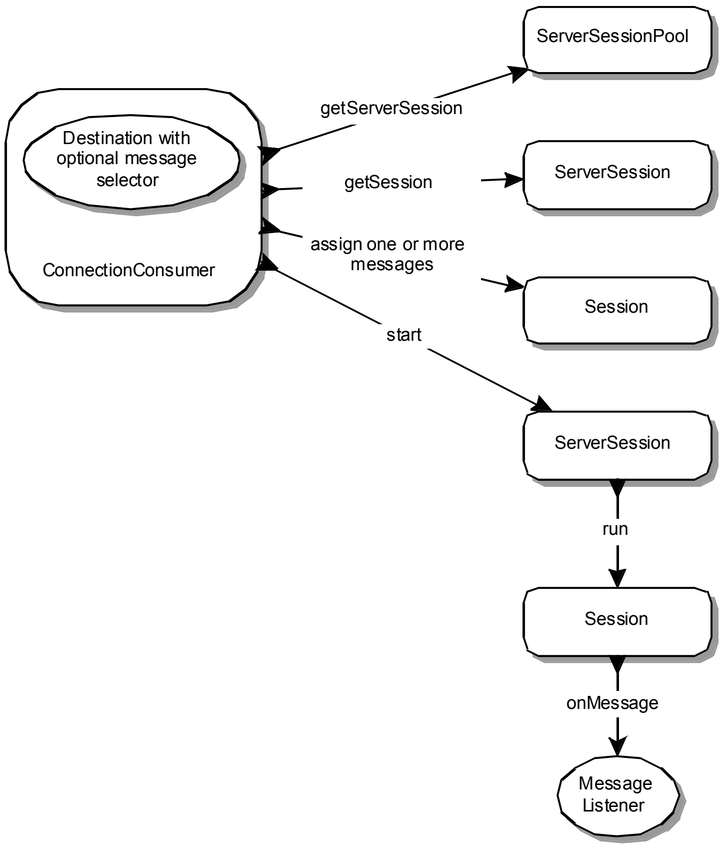
The JMS provider retains control of its messages until they are delivered to the `MessageListener`. This ensures it is under direct control of message acknowledgment.

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

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



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



11.3. Support for distributed transactions

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.

11.3.1. XA connection factory

A JMS provider exposes its JTA support using XA equivalents of the normal connection factory objects.

- For applications which use the classic or simplified APIs, a JMS provider exposes its JTA support using a JMS `XAConnectionFactory` which an application server uses to create `XAConnection` or `JMSXAContext` objects.
- For applications which use the domain-specific API for point-to-point messaging, a JMS provider exposes its JTA support using a JMS `XAQueueConnectionFactory` which the application server uses to create `XAQueueConnection` objects.
- For applications which use the domain-specific API for pub/sub messaging, a JMS provider exposes its JTA support using a JMS `XATopicConnectionFactory` which the application server uses to create `XATopicConnection` objects.

These connection factory objects provide the same authentication options as normal connection factory objects. They are JMS administered objects just like normal connection factory objects. It is expected that application servers will find them using JNDI.

11.3.2. XA connection

The XA connection objects extend the capability of normal connection objects by providing the ability to create XA session objects.

- An `XAConnection` provides the ability to create `XASession` objects.
- An `XAQueueConnection` provides the ability to create `XAQueueSession` objects.
- An `XATopicConnection` provides the ability to create `XATopicSession` objects.

11.3.3. XA session

The XA session objects (`XASession`, `XAQueueSession` and `XATopicSession`) provide access to what looks like a normal session object (a `Session`, `QueueSession` or `TopicSession`) and a `javax.transaction.xa.XAResource` object which controls its transaction context.

An application server controls the transactional assignment of an XA session object 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. A client of the application server is given the normal session object. Behind the scenes, the application server controls the transaction management of the underlying XA session object.

11.3.4. XAJMSContext

`XAJMSContext` provides access to what looks like a normal `JMSContext` object and a `javax.transaction.xa.XAResource` object which controls its transaction context.

An application server controls the transactional assignment of an `XAJMSContext` 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.

A client of the application server is given the `XAJMSContext`'s `JMSContext`. Behind the scenes, the application server controls the transaction management of the underlying `XAJMSContext`.

11.3.5. XAResource

The functionality of `XAResource` closely resembles that defined by the standard X/Open XA Resource interface.

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.

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.

11.4. JMS application server interfaces

The domain-specific APIs for point-to-point and pub/sub messaging provide their own versions of JTA aware JMS facilities.

However the classic API provides common interfaces, which should be used in preference to the domain-specific interfaces. These are listed as JMS common interfaces in Table 11-1.

Table 11-1 Relationship of optional interfaces in domains

Classic API	Domain-specific API for point-to-point messaging	Domain-specific API for pub/sub messaging
`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`
`XAJMSContext`	*Not domain-specific*	*Not domain-specific*

12. Use of JMS API in Java EE applications

12.1. 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.

12.2. 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 `JMSContext` object's `createContext` method then a `JMSRuntimeException` should be thrown, since the first `JMSContext` 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 `run`
- `javax.jms.Connection` method `createConnectionConsumer`
- `javax.jms.Connection` method `createSharedConnectionConsumer`
- `javax.jms.Connection` method `createDurableConnectionConsumer`
- `javax.jms.Connection` method `createSharedDurableConnectionConsumer`

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.JMSContext` method `setMessageListener`
- `javax.jms.JMSContext` method `getMessageListener`

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.JMSContext` method `setClientID`
- `javax.jms.JMSContext` method `stop`
- `javax.jms.JMSContext` method `setExceptionListener`

Applications which need to use a specific client identifier must set it on the connection factory, as described in section 6.1.2 "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.

12.3. 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 `JMSContext` objects, since these incorporate a JMS session.

When an application creates a `Session` or `JMSContext` 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 `JMSContext` 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 createContext 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 `Session` or `JMSContext` 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 `JMSContext` 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 `JMSContext` 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 `createContext(int sessionMode)`
- `javax.jms.ConnectionFactory` method `createContext(String userName, String password, int sessionMode)`

The following method to create a `JMSContext` from an existing `JMSContext` 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.JMSContext` method `createContext(int sessionMode)`

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

- `javax.jms.ConnectionFactory` method `createContext()`
- `javax.jms.ConnectionFactory` method `createContext(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 `JMSContext` 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 `JMSContext` is closed.

However, if a `Session` or `JMSContext` 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 `JMSContext` 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.

12.4. Injection of JMSContext objects

12.4.1. Support for injection

Injection of `JMSContext` objects is supported in those Java EE application classes which support dependency injection using CDI and for which CDI support has been enabled by means of a `META_INF/beans.xml` descriptor.

Section EE.5.24 of the Java EE specification lists the application classes that support dependency injection using CDI.

Section 12.1 of the CDI specification specifies how CDI support may be enabled for a particular application.

12.4.2. Container-managed and application-managed JMSContexts

A `JMSContext` object which has been injected is described as being *container-managed*, as it is created and closed by the container, not the application.

A `JMSContext` object which has been created by calling the `ConnectionFactory` method `createContext` is described as being *application-managed*. The application is responsible for calling the `close` method when the object is not longer needed.

12.4.3. Injection syntax

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

```
@Inject
private JMSContext context;
```

The container will inject a `JMSContext`. This object will have a scope as defined by section 12.4.4 "Scope of injected JMSContext objects".

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

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

If 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 `JMSContext` is created. For example:

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

Since it is undesirable to hardcode clear text passwords in an application, the password may be specified as an alias:

```
@Inject
@JMSPasswordCredential(
   username="admin", password="${ALIAS=myAdminPassword}")
private JMSContext context;
```

The use of a password alias allows the password to be defined in a secure manner separately from the application. See the Java EE 7 platform specification for more information on password aliases.

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

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

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

- In the Java EE application client container, session mode may be set to any of `JMSContext.SESSION_TRANSACTED`, `JMSContext.CLIENT_ACKNOWLEDGE`, `JMSContext.AUTO_ACKNOWLEDGE` or `JMSContext.DUPS_OK_ACKNOWLEDGE`. If no session mode is specified or the `JMSSessionMode` annotation is omitted a session mode of `JMSContext.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 `JMSContext.AUTO_ACKNOWLEDGE` or `JMSContext.DUPS_OK_ACKNOWLEDGE`. If no session mode is specified or the `JMSSessionMode` annotation is omitted a session mode of `JMSContext.AUTO_ACKNOWLEDGE` will be used.

For more information about the use of session mode when creating a messaging context, see section 12.3 "Behaviour of JMS sessions in the Java EE web or EJB container" and the API documentation for the `ConnectionFactory` method `createContext(int sessionMode)`.

12.4.4. Scope of injected JMSContext objects

The scope of an injected `JMSContext` defines whether different injected `JMSContext` objects will actually refer to the same `JMSContext` object

It also defines when the injected `JMSContext` will be closed by the container. When the object falls out of scope, the container will automatically call `close()`.

The scope of an injected `JMSContext` object will depend on whether the there is a JTA transaction in progress at the point where a particular method on the JMSContext is called.

- If a method is called on an injected `JMSContext` when there is a JTA transaction (either bean-managed or container-managed), the scope of the `JMSContext` will be `@TransactionScoped`. This scope is defined in the JTA specification. This means that:
  - The `JMSContext` object will be automatically created the first time a method on the `JMSContext` is called within the transaction.
  - The `JMSContext` object will be automatically closed when the transaction is committed or rolled back.
  - Within the same JTA transaction, `JMSContext` objects injected using identical annotations will refer to the same `JMSContext` object.
- If a method is called on an injected `JMSContext` when there is no JTA transaction then the scope of the `JMSContext` will be `@RequestScoped`. This scope is defined in the CDI specification. This means that:
  - The `JMSContext` object will be automatically created the first time a method on the `JMSContext` is called within a request.
  - The `JMSContext` object will be automatically closed when the request ends.
  - Within the same request, `JMSContext` objects injected using identical annotations will refer to the same `JMSContext` object.
- If a method is called on an injected `JMSContext` both in a JTA transaction and outside a JTA transaction then separate `JMSContext` objects will be used in each case, with a separate `JMSContext` object being used for each JTA transaction as described above.

12.4.5. Restrictions on use of injected JMSContext objects

Within the same scope, different injected `JMSContext` objects which are injected using identical annotations will all refer to the same `JMSContext` object.

This means that they will all use the same connection. This will reduce the resource usage of the application and improve performance.

It also means that messages would be sent using the same session. Messages sent using different JMSContext objects in the same scope will be therefore received in order in which they were sent (see section 6.2.9.2 "Order of message sends" for a few qualifications).

However, to avoid the possibility of code in one bean having an unexpected effect on a different bean, the following methods which change the public state of a `JMSContext` will not be permitted if the `JMSContext` is injected.

- `setClientID`
- `setExceptionListener`
- `stop`
- `acknowledge`

- `commit`
- `rollback`
- `recover`
- `setAutoStart`
- `start`
- `close`

These methods must throw a `IllegalStateRuntimeException` if the `JMSContext` is injected. These restrictions do not apply when the `JMSContext` is managed by the application; though note that several of these methods are in any case prohibited in a Java EE web or EJB container.

13. Resource adapter

The Java EE Connector Architecture (JCA) specification defines a standard architecture for connecting the Java EE platform to enterprise information systems (EISs).

A JMS provider (whether it forms part of a Java EE application server or not) is recommended to include a resource adapter which connects to that JMS provider and which conforms to the Java EE Connector Architecture specification and as further specified in this chapter.

13.1. MDB activation properties

Message-driven beans are defined in the Enterprise JavaBeans specification. JMS defines the following activation properties for message-driven beans.

Table 13-1 MDB activation properties defined by JMS

Activation property	**Description**
`destinationLookup`	This property may be used to specify the lookup name of an administratively-defined `javax.jms.Queue` or `javax.jms.Topic` object which defines the JMS queue or topic from which the endpoint (message-driven bean) is to receive messages.
`connectionFactoryLookup`	This property may be used to specify the lookup name of an administratively-defined `javax.jms.ConnectionFactory`, `javax.jms.QueueConnectionFactory` or `javax.jms.TopicConnectionFactory` object that will be used to connect to the JMS provider from which the endpoint (message-driven bean) is to receive messages.
`acknowledgeMode`	If bean-managed transaction demarcation is used, this property may be used to indicate whether JMS `AUTO_ACKNOWLEDGE` semantics or `DUPS_OK_ACKNOWLEDGE` semantics should apply. This property may be set to either `Auto-acknowledge` or `Dups-ok-acknowledge`. If this property is not specified, a default of `Auto-acknowledge` will be used.
`messageSelector`	This property may be used to specify a message selector. If this property is not specified then a message selector will not be used.
`destinationType`	This property may be used to specify whether the specified destination is a queue or topic. The valid values are `javax.jms.Queue` or `javax.jms.Topic`.

Activation property	Description
`subscriptionDurability`	This property only applies to endpoints (message-driven beans) that receive messages published to a topic. It may be used to specify whether the subscription is durable or non-durable. This property may be set to either `Durable` or `NonDurable`. If this property is not specified, a default of `NonDurable` will be used.
`clientId`	This property may be used to specify the client identifier that will be used when connecting to the JMS provider from which the endpoint (message-driven bean) is to receive messages. Setting this property is always optional.
`subscriptionName`	This property only applies to endpoints (message-driven beans) that receive messages published to a topic. It may be used to specify the name of the durable or non-durable subscription. It is not defined whether a shared or unshared subscription is used.

14. Examples of the classic API

This chapter gives some code examples that show how a JMS client could use the JMS classic API. It also demonstrates how to use several message types.

It is recommended that either the classic API or the simplified API be used in preference to the domain-specific APIs for point-to-point messaging. See also chapter 15 “Examples of the simplified API”.

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.

14.1. 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`

14.1.1. 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");
```

14.1.2. 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");
```

14.1.3. 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
}
```

14.1.4. 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 should 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);
```

14.1.5. 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);
```

14.1.6. Creating a MessageConsumer

Messages can be consumed either synchronously or asynchronously. This example shows how to create a message consumer that consumes messages synchronously. See section 14.3.1 "Receiving messages asynchronously" to learn more about consuming messages asynchronously.

A `MessageConsumer` is used to receive messages from the destination, which in this example is the `Queue stockQueue`. 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);
```

14.1.7. 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();
```

14.1.8. 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);
```

14.2. 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

14.2.1. 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);
```

14.2.2. 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.

```
Message stockMessage;
stockMessage = 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.

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

14.2.3. Unpacking a TextMessage

The stock quote information is sent using a `TextMessage`. There are two ways to extract the information from the message.

The `receive` method returns a `Message` object. You can cast this to a `TextMessage` and call the `getText` method. This returns the message content as a string:

```
String newStockData;
/* extract stock information from message */
newStockData = ((TextMessage)stockMessage)getText();
String newStockData;
```

Alternatively you can call the `Message` object's `getBody` method. In this case you do not need to cast the `Message` to a `TextMessage`. Instead you need to pass in the type expected:

```
String newStockData;
/* extract stock information from message */
newStockData = stockMessage.getBody(String.class);
```

14.3. 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

14.3.1. 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 = ((TextMessage)message).getText();
   if(...)
   {
      /* Logic related to the data */
   }
}
```

14.3.2. 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.

14.3.3. 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 topic provides information about news updates.

14.3.3.1. Creating a durable subscription

The following example sets up durable subscription that gets messages from a topic. First, the client program must perform the usual setup steps of looking up `ConnectionFactory` and a `Destination`, and creating a `Connection` and `Session`, as described in section 14.1 "Preparing to send and receive messages".

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

/* Look up connection factory */
ConnectionFactory connectionFactory;
```

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

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

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

14.3.3.2. Creating a consumer on an existing durable subscription

Once a durable subscription has been created it will continue to accumulate messages until the subscription is deleted using the `Session` method `unsubscribe`, even if the original consumer is closed leaving no consumer on the durable subscription.

A client application may create a consumer on an existing durable subscription by calling one of the `Session` methods `createDurableConsumer` or `createDurableSubscriber`, supplying the same parameters that were specified when the durable subscription was first created.

```
/* Create a consumer on an existing durable subscription */
session.createDurableConsumer(newsFeedTopic, "mySubscription");
```

If there were no consumers on the durable subscription prior to calling this method then any messages which were added to the subscription whilst it had no consumers will be delivered.

A durable subscription may have more than one active consumer (this was not permitted prior to JMS 2.0). Each message from the subscription will be delivered to only one of the consumers on that subscription.

When creating a consumer on an existing durable subscription there are some important restrictions to be aware of:

- The `Destination` and subscription name must be the same as when the durable subscription was first created.
- If the connection's client identifier was set when the durable subscription was first created then the same client identifier must be set when subsequently creating a consumer on it.

- If a message selector was specified when the durable subscription was first created then the same message selector must be specified when subsequently creating a consumer on it.

14.4. 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.

14.4.1. 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);
```

14.4.2. Unpacking a TextMessage

There are two ways to extract the text from a `TextMessage`. You can call the `getText` method on `TextMessage`:

```
String stockInto;
stockInto = stockMessage.getText();
String newStockData;
```

Alternatively you can call the `getBody` method on `Message`, which is the common supertype of all message types. In this case you need to pass in the body type expected:

```
String stockInto;
stockInto = stockMessage.getBody(String.class);
```

The use of `getBody` avoids the need to cast a newly-received `Message` object to a `TextMessage`.

14.4.3. 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);
```

14.4.4. Unpacking a BytesMessage

There are several ways to extract the byte array from a `BytesMessage`. The simplest is to call the `readBytes` method on `BytesMessage`. This copies the bytes to the specified byte array.

```
int bodyLength = message.getBodyLength();
byte[] stockData = new byte[bodyLength];
int bytesCopied = message.readBytes(stockData);
```

The `readBytes` method can also be used to read bytes in increments, by supplying a byte array whose length is less than the number of bytes available. The `readBytes` method will fill the array and set the return value to the number of bytes copied. A subsequent call reads the next increment and so on.

Alternatively you can call the `getBody` method on `Message`, which is the common supertype of all message types. In this case you need to pass in the body type expected. This method creates a byte array of the required size and copies all the bytes to it:

```
byte[] stockData = message.getBody(byte[].class);
```

The use of `getBody` avoids the need to cast a newly-received `Message` object to a `BytesMessage`.

14.4.5. 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 methods (`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");
```

14.4.6. Unpacking a MapMessage

There are two ways to extract body data from a MapMessage.

You can use 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");
```

Alternatively you can call the `getBody` method on `Message`, which is the common supertype of all message types. In this case you need to pass in the body type expected. This method returns a `java.util.Map` containing all the keys and values in the `MapMessage`.

```
Map stockData = message.getBody(Map.class);
stockName  = (String)stockData.getString("Name");
stockDiff  = (Double)stockData.getDouble("Diff");
stockValue = (Double)stockData.getDouble("Value");
stockTime  = (Long)stockData.getLong("Time");
```

The use of `getBody` avoids the need to cast a newly-received `Message` object to a `BytesMessage`.

If an application needs to get a list of the elements in a `MapMessage`, it can use the method `MapMessage.getMapNames`.

14.4.7. 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, it 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);
```

14.4.8. 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();
```

The `getBody` method cannot be used with a `StreamMessage`.

14.4.9. 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 send Java objects.

These values are set using methods that are unique to the `StockObject` implementation. For example, `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);
```

14.4.10. Unpacking an ObjectMessage

There are two ways to extract the object from an `ObjectMessage`. You can call the `getObject` method on `ObjectMessage`:

```
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();
```

Alternatively you can call the `getBody` method on `Message`, which is the common supertype of all message types. In this case you need to specify the object type expected:

```
StockObject stockObject;

stockObject = message.getBody(StockObject.class());

/* 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();
```

The use of `getBody` avoids the need to cast a newly-received Message object to an `ObjectMessage`. It also avoids the need to cast the object returned by `getObject` to the appropriate type.

15. Examples of the simplified API

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

15.1. 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.

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

15.1.2. 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 body) {

   try (JMSContext context = connectionFactory.createContext();){
      context.createProducer().send(inboundQueue,body);
   }
}
```

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

15.1.3. Example using the simplified API and injection

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

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

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

public void sendMessageNew(String body) {
   context.send(inboundQueue, body);
}
```

15.2. 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.

15.2.1. Example using the standard API

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

```
public void sendMessageOld(String body)
      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(body);
      messageProducer.send(textMessage);
   }
}
```

15.2.2. Example using the simplified API

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

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

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

   try (JMSContext context = connectionFactory.createContext();){
      context.createProducer().send(inboundQueue,body);
   }
}
```

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

15.3. Sending a message with properties (Java SE)

This example is similar to the previous example in that it compares the use of the standard and simplified JMS APIs for sending a `TextMessage` in a Java SE environment.

However this example also configures various attributes of the message that is sent:

- The message property `"foo"` is set to a value of `"bar"`.
- The message is sent using a delivery mode of `NON_PERSISTENT`.
- The JMS provider is informed that message timestamps are not required.

15.3.1. Example using the standard API

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

```
public void sendMessageOld(String body)
      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(body);
      textMessage.setStringProperty("foo", "bar");
      messageProducer.setDeliveryMode(NON_PERSISTENT);
      messageProducer.setDisableMessageTimestamp(true);
      messageProducer.send(textMessage);
   }
}
```

15.3.2. Example using the simplified API

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

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

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

   try (JMSContext context = connectionFactory.createContext();){
      context.createProducer().
              setProperty("foo", "bar").
              setTimeToLive(10000).
              setDeliveryMode(NON_PERSISTENT).
              setDisableMessageTimestamp(true).
              send(inboundQueue, body);
   }
}
```

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

15.4. 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.

15.4.1. 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 body = textMessage.getText();
      return body;
   }
}
```

15.4.2. 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 (JMSContext context = connectionFactory.createContext()) {
      JMSConsumer consumer = context.createConsumer(inboundQueue);
      return consumer.receiveBody(String.class);
   }
}
```

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

15.4.3. *Example using the simplified API and injection*

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

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

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

public String receiveMessageNew() {
   JMSConsumer consumer = context.createConsumer(inboundQueue);
   return consumer.receiveBody(String.class);
}
```

15.5. *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.

15.5.1. *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();
   }
}
```

15.5.2. 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 (JMSContext context =
         connectionFactory.createContext(AUTO_ACKNOWLEDGE);) {
      JMSConsumer consumer = context.createConsumer(inboundQueue);
      return consumer.receiveBody(String.class);
   }
}
```

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

15.6. 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.

15.6.1. 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/inboundTopic")
Topic inboundTopic;

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();
   }
}
```

15.6.2. *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/inboundTopic")
Topic inboundTopic;

public String receiveMessageNew() {

   try (JMSContext context = connectionFactory.createContext()) {
      JMSConsumer consumer =
         context.createDurableConsumer(inboundTopic, "mysub");
      return consumer.receiveBody(String.class);
   }
}
```

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

15.6.3. *Example using the simplified API and injection*

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

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

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

public String receiveMessageNew() {
    JMSConsumer consumer =
       context.createDurableConsumer(inboundTopic, "mysub");
    return consumer.receiveBody(String.class);
}
```

15.7. 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.

15.7.1. 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");
   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
    }
}
```

15.7.2. 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");
   Queue inboundQueue = (Queue)
      initialContext.lookup("jms/inboundQueue");

   try (JMSContext context =
         connectionFactory.createContext(AUTO_ACKNOWLEDGE);){
      JMSConsumer consumer = context.createConsumer(inboundQueue);
      MessageListener messageListener = new MyListener();
      consumer.setMessageListener(messageListener);

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

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

15.8. 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.

15.8.1. 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");
   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
   }
}
```

15.8.2. 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");
   Topic inboundTopic =
     (Topic) initialContext.lookup("jms/inboundTopic");

   try (JMSContext context =
         connectionFactory.createContext(AUTO_ACKNOWLEDGE);){
      JMSConsumer consumer =
         context.createDurableConsumer(inboundTopic, "mysub");
      MessageListener messageListener = new MyListener();
      consumer.setMessageListener(messageListener);

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

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

15.9. Receiving messages 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.

15.9.1. *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("One");
      messageConsumer1.setMessageListener(messageListener1);

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

      connection.start();

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

15.9.2. *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 (JMSContext context1 =
        connectionFactory.createContext(AUTO_ACKNOWLEDGE);
      JMSContext context2 =
         context1.createContext(AUTO_ACKNOWLEDGE);){
      JMSConsumer consumer1 =
         context1.createConsumer(inboundQueue);
      MyListener messageListener1 = new MyListener("One");
      consumer1.setMessageListener(messageListener1);

      JMSConsumer consumer2 =
         context2.createConsumer(inboundQueue);
      MyListener messageListener2 = new MyListener("Two");
      consumer2.setMessageListener(messageListener2);

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

15.10. 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`.

15.10.1. 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);
   }
}
```

15.10.2. 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 (JMSContext context =
         connectionFactory.createContext(SESSION_TRANSACTED);){
      JMSConsumer consumer = context.createConsumer(inboundQueue);
      TextMessage textMessage = null;
      do {
         textMessage = (TextMessage) consumer.receive(1000);
         if (textMessage != null) {
            context.createProducer().send(
               outboundQueue, textMessage);
            context.commit();
         }
      } while (textMessage != null);
   }
}
```

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

15.11. 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`.

15.11.1. 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 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
     senderBean.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
   }
}
}
```

15.11.2. 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 (JMSContext context = connectionFactory.createContext()) {
      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
      JMSConsumer consumer = context.createConsumer(replyQueue);
      return consumer.receiveBody(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 (JMSContext context = connectionFactory.createContext()) {
      TextMessage requestMessage =
         context.createTextMessage(requestString);
      // this call may throw JMSException
      requestMessage.setJMSReplyTo(replyQueue);
      context.createProducer().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 (JMSContext context = connectionFactory.createContext()){

      // 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.createProducer().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`.

15.11.3. Example using the simplified API and injection

Here's how the same example might look when using the simplified API with the `JMSContext` 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 JMSContext 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
   JMSConsumer consumer = context.createConsumer(replyQueue);
   return consumer.receiveBody(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 JMSContext 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.createProducer().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 JMSContext 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.createProducer().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. Change history

A.1. 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.

A.1.1 Reorganisation of chapters

This introduction of the simplified API in JMS 2.0 has necessitated a major reorganisation of this specification.

The structure of the JMS 1.1 specification reflected the domain-specific APIs introduced in JMS 1.0, with section titles such as “QueueConnection” and “TopicSubscriber. This was an inappropriate structure even in JMS 1.1 since these interfaces had been superseded in JMS 1.1 by the “unified” API. The addition of the simplified API in JMS 2.0 makes that structure even more inappropriate.

This version of the specification has therefore been completely restructured along functional lines, with chapter headings such as “connecting to a JMS provider” and “receiving messages”. These describe each area of functionality in generic terms followed by a description of how it is implemented in the various APIs. In general these chapters contain the same text as in the previous version.

In addition the following completely new chapters have been added:

- chapter 12 "Use of JMS API in Java EE applications"
- chapter 13 "Resource adapter"

A.1.2 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".

A.1.3 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 "Integration of JMS with the EJB components" has been deleted. It is superseded by the new chapter 12 "Use of JMS API in Java EE applications".

A.1.4 Resource adapter (JMS_SPEC-25)

A new chapter 13 “Resource adapter” has been added which recommends, but does not require, that a JMS provider (whether it forms part of a Java EE application server or not) includes a resource adapter which connects to that JMS provider and which conforms to the Java EE Connector Architecture specification.

A.1.5 MDB activation properties (JMS_SPEC-30, JMS_SPEC-54, JMS_SPEC-55)

A new section 13.1 “MDB activation properties” has been added which defines a set of activation properties for use with JMS message-driven beans.

- The `acknowledgeMode`, `messageSelector`, `destinationType`, `subscriptionDurability`, `clientId` and `subscriptionName` properties were previously defined in appendix B “Activation Configuration for Message Inflow to JMS Endpoints” in the Java EE Connector Architecture specification, version 1.6. Their definition has now have been moved to the JMS specification.

- The `connectionFactoryLookup` property is new and may be used to specify the the lookup name of an administratively-defined connection factory which will be used used by the MDB.

- The `destinationLookup` property is new and may be used to specify the the lookup name of an administratively-defined queue or topic from which the MDB will receive messages.

- The activation property `clientId` is now optional when using a durable subscription on a topic. This reflects the new shared durable subscriptions feature in JMS 2.0 which does not require clientId to be set.

These MDB activation properties are also defined in the Enterprise JavaBeans specification, version 3.2.

A.1.6 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 14.1.4 "Creating a Session" and 14.3.3.1 "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.

A.1.7 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.

A.1.8 Multiple consumers now allowed on the same topic subscription (JMS_SPEC-40)

In JMS 1.1, a durable or non-durable topic subscription was not permitted to have more than one consumer at a time. This meant that the work of processing messages on a subscription could not be shared amongst multiple threads, connections or JVMs, thereby limiting scalability. This restriction has therefore been removed in JMS 2.0.

A.1.8.1. Non-durable subscriptions

In order to maintain backwards compatibility with JMS 1.1, the existing methods for creating non-durable subscriptions remain unchanged. Subscriptions created using the existing `createConsumer` methods on `Session` and `TopicSession` and the existing `createSubscriber` methods on `TopicSession`, as well as the new `createConsumer` methods on `JMSContext`, will continue to be restricted to a single consumer and are now referred to as "unshared non-durable subscriptions". These are described in a new section 8.3.1 "Unshared non-durable subscriptions".

New `createSharedConsumer` methods have been added to `Session`, `TopicSession` and `JMSContext` to create a new type of non-durable subscription which may have more than one consumer. These are referred to as "shared non-durable subscriptions" and are identified by name and client identifier (if set). They are described in a new section 8.3.2 "Shared non-durable subscriptions". The `noLocal` parameter is not supported for shared non-durable subscriptions.

A.1.8.2. Durable subscriptions

In order to maintain backwards compatibility with JMS 1.1, the existing methods for creating durable subscriptions also remain unchanged. Subscriptions created using the existing `createDurableSubscriber` methods on `Session` and `TopicSession`, as well as the new `createDurableConsumer` methods on `Session`, `TopicSession` and `JMSContext`) will continue to be restricted to a single consumer and setting the client identifier will continue to be required. These now referred to as "unshared durable subscriptions" and are described in a new section 8.3.3 "Unshared durable subscriptions".

New `createSharedDurableConsumer` methods have been added to `Session`, `TopicSession` and `JMSContext` to create a new type of durable subscription which may have more than one consumer and which do not require the client identifier to be set. These are referred to as "shared durable subscriptions" and are described in a new section 8.3.4 “Shared durable subscriptions”. The `noLocal` parameter is not supported for shared durable subscriptions.

A.1.9 Client ID optional on shared durable subscriptions (JMS_SPEC-39)

In JMS 1.1 it was mandatory for the client identifier to be set when creating or activating a durable subscription.

In JMS 2.0, shared durable subscriptions (see A.1.8 above) will not have this restriction. However in order to maintain backwards compatibility with JMS 1.1, unshared durable subscriptions will continue to require the client identifier to be set.

A.1.10 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 7.9 "Message delivery delay" and a corresponding new section 3.4.13 "JMSDeliveryTime" have been added to describe this new feature. Section 6.2.9.2 "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 6.2.10 "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 7.1 "Producers" has been updated to mention that a client may now define a default delivery delay for messages sent by a producer.

A.1.11 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 `JMSContext` interface.

Section 7.1 "Producers" has been extended to describe these additional `send` methods..

A.1.12 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 `JMSContext` and `JMSConsumer` interfaces also extend the `java.lang.Autocloseable` interface.

Sections 6.1.8 "Closing a connection" and 6.2.15 "Closing a session" 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 14.1.3 "Creating a Connection" has been extended to add a second example which uses the the try-with-resources statement.

A.1.13 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 6.2.10 "Message acknowledgment": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well.

6.2.11 "Duplicate delivery of messages": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well..

9.1 "Reliability": A sentence which mentions the `JMSRedelivered` flag has been amended to mention the `JMSXDeliveryCount` property as well.

A.1.14 Simplified API (JMS_SPEC-64)

Three new objects `JMSContext`, `JMSProducer` and `JMSConsumer` 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".

`JMSContext` objects may be created using new methods on `ConnectionFactory`. Java EE applications may alternatively create `JMSContext` objects using injection.

The simplified API is described in section 2.8 “Simplified API interfaces”.

Developers now have a choice as to whether to use the "classic” API (the `Connection`, `Session`, `MessageProducer` and `MessageConsumer` objects) or the "simplified API" (the `JMSContext, JMSProducer` and `JMSConsumer` objects).

The two APIs are intended to offer similar functionality. The standard API is not deprecated and will remain part of JMS indefinitely.

Section 15 "Examples of the simplified API" contains a number of examples which compare the use of the simplified and standard APIs in a number of simple Java EE and Java SE use cases.

A.1.15 New method to extract the body directly from a Message (JMS_SPEC-101)

Two new methods have been added to `Message`:

- `<T> T getBody(Class<T> c)`
- `boolean isBodyAssignableTo(Class c)`

The `getBody` method returns the message body as an object of the specified type. This provides a convenient way to obtain the body from a newly-received `Message` object. It can be used either

- to return the body of a `TextMessage`, `MapMessage` or `BytesMessage` as a `String`, `Map` or `byte[]` without the need to cast the `Message` first to the appropriate subtype, or
- to return the body of an `ObjectMessage` without the need to cast the `Message` to `ObjectMessage`, extract the body as a `Serializable`, and cast it to the specified type.

The `isBodyAssignableTo` method is a companion method which can be used to determine whether a subsequent call to `getBody` would be able to return the body of a particular `Message` object as a particular type.

The example in section 14.2.3 "Unpacking a TextMessage" has been updated to demonstrate the use of the `getBody` method.

A.1.16 Subscription name characters and length

JMS 1.1 did not define what characters were valid in a durable subscription name, or what length of name was supported.

JMS 2.0 defines a minimum set of characters that must be valid in a durable or non-durable subscription name. It also defines that subscription names of up to 128 characters long must be supported.

For details see section 4.2.4 "Subscription name characters and length"

A.1.17 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 6.2.4 "Optimized message implementations" has been updated accordingly.

A.1.18 Clarification: use of ExceptionListener (JMS_SPEC-49)

Section 6.1.7 "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 `JMSContext` interface have been amended to clarify that, if an exception listener for the `JMSContext`'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.

A.1.19 Clarification: use of stop or close from a message listener (JMS_SPEC-48)

The specification has been clarified to clarify the required behaviour if various `stop` or `close` methods are called from within the `onMessage` method of a `MessageListener`.

The JMS 1.1 specification states that the `stop` method on `Connection` and the `close` methods on `Connection`, `Session` and `MessageConsumer` must not return until any message listeners have returned. This means that if these methods are called from a message listener on its own `Connection`, `Session` or `MessageConsumer` then deadlock would occur.

The JMS 2.0 specification amends the required behaviour to avoid the possibility of deadlock.

If a `MessageListener`'s `onMessage` method calls `stop` or `close` on its own `Connection`, `close` on its own `Session`, or `stop` or `close` on its own `JMSContext`, then the the JMS provider must throw a `javax.jms.IllegalStateException.`

However a different approach has been taken for the `close` methods on `MessageConsumer` and `JMSConsumer`. A `MessageListener`'s `onMessage` method is explicitly allowed to call `close` on its own `MessageConsumer` or `JMSConsumer`.

For details see the following sections:

- Section 6.1.5 "Pausing delivery of incoming messages"
- Section 6.1.8 "Closing a connection"
- Section 6.2.15 “Closing a session”
- Section 8.8 “Closing a consumer”.

The JMS 1.1 specification states that the close methods on `Connection` or `Session` are exempt from the requirement that the resources of a session may only be used by one thread at a time.

In JMS 2.0 this exemption also applies to the `close` method on `JMSContext`, and has been extended to cover the `close` methods on a `MessageConsumer` or `JMSConsumer`.

For details see the following sections:

- Section 6.2.5 “Threading restrictions on a session”
- Section 6.2.6 “Threading restrictions on a JMSContext”

A.1.20 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. This was poorly defined in JMS 1.1.

The new definition of `noLocal` is given in section 8.3.3 “Unshared durable subscriptions”. This states 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 or any other with the same client identifier will not be added to the durable subscription. It also states that if the client identifier is unset then setting `noLocal` to true will cause an exception to be thrown.

A.1.21 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 A.1.9).

Section 3.4.11 "How message header values are set" has been extended to explain this.

A.1.22 Clarification: Session methods createQueue and createTopic (JMS_SPEC-31)

The javadoc comments for the `createQueue` and `createTopic` methods on `Session` and `JMSContext` have been reworded to clarify that these methods simply create a `Queue` or `Topic` object which encapsulates the name of the queue or topic and do not create the physical queue or topic in the JMS provider.

In addition a note has been added to these javadoc comments to explain that although creating a physical queue or topic is provider-specific and is typically an administrative task performed by an administrator, some providers may create them automatically when needed.

A.1.23 Clarification: Definition of JMSExpiration (JMS_SPEC-82)

In the JMS 1.1 specification, section 3.4.9 "JMSExpiration", a message's expiration time was defined as "the sum of the time-to-live value specified on the send method and the current GMT value".

However the `JMSExpiration` header field is a `long` value and the specification does not define how the expiration time is converted to a `long`.

This has now been clarified to state that it is "the difference, measured in milliseconds, between the expiration time and midnight, January 1, 1970 UTC." This definition is chosen to be consistent with the `java.lang.System` method `currentTimeMillis`.

The updated text can be seen in section 3.4.9 "JMSExpiration" and section 7.8 "Message time-to-live".

A.1.24 Correction: Reconnecting to a durable subscription (JMS_SPEC-80)

In the JMS 1.1 specification, section 9.3.3.2 "Reconnect to a topic using a durable subscription" stated that "the client must be attached to the same Connection". This was incorrect and has now been corrected to state that the client must use a connection with the same client identifier.

In addition this section has been renamed 14.3.3.2 "Creating a consumer on an existing durable subscription" and rewritten to make it clearer.

A.1.25 Correction: MapMessage when name is null (JMS_SPEC-77)

In the JMS 1.1 API documentation for `javax.jms.MapMessage`, the method `setBytes (String name, byte[] value)` is defined as throwing a `NullPointerException` "if the name is null, or if the name is an empty string."

However there are eleven other methods on `MapMessage` of the form `setSomething(name,value)`. These all specify that a `IllegalArgumentException` is thrown "if the name is null or if the name is an empty string."

This appears to be an error in the API documentation. This is confirmed by the JMS compliance tests which already expect `setBytes` to throw a `IllegalArgumentException`.

The API documentation for `setBytes` has therefore been changed to match the other methods and specify that an `IllegalArgumentException` should be thrown in this case.